ACE

A GAP4 Package

Version 5.6.2
Based on ACE Standalone Version 3.001

by

George Havas and Colin Ramsay

Centre for Discrete Mathematics and Computing

Department of Information Technology and Electrical Engineering

The University of Queensland, St. Lucia 4072, Australia

GAP interface code by

Alexander Hulpke

Department of Mathematics, Colorado State University

Weber Building, Fort Collins, CO 80523, USA

and

Greg Gamble

Department of Mathematics and Statistics

Curtin University

GPO Box U 1987, Perth WA 6845, Australia

email: Greg.Gamble@uwa.edu.au

January 2025

Contents

1 The ACE Package

The "Advanced Coset Enumerator" ACE:

```
ACE coset enumerator (C) 1995-2001 by George Havas and Colin Ramsay
```

```
http://staff.itee.uq.edu.au/havas
```

can be called from within GAP through an interface, written by Alexander Hulpke and Greg Gamble, which is described in this manual.

The interface links to an external binary and therefore is only usable under UNIX (see Section 2.1 for how to install ACE). It will not work on Windows. The current version requires at least GAP 4.4.

- one may supplant the usual GAP coset enumerator (see Section 1.1),
- one may generate a coset table using ACE without redefining the usual GAP coset enumerator (see Section 1.2),
- one may simply test whether a coset enumeration will terminate (see Section 1.3),
- one may use GAP to write a script for the ACE standalone (see Section 1.4), and
- one may interact with the ACE standalone from within GAP (see Section 1.5). Among other things, the interactive ACE interface functions (described in Chapter 6) enable the user to search for subgroups of a group (see the note of Section 1.5).

Each of these ways gives the user access to a welter of options, which are discussed in full in Chapters 4 and 5. Yet more options are provided in Appendix D, but please take note of the Appendix's introductory paragraph. Check out Appendix B for numerous examples of the ACE commands.

Note: Some care needs to be taken with options; be sure to read Section 1.7 and the introductory sections of Chapter 4 for some warnings regarding them and a general discussion of their use, before using any of the functions provided by this interface to the ACE binary.

1.1 Using ACE as a Default for Coset Enumerations

After loading ACE (see Section 2.2), if you want to use the ACE coset enumerator as a default for all coset enumerations done by GAP (which may also get called indirectly), you can achieve this by setting the global variable `TCENUM` to `ACETCENUM`.

```
gap> TCENUM:=ACETCENUM;;
```

This sets the function `CosetTableFromGensAndRels` (see Section 47.6 in the GAP Reference Manual) to be the function `ACECosetTableFromGensAndRels` (described in Section 1.2), which then can be called with all the options defined for the ACE interface, not just the options `max` and `silent`. If `TCENUM` is set to `ACETCENUM` without any further action, the `default` strategy (option) of the ACE enumerator will be used (see Chapter 5). This and other options can be modified in the ways described in Chapter 4. For instance,

```
gap> FactorCosetAction(G,H : workspace:=10^9);
```

will result in ACE called with workspace of size 10^9, instead of the default value.

You can switch back to the coset enumerator built into the GAP library by assigning `TCENUM` to `GAPTCENUM`.

```
gap> TCENUM:=GAPTCENUM;;
```

1.2 Using ACE Directly to Generate a Coset Table

If, on the other hand you do not want to set up ACE globally for your coset enumerations, you may call the ACE interface directly, which will allow you to decide for yourself, for each such call, which options you want to use for running ACE. Please note the warnings regarding options in Section 1.7. The functions discussed in this and the following section (`ACECosetTableFromGensAndRels` and `ACEStats`) are non-interactive, i.e. by their use, a file with your input data in ACE readable format will be handed to ACE and you will get the answer back in GAP format. At that moment however the ACE job is terminated, that is, you cannot send any further questions or requests about the result of that job to ACE. For an interactive use of ACE from GAP see Section 1.5 and Chapter 6.

Using the ACE interface directly to generate a coset table is done by either of

1 ▶ `ACECosetTableFromGensAndRels(` *fgens*, *rels*, *sgens* `[:` *options*`] )` F
▶ `ACECosetTable(` *fgens*, *rels*, *sgens* `[:` *options*`] )` F

Here *fgens* is a list of free generators, *rels* a list of words in these generators giving relators for a finitely presented group, and *sgens* the list of subgroup generators, again expressed as words in the free generators. All these are given in the standard GAP format (see Chapter 47 of the GAP Reference Manual). Note that the 3-argument form of `ACECosetTable` described here is merely a synonym for `ACECosetTableFromGensAndRels`, and that `ACECosetTable` may be called in a different way in an interactive ACE session (see Sections 1.5 and 6.6.1).

Behind the colon any selection of the options available for the interface (see Chapters 4 and 5) can be given, separated by commas like record components. These can be used e.g. to preset limits of space and time to be used, to modify input and output and to modify the enumeration procedure. Note that strategies are simply special options that set a number of the options, detailed in Chapter 4, all at once.

Please see Section 1.7 for a discussion regarding global and local passing of options, and the non-orthogonal nature of ACE's options.

Each of `ACECosetTableFromGensAndRels` and `ACECosetTable` calls the ACE binary and, if successful, returns a standard coset table, as a GAP list of lists. At the time of writing, two coset table standardisations schemes were possible: `lenlex` and `semilenlex` (see Section 3.4). The user may control which standardisation scheme is used by selecting either the `lenlex` (see 4.11.4) or `semilenlex` (see 4.11.5) option; otherwise the table is standardised according to GAP's the value of `CosetTableStandard` (which by default is `lenlex`). We provide `IsACEStandardCosetTable` (see 1.2.2) to determine whether a table (list of lists) is standard relative to GAP's default standardisation scheme, or with the use of options (e.g. `lenlex` or `semilenlex`) to another standardisation scheme.

If the determination of a coset table is unsuccessful, then one of the following occurs:

- with the `incomplete` option (see 4.11.6) an incomplete coset table is returned (as a list of lists), with zeros in positions where valid coset numbers could not be determined; or
- with the `silent` option (see 4.11.3), `fail` is returned; or
- a `break`-loop is entered. This last possibility is discussed in detail via the example that follows.

The example given below is the call for a presentation of the Fibonacci group F(2,7) for which we shall discuss the impact of various options in Section B.4. Observe that in the example, no options are passed, which means that ACE uses the `default` strategy (see Chapter 5).

```
gap> F:= FreeGroup( "a", "b", "c", "d", "e", "x", "y");;
gap> a:= F.1;; b:= F.2;; c:= F.3;; d:= F.4;; e:= F.5;; x:= F.6;; y:= F.7;;
gap> fgens:= [a, b, c, d, e, x, y];;
gap> rels:= [ a*b*c^-1, b*c*d^-1, c*d*e^-1, d*e*x^-1,
>             e*x*y^-1, x*y*a^-1, y*a*b^-1];;
gap> ACECosetTable(fgens, rels, [c]);;
```

In computing the coset table, `ACECosetTableFromGensAndRels` must first do a coset enumeration (which is where ACE comes in!). If the coset enumeration does not finish in the preset limits a `break`-loop is entered, unless the `incomplete` (see 4.11.6) or `silent` (see 4.11.3) options is set. In the event that a `break`-loop is entered, don't despair or be frightened by the word `Error`; by tweaking ACE's options via the `SetACEOptions` function that becomes available in the `break`-loop and then typing `return;` it may be possible to help ACE complete the coset enumeration (and hence successfully compute the coset table); if not, you will end up in the `break`-loop again, and you can have another go (or `quit;` if you've had enough). The `SetACEOptions` function is a no-argument function; it's there **purely** to pass **options** (which, of course, are listed behind a colon (`:`) with record components syntax). Let's continue the Fibonacci example above, redoing the last command but with the option `max := 2` (see 4.17.6), so that the coset enumeration has only two coset numbers to play with and hence is bound to fail to complete, putting us in a `break`-loop.

```
gap> ACECosetTable(fgens, rels, [c] : max := 2);
Error, no coset table ...
 the 'ACE' coset enumeration failed with the result:
 OVERFLOW (a=2 r=1 h=1 n=3; l=5 c=0.00; m=2 t=2)
 called from
<function "ACECosetTable">( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
 try relaxing any restrictive options
 e.g. try the 'hard' strategy or increasing 'workspace'
 type: '?strategy options' for info on strategies
 type: '?options for ACE' for info on options
 type: 'DisplayACEOptions();' to see current ACE options;
 type: 'SetACEOptions(:<option1> := <value1>, ...);'
 to set <option1> to <value1> etc.
 (i.e. pass options after the ':' in the usual way)
 ... and then, type: 'return;' to continue.
 Otherwise, type: 'quit;' to quit to outer loop.
brk> SetACEOptions(: max := 0);
brk> return;
[ [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ],
  [ 1 ], [ 1 ], [ 1 ], [ 1 ] ]
```

Observe how the lines after the "`Entering break read-eval-print loop`" announcement tell you **exactly** what to do. At the `break`-loop prompt `brk>` we relaxed all restrictions on `max` (by re-setting it to 0) and typed `return;` to leave the `break`-loop. The coset enumeration was then successful, allowing the computation of what turned out to be a trivial coset table. Despite the fact that the eventual coset table only has one line (i.e. there is exactly one coset number) ACE **did** need to define more than 2 coset numbers. To find out just how many were required before the final collapse, let's set the `InfoLevel` of `InfoACE` (see 1.9.3) to 2, so that the ACE enumeration result is printed.

```
gap> SetInfoACELevel(2);
gap> ACECosetTable(fgens, rels, [c]);
#I  INDEX = 1 (a=1 r=2 h=2 n=2; l=6 c=0.00; m=2049 t=3127)
[ [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ],
  [ 1 ], [ 1 ], [ 1 ], [ 1 ] ]
```

The enumeration result line is the `Info` line beginning "`#I `". Appendix A explains how to interpret such output messages from ACE. In particular, it explains that `t=3127` tells us that a `t`otal number of 3127 coset numbers needed to be defined before the final collapse to 1 coset number. Using some of the many options that ACE provides, one may achieve this result more efficiently, e.g. with the `purec` strategy (see 5.1.6):

```
gap> ACECosetTable(fgens, rels, [c] : purec);
#I  INDEX = 1 (a=1 r=2 h=2 n=2; l=4 c=0.00; m=332 t=332)
[ [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ],
  [ 1 ], [ 1 ], [ 1 ], [ 1 ] ]
```

ACE needs to define a total number of only (relatively-speaking) 332 coset numbers before the final collapse to 1 coset number.

Notes: To initiate the coset enumeration, the `start` option (see D.3.2) is quietly inserted after the user's supplied options, unless the user herself supplies one of the enumeration-invoking options, which are: `start`, or one of its synonyms, `aep` (see D.1.1) or `rep` (see D.1.2).

When a user calls `ACECosetTable` with the `lenlex` option (see 4.11.4), occasionally it is necessary to enforce `asis` = 1 (see 4.13.1), which may be counter to the desires of the user. The occasions where this action is necessary are precisely those for which, for the arguments *gens* and *rels* of `ACECosetTable`, `IsACEGeneratorsInPreferredOrder` would return `false`.

The non-interactive `ACECosetTable` and `ACECosetTableFromGensAndRels` now use an iostream to communicate with the ACE binary in order to avoid filling up a temporary directory with an incomplete coset table in the case where an enumeration overflows. This is generally advantageous. However, on some systems, it may **not** be advisable to repeatedly call `ACECosetTable` or `ACECosetTableFromGensAndRels` (e.g. in a loop), since one may run out of the pseudo ttys used for iostreams. If you encounter this problem, consider using an adaptation of the usage of the interactive forms of `ACECosetTable` and `ACEStats` (see 6.6.1 and 6.6.2), together with `ACEStart` initialisation steps, that is sketched in the schema below. For the following code, it is imagined the scenario is one of looping over several possibilities of *fgens*, *rels* and *sgens*; the two special forms of `ACEStart` used, allow one to continually re-use a single interactive ACE process (i.e. only **one** iostream is used).

```
# start the interactive ACE process with no group information
procId := ACEStart(0);
while expr do
  fgens := ...; rels := ...; sgens := ...;
  ACEStart(procId, fgens, rels, sgens : options);
  if ACEStats(procId).index > 0 then
    table := ACECosetTable(procId);
    ...
  fi;
od;
```

For an already calculated coset table, we provide the following function to determine whether or not it has been standardised.

2 ► `IsACEStandardCosetTable(` *table* `[:` *option*`] )` F

returns `true` if *table* (a list of lists of integers) is standard according to GAP's default or the *option* (either `lenlex` or `semilenlex`) standardisation scheme, or `false` otherwise. See Section 3.4 for a detailed discussion of the `lenlex` and `semilenlex` standardisation schemes.

Note: Essentially, `IsACEStandardCosetTable` extends the GAP function `IsStandardized`.

Users who wish their coset tables to use `ACECosetTable` with the `lenlex` (see 4.11.4) option, which causes `lenlex` standardisation to occur at the ACE (rather than GAP) level, should be aquainted with the following function.

3 ► `IsACEGeneratorsInPreferredOrder(` *gens*`,` *rels* `)` F

returns `true` if *gens*, a list of free group generators, are in an order which will not be changed by ACE, for the group with presentation $\langle gens \mid rels \rangle$, where *rels* is a list of relators (i.e. words in the generators *gens*). `IsACEGeneratorsInPreferredOrder` may also be called in a different way for interactive ACE processes (see 6.6.3).

For a presentation with more than one generator, the first generator of which is an involution and the second is not, ACE prefers to switch the first two generators. `IsACEGeneratorsInPreferredOrder` returns `true` if the order of the generators *gens* would not be changed within ACE and `false`, otherwise. (Technically, by "involution" above, we really mean "a generator `x` for which there is a relator in *rels* of form `x*x` or `x^2`". Such a generator may, of course, turn out to actually be the identity.)

Guru Notes: If `IsACEGeneratorsInPreferredOrder(`*gens*`, `*rels*`)` would return `false`, it is possible to enforce a user's order of the generators within ACE, by setting the option `asis` (see 4.13.1) to 1 and, by passing the relator that determines that *gens*`[1]` (which we will assume is `x`) has order at most 2, as: `x*x` (rather than `x^2`). Behind the scenes this is precisely what is done, if necessary, when `ACECosetTable` is called with the `lenlex` option.

The user may avoid all the technicalities by either not using the `lenlex` option (and allowing GAP to take care of the `lenlex` standardisation), or by swapping the first two generators in those cases where `IsACEGeneratorsInPreferredOrder(`*gens*`, `*rels*`)` would return `false`.

1.3 Using ACE Directly to Test whether a Coset Enumeration Terminates

If you only want to test, whether a coset enumeration terminates, and don't want to transfer the whole coset table to GAP, you can call

1 ► `ACEStats(` *fgens*`, `*rels*`, `*sgens* `[:` *options*`] )` F

which calls ACE non-interactively to do the coset enumeration, the result of which is parsed and returned as a GAP record with fields

`index`
: the index of the subgroup $\langle sgens \rangle$ in $\langle fgens \mid rels \rangle$, or 0 if the enumeration does not succeed;

`cputime`
: the total CPU time used as an integer number of `cputimeUnits` (the next field);

`cputimeUnits`
: the units of the `cputime` field, e.g. `"10^-2 seconds"`;

`activecosets`
: the number of currently **active** (i.e. **alive**) coset numbers (see Section 3.5);

`maxcosets`
: the **maximum** number of alive coset numbers at any one time in the enumeration (see Section 3.5); and

`totcosets`
: the **total** number of coset numbers that were defined in the enumeration (see Section 3.5).

Options (see Chapters 4 and 5) are used in exactly the same way as for `ACECosetTableFromGensAndRels`, discussed in the previous section; and the same warnings alluded to previously, regarding options (see Section 1.7), apply.

Notes: To initiate the coset enumeration, the `start` option (see D.3.2) is quietly inserted after the user's supplied options, unless the user herself supplies one of the enumeration-invoking options, which are: `start`, or one of its synonyms, `aep` (see D.1.1) or `rep` (see D.1.2).

The fields of the `ACEStats` record are determined by parsing a "results message" (see Appendix A) from ACE.

`ACEStats` may also be called in a different way in an interactive ACE session (see 6.6.2).

1.4 Writing ACE Standalone Input Files to Generate a Coset Table

If you want to use ACE as a standalone with its own syntax, you can write an ACE standalone input file by calling `ACECosetTable` with three arguments (see 1.2.1) and the option `aceinfile := `*filename* (see 4.11.7). This will keep the input file for the ACE standalone produced by the GAP interface under the file name *filename* (and just return) so that you can perform interactive work in the standalone.

1.5 Using ACE Interactively

An interactive ACE process is initiated with the command

1 ► `ACEStart(` *fgens*, *rels*, *sgens* [:*options*] `)` F

whose arguments and options are exactly as for `ACECosetTableFromGensAndRels` and `ACEStats`, as discussed in Sections 1.2 and 1.3. The usual warnings regarding options apply (see Section 1.7). `ACEStart` has a number of other forms (see 6.1.1).

The return value is an integer (numbering from 1) which represents the running process. (It is possible to have more than one interactive process running at once.) The integer returned may be used to index which of these processes an interactive ACE function should be applied to.

An interactive ACE process is terminated with the command

2 ► `ACEQuit(` *i* `)` F

where *i* is the integer returned by `ACEStart` when the process was begun. `ACEQuit` may also be called with no arguments (see 6.1.2).

We discuss each of these commands, as well as the range of functions which enable one to access features of the ACE standalone not available non-interactively, in depth, in Chapter 6.

Note:

ACE not only allows one to do a coset enumeration of a group by a given (and then fixed) subgroup but it also allows one to search for subgroups by starting from a given one (possibly the trivial subgroup) and then augmenting it by adding new subgroup generators either explicitly via `ACEAddSubgroupGenerators` (see 6.7.4) or implicitly by introducing **coincidences** (see `ACECosetCoincidence`: 6.7.7, or `ACERandomCoincidences`: 6.7.8); or one can find smaller subgroups by deleting subgroup generators via `ACEDeleteSubgroupGenerators` (see 6.7.6).

1.6 Accessing ACE Examples with ACEExample and ACEReadResearchExample

There are a number of examples available in the `examples` directory, which may be accessed via

1 ► `ACEExample()` F
► `ACEExample(` *examplename* [:*options*] `)` F
► `ACEExample(` *examplename*, *ACEfunc* [:*options*] `)` F

where *examplename* is a string, the name of an example (and corresponding file in the `examples` directory); and *ACEfunc* is the ACE function with which the example is to be executed.

If `ACEExample` is called with no arguments, or with the argument: `"index"` (meant in the sense of "list"), or with a non-existent example name, a list of available examples is displayed. See Section B.4 where the list is displayed.

By default, examples are executed via `ACEStats`. However, if `ACEExample` is called with a second argument (choose from the (other) alternatives: `ACECosetTableFromGensAndRels` (or, equivalently `ACECosetTable`), or `ACEStart`), the example is executed using that function, instead. Note that, whereas the first argument appears in double quotes (since it's a string), the second argument does not (since it's a function); e.g. to execute example `"A5"` with function `ACECosetTable`, one would type: `ACEExample("A5", ACECosetTable);`.

`ACEExample` also accepts user options, which may be passed either globally (i.e. by using `PushOptions` to push them onto the `OptionsStack`) or locally behind a colon after the `ACEExample` arguments, and they are passed to `ACEStats` or *ACEfunc* as if they were **appended** to the existing options of *examplename*; in this way, the user may **over-ride** any or all of the options of *examplename*. This is done by passing an option `aceexampleoptions` (see 4.11.15), which sets up a mechanism to reverse the usual order in which options of recursively called functions are pushed onto the `OptionsStack`. The option `aceexampleoptions` is **not** a user option; it is intended only for **internal** use by `ACEExample`, for the above purpose. In the portion of the output due to the `echo` option, if one has passed options to

`ACEExample`, one will see `aceexampleoptions` listed first and the result of the interaction of *examplename*'s options and the additional options.

Consider the example `"A5"`. The effect of running

```
gap> ACEExample("A5", ACEStart);
```

is essentially equivalent to executing:

```
gap> file := Filename(DirectoriesPackageLibrary("ace", "examples"), "A5");;
gap> ACEfunc := ACEStart;;
gap> ReadAsFunction(file)();
```

except that some internal "magic" of `ACEExample` edits the example file and displays equivalent commands a user "would" execute. If the user has passed options to `ACEExample` these appear in a "`# User Options`" block after the original options of the example in the `Info` portion of the output. By comparing with the portion of the output from the `echo` option (unless the user has over-ridden the `echo` option), the user may directly observe any over-riding effects of user-passed options.

Please see Section B.4 for some sample interactions with `ACEExample`.

Notes

Most examples use the `mess` (= `messages`) option. To see the effect of this, it is recommended to do: `SetInfoACELevel(3);` before calling `ACEExample`, with an example.

The coset table output from `ACEExample`, when called with many of the examples and with the ACE function `ACECosetTableFromGensAndRels` is often quite long. Recall that the output may be suppressed by following the (`ACEExample`) command with a double semicolon (`;;`).

Also, try `SetInfoACELevel(2);` before calling `ACEExample`, with an example.

If you unexpectedly observe `aceexampleoptions` in your output, then most probably you have unintentionally passed options by the global method, by having a non-empty `OptionsStack`. One possible remedy is to use `FlushOptionsStack();` (see 4.2.1), before trying your `ACEExample` call again.

As discussed in Section 4.6, there is generally no sensible meaning that can be attributed to setting a strategy option (see Chapter 5) to `false`; if you wish to nullify the effect of a strategy option, pass another strategy option, e.g. pass the `default` (see 5.1.1) strategy option.

Also provided are:

2 ► `ACEReadResearchExample(` *filename* `)` F
► `ACEReadResearchExample()` F

which perform `Read` (see Section 9.7.1 in the GAP Reference Manual) on *filename* or, with no argument, the file with filename `"pgrelfind.g"` in the `res-examples` directory; e.g. the effect of running

```
gap> ACEReadResearchExample("pgrelfind.g");
```

is equivalent to executing:

```
gap> Read( Filename(DirectoriesPackageLibrary("ace", "res-examples"),
>                   "pgrelfind.g") );
```

The examples provided in the `res-examples` directory were used to solve a genuine research problem, the results of which are reported in [CHHR02]. Please see Section B.5 for a detailed description and examples of its use.

3 ► `ACEPrintResearchExample(` *example-filename* `)` F
► `ACEPrintResearchExample(` *example-filename*`,` *output-filename*`)` F

print the "essential" contents of the file *example-filename* in the `res-examples` directory to the terminal, or with two arguments to the file *output-filename*; *example-filename* and *output-filename* should be strings. `ACEPrintResearchExample` is provided to make it easy for users to copy and edit the examples for their own purposes.

1.7 General Warnings regarding the Use of Options

Firstly, let us mention (and we will remind you later) that an ACE strategy is merely a special option of ACE that sets a number of the options described in Chapter 4 all at once. The strategy options are discussed in their own chapter (Chapter 5).

In Section 4.1, we describe the two means provided by GAP by which ACE options may be passed. In Section 4.2, we discuss how options "left over" from previous calculations can upset subsequent calculations; and hence, to "clear the decks" in such circumstances, why we have provided `FlushOptionsStack` (see 4.2.1). However, removing `OptionsStack` options does not remove the options previously passed to an **interactive** ACE process; Section 4.2 discusses that too.

Note, that the ACE package treatment of options is an "enhancement" over the general way GAP treats options. Firstly, the ACE binary allows abbreviations and mixed case of options and so the ACE package does also, as much as is possible (see 4.3). Secondly, since ACE's options are in general not orthogonal, the order in which they are put to ACE is, in general, honoured (see 4.4). Thirdly, the ACE binary's concept of a boolean option is slightly different to that of GAP's; Section 4.6 discusses, in particular, how an option detected by GAP as `true` is passed to the ACE binary.

Finally, Section 4.5 discusses what happens if no option is selected.

1.8 The ACEData Record

1 ► `ACEData` V

is a GAP record in which the essential data for an ACE session within GAP is stored; its fields are:

`binary`
: the path of the ACE binary;

`tmpdir`
: the path of the temporary directory containing the non-interactive ACE input and output files (also see 1.8.2 below);

`ni`
: the data record for a non-interactive ACE process;

`io`
: list of data records for `ACEStart` (see below and 6.1.1) processes;

`infile`
: the full path of the ACE input file;

`outfile`
: the full path of the ACE output file; and

`version`
: the version of the current ACE binary. More detailed information regarding the compilation of the binary is given by `ACEBinaryVersion` (see 6.5.25).

Non-interactive processes used to use files rather than streams (hence the fields `infile` and `outfile` above; these may disappear in a later version of the ACE package).

Each time an interactive ACE process is initiated via `ACEStart` (see 6.1.1), an identifying number *ioIndex* is generated for the interactive process and a record `ACEData.io[`*ioIndex*`]` with the following fields is created. A non-interactive process has similar fields but is stored in the record `ACEData.ni`.

`procId`
: the identifying number of the process (*ioIndex* for interactive processes, and 0 for a non-interactive process);

`args`
: a record with fields: `fgens`, `rels`, `sgens` whose values are the corresponding arguments passed originally to `ACEStart`;

`options`
: the current options record of the interactive process;

`acegens`
: a list of strings representing the generators used by ACE (if the names of the generators passed via the first argument *fgens* of `ACEStart` were all lowercase alphabetic characters, then `acegens` is the `String` equivalent of *fgens*, i.e. `acegens[1] = String(`*fgens*`[1])` etc.);

`stream`
: the IOStream opened for an interactive ACE process initiated via `ACEStart`; and

`enumResult`
: the enumeration result (string) without the trailing newline, output from ACE;

`stats`
: a record as output by the function `ACEStats`.

`enforceAsis`
: a boolean that is set to `true` whenever the `asis` option (see 4.13.1) must be set to `1`. It is usually `false`. See 1.2.3 **Guru Notes** for the details.

2 ► `ACEDirectoryTemporary(` *dir* `)` F

calls the UNIX command `mkdir` to create *dir*, which must be a string, and if successful a directory object for *dir* is both assigned to `ACEData.tmpdir` and returned. The fields `ACEData.infile` and `ACEData.outfile` are also set to be files in `ACEData.tmpdir`, and on exit from GAP *dir* is removed. Most users will never need this command; by default, GAP typically chooses a "random" subdirectory of `/tmp` for `ACEData.tmpdir` which may occasionally have limits on what may be written there. `ACEDirectoryTemporary` permits the user to choose a directory (object) where one is not so limited.

1.9 Setting the Verbosity of ACE via Info and InfoACE

1 ► `InfoACE` V

The output of the ACE binary is, by default, not displayed. However the user may choose to see some, or all, of this output. This is done via the `Info` mechanism (see Chapter 7.4 in the GAP Reference Manual). For this purpose, there is the *InfoClass* `InfoACE`. Each line of output from ACE is directed to a call to `Info` and will be displayed for the user to see if the `InfoLevel` of `InfoACE` is high enough. By default, the `InfoLevel` of `InfoACE` is 1, and it is recommended that you leave it at this level, or higher. Only messages which we think that the user will really want to see are directed to `Info` at `InfoACE` level 1. To turn off **all** `InfoACE` messaging, set the `InfoACE` level to 0 (see 1.9.3). For convenience, we provide the function

2 ► `InfoACELevel()` F

which returns the current `InfoLevel` of `InfoACE`, i.e. it is simply a shorthand for `InfoLevel(InfoACE)`.

To set the `InfoLevel` of `InfoACE` we also provide

3 ► `SetInfoACELevel(` *level* `)` F
► `SetInfoACELevel()` F

which for a non-negative integer *level*, sets the `InfoLevel` of `InfoACE` to *level* (i.e. it is a shorthand for `SetInfoLevel(InfoACE,` *level*`)`), or with no arguments sets the `InfoACE` level to the default value 1. Currently, information from ACE is directed to `Info` at four `InfoACE` levels: 1, 2, 3 and 4. The command

```
gap> SetInfoACELevel(2);
```

enables the display of the results line of an enumeration from ACE, whereas

```
gap> SetInfoACELevel(3);
```

enables the display of all of the output from ACE (including ACE's banner, containing the host machine information); in particular, the progress messages, emitted by ACE when the `messages` option (see 4.18.1) is set to a non-zero value, will be displayed via `Info`. Finally,

```
gap> SetInfoACELevel(4);
```

enables the display of all the input directed to ACE (behind a "`ToACE> `" prompt, so you can distinguish it from other output). The `InfoACE` level of 4 is really for gurus who are familiar with the ACE standalone.

1.10 Acknowledgements

Large parts of this manual, in particular the description of the options for running ACE, are directly copied from the respective descriptions in the manual [Ram99] for the standalone version of ACE by Colin Ramsay. Most of the examples, in the `examples` directory and accessed via the `ACEExample` function, are direct translations of Colin Ramsay's `test###.in` files in the `src` directory.

Many thanks to Joachim Neubüser who not only provided one of the early manual drafts and hence a template for the style of the manual and conceptual hooks for the design of the Package code, but also meticulously proof-read all drafts and made many insightful comments.

Many thanks also to Volkmar Felsch who, in testing the ACE Package, discovered a number of bugs, made a number of important suggestions on code improvement, thoroughly checked all the examples, and provided the example found at the end of Section 6.7 which demonstrates rather nicely how one can use the function `ACEConjugatesForSubgroupNormalClosure`.

We also wish to acknowledge the contribution of Charles Sims. The inclusion of the `incomplete` (see 4.11.6) and `lenlex` (see 4.11.4) options, were made in response to requests to the GAP developers to include such options for coset table functions. Also, the definition of `lenlex` standardisation of coset tables (see Section 3.4), is due to him.

Finally, much of the work since 2005 in modifying the ACE package particularly for new versions of GAP and in getting ACE to its new home on `GitHub` has been due to Max Horn.

1.11 Changes from earlier versions

A reasonably comprehensive history of the evolution of pre-release versions of the ACE Package, is contained in the file `CHANGES` in the `gap` directory.

The `3.xxx` versions of the ACE Package were compatible with GAP 4.2, but were built for use with GAP 4.3. However, the current version of the ACE Package requires at least GAP 4.4. Users who still have GAP 4.3 will need to use ACE 4.1. One important change in GAP from GAP 4.2 to GAP 4.3 that has relevance for ACE Package users is the change of the default standard for the numbering of cosets in a coset table (see Section 3.4).

1 ► `ACEPackageVersion()` F

gives the current version of the ACE Package (i.e. the GAP code component; the function `ACEBinaryVersion` (see 6.5.25) gives details of the C code compilation).

2 Installing and Loading the ACE Package

2.1 Installing the ACE Package

To install, unpack the archive file, which should have a name of form `ace-`*XXX*`.tar.gz` for some package version number *XXX*, as a sub-directory in the `pkg` hierarchy of your version of GAP 4. This might be the `pkg` directory of the GAP 4 home directory; it is however also possible to keep an additional `pkg` directory in your private directories. The only essential difference with installing ACE in a `pkg` directory different to the GAP 4 home directory is that one must start GAP with the `-l` switch (see Section 3.1), e.g. if your private `pkg` directory is a subdirectory of `mygap` in your home directory you might type:

`gap -l ";`*myhomedir*`/mygap"`

where *myhomedir* is the path to your home directory, which may be replaced by a tilde. The empty path before the semicolon is filled in by the default path of the GAP 4 home directory.

After unpacking the archive, go to the newly created `ace` directory and call `./configure` *path* where *path* is the path to the GAP home directory. So for example if you install the package in the main `pkg` directory call

```
./configure ../..
```

This will fetch the architecture type for which GAP has been compiled last and create a `Makefile`. Now simply call

```
make
```

to compile the binary and to install it in the appropriate place.

Note that the current version of the configuration process only sets up directory paths. If you need a different compiler or different compiler options, you need to edit `src/Makefile.in` yourself, prior to calling `make`.

If you use this installation of GAP on different hardware platforms you will have to compile the binary for each platform separately. This is done by calling `configure`, editing `src/Makefile.in` possibly, and calling `make` for the package anew immediately after compiling GAP itself for the respective architecture. If your version of GAP is already compiled (and has last been compiled on the same architecture) you do not need to compile GAP again, it is sufficient to call the `configure` script in the GAP home directory.

The manual you are currently reading describes how to use the ACE Package; it can be found in the `doc` subdirectory of the package.

The subdirectory `standalone-doc` contains the file `ace3001.ps` which holds a version of the user manual for the ACE standalone; it forms part of [Ram99]). You should consult it if you are going to switch to the ACE standalone, e.g. in order to directly use interactive facilities.

The `src` subdirectory contains a copy of the original source of ACE. (The only modification is that a file `Makefile.in` was obtained from the different `make.xyz` and will be used to create a `Makefile`.) You can replace the source by a newer version before compiling.

If you encounter problems in installation please read the file `README.md`.

2.2 Loading the ACE Package

To use the ACE Package you have to request it explicitly. This is done by calling

```
gap> LoadPackage("ace");
---------------------------------------------------------------------------
Loading    ACE (Advanced Coset Enumerator) 5.4
GAP code by Greg Gamble <Greg.Gamble@uwa.edu.au> (address for correspondence)
       Alexander Hulpke (https://www.math.colostate.edu/~hulpke)
           [uses ACE binary (C code program) version: 3.001]
C code by  George Havas (http://staff.itee.uq.edu.au/havas)
           Colin Ramsay <cram@itee.uq.edu.au>
Co-maintainer: Max Horn <horn@mathematik.uni-kl.de>

                 For help, type: ?ACE
---------------------------------------------------------------------------
true
```

The banner may be suppressed by providing `false` as second argument to the `LoadPackage` command. The `LoadPackage` command is described in Section 76.2.1 in the GAP Reference Manual.

If GAP cannot find a working binary, the call to `LoadPackage` will fail.

If you want to load the ACE package by default, you can put the `LoadPackage` command into your `gaprc` (or `.gaprc` file) (see Section 3.2 in the GAP Reference Manual).

3 Some Basics

Throughout this manual for the use of ACE as a GAP package, we shall assume that the reader already knows the basic ideas of coset enumeration, as can be found for example in [Neu82]. There, a simple proof is given for the fact that a coset enumeration for a subgroup of finite index in a finitely presented group must eventually terminate with the correct result, provided the enumeration process obeys a simple condition (Mendelsohn's condition) formulated in Lemma 1 and Theorem 2 of [Neu82]. This basic condition leaves room for a great variety of **strategies** for coset enumeration; two "classical" ones have been known for a long time as the **Felsch strategy** and the **HLT strategy** (for Haselgrove, Leech and Trotter). Extensive experimental studies on many strategies can be found in [CDHW73], [Hav91], [HR99], and [HR01], in particular.

A few basic points should be particularly understood:

- "Subgroup (generator) and relator tables" that are used in the description of coset enumeration in [Neu82], and to which we will also occasionally refer in this manual, do **not** physically exist in the implementation of coset enumeration in ACE. For a terminology that is closer to the actual implementation and also to the formulations in the manual for the ACE standalone see [CDHW73] and [Hav91].
- Coset enumeration proceeds by defining **coset numbers** that really denote possible representatives for cosets written as words in the generators of the group. At the time of their generation it is not guaranteed that any two of these words do indeed represent different cosets. The state of an enumeration at any time is stored in a 2-dimensional array known as a **coset table** whose rows are indexed by coset numbers and whose columns are indexed by the group generators and their inverses. Entries of the coset table that are not yet defined are known as **holes** (typically they are filled with 0, i.e. an invalid coset number).
- It is customary in talking about coset enumeration to speak of **cosets** when really coset numbers are meant. While we try to avoid this in this interface manual, in certain word combinations such as **coset application** we will follow this custom.
- The definition of a coset number may lead to **deductions** from the "closing of rows in subgroup or relator tables". These are kept in a **deduction stack**.
- Also it may be found that (different) words in the generators defining different coset numbers really lie in the same coset of the given subgroup. This is called a **coincidence** and will eventually lead to the elimination of the larger of the two coset numbers. Until this elimination has been performed pending coincidences are kept in a **queue of coincidences**.
- A definition that will actually close a row in a subgroup or relator table will immediately yield twice as many entries in the coset table as other definitions. Such definitions are called **preferred definitions**, the places in rows of a subgroup or relator table that they close are also referred to as "gaps of length one" or minimal gaps. Such gaps can be found at little extra cost when "relators are traced from a given coset number". ACE keeps a selection of them in a **preferred definition stack** for use in some definition strategies (see [Hav91]).

It will also be necessary to understand some further basic features of the implementation and the corresponding terminology which we will explain in the sequel.

3.1 Enumeration Style

The first main decision for any coset enumeration is in which sequence to make definitions. When a new coset number has to be defined, in ACE there are basically three possible methods to choose from:

- One may fill the next empty entry in the coset table by scanning from the left/top of the coset table towards the right/bottom – that is, in order row by row. This is called **C style definition** (for **C**oset Table Based definition) of coset numbers. In fact a procedure needs to follow a method like this to some extent for the proofs that coset enumeration eventually terminates in the case of finite index (see [Neu82]).
- For an **R style definition** (for **R**elator Based definition), the order in which coset numbers are defined is explicitly prescribed by the order in which rows of (the subgroup generator tables and) the relator tables are filled by making definitions.
- One may choose definitions from a **Preferred Definition Stack**. In this stack possibilities for definition of coset numbers are stored that will close a certain row of a relator table. Using these **preferred definitions** is sometimes also referred to as a **minimal gaps strategy**. The idea of using these is that by closing a row in a relator table, thus, one will immediately get a consequence. We will come back to the obvious question of where one obtains this "preferred definition stack".

The **enumeration style** is mainly determined by the balance between C style and R style definitions, which is controlled by the values of the `ct` and `rt` options (see 4.13.2 and 4.13.3).

However this still leaves us with plenty of freedom for the design of definition strategies, freedom which can, for example, be used to great advantage in Felsch-type strategies. Though it is not strictly necessary, before embarking on further enumeration, Felsch-type programs generally start off by ensuring that each of the given subgroup generators produces a cycle of coset numbers at coset 1. To explain the idea, an example may help. Suppose a,b are the group generators and $w = Abab$ is a subgroup generator, where A represents the inverse of a; then to say that "$(1,i,j,k)$ is a cycle of coset numbers produced at coset 1 by w" means that the successive application of the "letters" A,b,a,b of w takes one successively from coset 1, through cosets i, j and k, and back to coset 1, i.e. A applied to coset 1 results in coset i, b applied to coset i results in coset j, a applied to coset j results in coset k, and finally b applied to coset k takes us back to coset 1. In this way, a hypothetical subgroup table is filled first. The use of this and other possibilities leads to the following table of **enumeration styles**.

```
Rt value     Ct value     style name
----------------------------------------

   0           >0         C
  <0           >0         Cr
  >0           >0         CR

  >0            0         R
  <0            0         R*
  >0           <0         Rc
  <0           <0         R/C
   0            0         R/C (defaulted)

----------------------------------------
```

C style

In this style, most definitions are made in the next empty coset table slot and are (in principle) tested in all essentially different positions in the relators; i.e. this is a Felsch-like style.

However, in C style, some definitions may be made following a preferred definition strategy, controlled by the `pmode` and `psize` options (see 4.14.2 and 4.14.3).

Cr style
: is like C style except that a single R style pass is done after the initial C style pass.

CR style
: In this style, alternate passes of C style and R style are performed.

R style
: In this style, all the definitions are made via relator scans; i.e. this is an HLT-like style.

R* style
: makes definitions the same as R style, but tests all definitions as for C style.

Rc style
: is like R style, except that a single C style pass is done after the initial R style pass.

R/C style
: In this style, we run in R style until an overflow, perform a lookahead on the entire table, and then switch to CR style.

Defaulted R/C (= **R/C (defaulted)**) **style**
: is the default style used if you call ACE without specifying options. In it, we use R/C style with `ct` set to 1000 and `rt` set to approximately 2000 divided by the total length of the relators in an attempt to balance R style and C style definitions when we switch to CR style.

3.2 Finding Deductions, Coincidences, and Preferred Definitions

First, let us broadly discuss strategies and how they influence "definitions". By **definition** we mean the allocation of a coset number. In a complete coset table each group relator produces a cycle of cosets numbers at each coset number, in particular, at coset 1; i.e. for each relator w, and for each coset number i, successive application of the letters of w trace through a sequence of coset numbers that begins and ends in i (see Section 3.1 for an example). It has been found to be a good general rule to use the given group relators as subgroup generators. This ensures the early definition of some useful coset numbers, and is the basis of the `default` strategy (see 5.1.1). The number of group relators included as subgroup generators is determined by the `no` option (see 4.13.4). Over a wide range of examples the use of group relators in this way has been shown to produce generally beneficial results in terms of the maximum number of cosets numbers defined at any one time and the total number of coset numbers defined. In [CDHW73], it was reported that for some Macdonald group $G(\alpha,\beta)$ examples, (pure) Felsch-type strategies (that don't include the given group relators as subgroup generators) e.g. the `felsch := 0` strategy (see 5.1.3) defined significantly more coset numbers than HLT-type (e.g. the `hlt` strategy, see 5.1.5) strategies. The comparison of these strategies in terms of total number of coset numbers defined, in [Hav91], for the enumeration of the cosets of a certain index 40 subgroup of the $G(3,21)$ Macdonald group were 91 for HLT versus 16067 for a pure Felsch-type strategy. For the Felsch strategy with the group relators included as subgroup generators, as for the `felsch := 1` strategy (see 5.1.3) the total number of coset numbers defined reduced markedly to 59.

A **deduction** occurs when the scanning of a relator results in the assignment of a coset table body entry. A completed table is only valid if every table entry has been tested in all essentially different positions in all relators. This testing can either be done directly (Felsch strategy) or via relator scanning (HLT strategy). If it is done directly, then more than one deduction can be waiting to be processed at any one time. The untested deductions are stored in a stack. How this stack is managed is determined by the `dmode` option (see 4.16.1), and its size is controlled by the `dsize` option (see 4.16.2).

As already mentioned a **coincidence** occurs when it is determined that two coset numbers in fact represent the same coset. When this occurs the larger coset number becomes a **dead coset number** and the coincidence is placed in a queue. When and how these dead coset numbers are eventually eliminated is controlled by the options `dmode`, `path` and `compaction` (see 4.16.1, 4.17.4 and 4.17.5). The user may also force coincidences to occur (see Section 3.3), which, however, may change the subgroup whose cosets are enumerated.

The key to performance of coset enumeration procedures is good selection of the next coset number to be defined. Leech in [Lee77] and [Lee84] showed how a number of coset enumerations could be simplified by removing coset numbers needlessly defined by computer implementations. Human enumerators intelligently choose which coset number should be defined next, based on the value of each potential definition. In particular, definitions which close relator cycles (or at least shorten gaps in cycles) are favoured. A definition which actually closes a relator cycle immediately yields twice as many table entries (deductions) as other definitions. The value of the `pmode` option (see 4.14.2) determines which definitions are **preferred**; if the value of the `pmode` option is non-zero, depending on the `pmode` value, gaps of length one found during relator C style (i.e. Felsch-like) scans are either filled immediately (subject to the value of `fill`) or noted in the **preferred definition stack**. The preferred definition stack is implemented as a ring of size determined by the `psize` option (see 4.14.3). However, making preferred definitions carelessly can violate the conditions required for guaranteed termination of the coset enumeration procedure in the case of finite index. To avoid such a violation ACE ensures a fraction of the coset table is filled before a preferred definition is made; the reciprocal of this fraction, the `fill factor`, is manipulated via the `fill` option (see 4.14.1). In [Hav91], the `felsch := 1` type enumeration of the cosets of the certain index 40 subgroup of the $G(3,21)$ Macdonald group was further improved to require a total number of coset numbers of just 43 by incorporating the use of preferred definitions.

3.3 Finding Subgroups

The ACE package, via its interactive ACE interface functions (described in Chapter 6), provides the possibility of searching for subgroups. To do this one starts at a known subgroup (possibly the trivial subgroup). Then one may augment it by adding new subgroup generators either explicitly via `ACEAddSubgroupGenerators` (see 6.7.4) or implicitly by introducing **coincidences** (see `ACECosetCoincidence`: 6.7.7, or `ACERandomCoincidences`: 6.7.8). Also, one may descend to smaller subgroups by deleting subgroup generators via `ACEDeleteSubgroupGenerators` (see 6.7.6).

3.4 Coset Table Standardisation Schemes

The default standardisation scheme for GAP and the standardisation scheme provided by ACE is the `lenlex` scheme, of Charles Sims [Sim94]. This scheme numbers cosets first according to word-length and then according to a lexical ordering of coset representatives. Each coset representative is a word in an alphabet consisting of the user-supplied generators and their inverses, and the lexical ordering of `lenlex` is that implied by ordering that alphabet so that each generator is followed by its inverse, and the generators appear in user-supplied order. See below for an example which gives the first 20 lines of the `lenlex` standard coset table of the (infinite) group with presentation $\langle x,y,a,b \mid x^2,y^3,a^4,b^2\rangle$.

In the table each inverse of a generator is represented by the corresponding uppercase letter (X represents the inverse of x etc.), and the lexical ordering of the representatives is that implied by defining an ordering of the alphabet of user-supplied generators and their inverses to be $x < X < y < Y < a < A < b < B$.

A `lenlex` standard coset table whose columns correspond, in order, to the already-described alphabet, of generators and their inverses, has an important property: a scan of the body of the table row by row from left to right, encounters new coset numbers in numeric order. Observe that the table below has this property: the definition of coset 1 is implicit; the first coset number we encounter in the table body is 2, then 2 again, 3, 4, 5, 6, 7, then 7 again, etc.

With the `lenlex` option (see 4.11.4), the coset table output by `ACECosetTable` or `ACECosetTableFromGensAndRels` is standardised according to the `lenlex` scheme.

coset no.	x	X	y	Y	a	A	b	B	rep've
1	2	2	3	4	5	6	7	7	
2	1	1	8	9	10	11	12	12	x
3	13	13	4	1	14	15	16	16	y
4	17	17	1	3	18	19	20	20	Y
5	21	21	22	23	24	1	25	25	a
6	26	26	27	28	1	24	29	29	A
7	30	30	31	32	33	34	1	1	b
8	35	35	9	2	36	37	38	38	xy
9	39	39	2	8	40	41	42	42	xY
10	43	43	44	45	46	2	47	47	xa
11	48	48	49	50	2	46	51	51	xA
12	52	52	53	54	55	56	2	2	xb
13	3	3	57	58	59	60	61	61	yx
14	62	62	63	64	65	3	66	66	ya
15	67	67	68	69	3	65	70	70	yA
16	71	71	72	73	74	75	3	3	yb
17	4	4	76	77	78	79	80	80	Yx
18	81	81	82	83	84	4	85	85	Ya
19	86	86	87	88	4	84	89	89	YA
20	90	90	91	92	93	94	4	4	Yb

Another standardisation scheme for coset tables numbers cosets according to coset representative word-length in the group generators and lexical ordering imposed by the user-supplied ordering of the group generators; it is known as `semilenlex` since though like `lenlex`, generator inverses do not feature. Here again is 20 lines of the coset table of the group with presentation $\langle x, y, a, b \mid x^2, y^3, a^4, b^2 \rangle$, this time `semilenlex` standardised.

coset no.	x	y	a	b	rep've
1	2	3	4	5	
2	1	6	7	8	x
3	9	10	11	12	y
4	13	14	15	16	a
5	17	18	19	1	b
6	20	21	22	23	xy
7	24	25	2	26	xa
8	27	28	29	2	xb
9	3	30	31	32	yx
10	33	1	34	35	yy
11	36	37	38	39	ya
12	40	41	42	3	yb
13	4	43	44	45	ax
14	46	47	48	49	ay
15	50	51	52	53	aa
16	54	55	56	4	ab
17	5	57	58	59	bx
18	60	61	62	63	by
19	64	65	66	67	ba
20	6	68	69	70	xyx

The term `semilenlex` was coined by Edmund Robertson and Joachim Neubüser, for the scheme's applicability to semigroups where generator inverses need not exist. This scheme ensures that as one scans the columns corresponding to the group generators (in user-supplied order) row by row, one encounters new coset numbers in numeric order.

Observe that the representatives are ordered according to length and then the lexical ordering implied by defining $x < y < a < b$ (with some words omitted due to their equivalence to words that precede them in the ordering). Also observe that as one scans the body of the table row by row from left to right new coset numbers appear in numeric order without gaps (2, 3, 4, 5, then 1 which we have implicitly already seen, 6, 7, etc.).

3.5 Coset Statistics Terminology

There are three statistics involved in the counting of coset number definitions: `activecosets`, `maxcosets` and `totcosets`; these are three of the fields of the record returned by `ACEStats` (see Section 1.3), and they correspond to the `a`, `m` and `t` statistics of an ACE "results message" (see Appendix A). As already described, coset enumeration proceeds by defining coset numbers; `totcosets` counts **all** such definitions made during an enumeration. During the coset enumeration process, **coincidences** usually occur, resulting in the larger of each coincident pair becoming a **dead coset number**. The statistic `activecosets` is the count of coset numbers left **alive** (i.e. not dead) at the end of an enumeration; and `maxcosets` is the maximum number of **alive** cosets at any point of an enumeration.

In practice, the statistics `maxcosets` and `totcosets` tend to be of a similar order, though, of course, `maxcosets` can never be more than `totcosets`.

3.6 Other Terminology

In various places in this manual, we will refer to a (main) **loop** or a **pass** through such a loop. We don't intend to give a precise meaning to these terms. The reader will need to forgive us for giving somewhat circular definitions in our attempt to make these terms less nebulous. It is sufficient to appreciate that the ACE enumerator is organised as a state machine, where each **state** is a value of the coset table held internally by ACE at the end of each "main loop". Each step from one state to the next (i.e. each passage through the main loop) performs an "action" (i.e., `lookahead`, `compaction`; see 4.15.2 and 4.17.5) or a block of actions (i.e., `|ct|` coset number definitions, `|rt|` coset number applications). ACE counts the number of passes through the main loop; if the option `loop` (see 4.17.3) is set to a positive integer, ACE makes an early return when the loop count hits the value of `loop`.

4 Options for ACE

ACE offers a wide range of options to direct and guide a coset enumeration, most of which are available from GAP through the interface provided by the ACE Package. We describe most of the options available via the interface in this chapter; other options, termed strategies, are defined in Chapter 5. (Strategies are merely special options of ACE that set a number of options described in this chapter, all at once.) Yet other options, for which interactive function alternatives are provided in Chapter 6, or which most GAP users are unlikely to need, are described in Appendix D. From within a GAP session, one may see the complete list of ACE options, after loading the ACE Package (see Section 2.2), by typing

```
gap> RecNames(KnownACEOptions);
[ "default", "help", "check", "generators", "start", "path", "cycles",
  "normal", "ds", "group", "subgroup", "relators", "order", "max", "rep",
  "system", "silent", "time", "begin", "text", "options", "fill",
  "aceinfile", "aceignore", "aceignoreunknown", "acenowarnings", "aceecho",
  "aceincomment", "aceexampleoptions", "lenlex", "semilenlex", "incomplete",
  "sg", "rl", "aep", "ai", "ao", "aceoutfile", "asis", "bye", "exit", "qui",
  "cc", "cfactor", "ct", "redo", "compaction", "continu", "dmode", "dsize",
  "dr", "dump", "easy", "echo", "enumeration", "felsch", "ffactor", "hard",
  "hlt", "hole", "lookahead", "loop", "mendelsohn", "messages", "monitor",
  "mode", "nc", "no", "oo", "pmode", "psize", "sr", "print", "purec",
  "purer", "rc", "recover", "contiguous", "rfactor", "rt", "row", "sc",
  "stabilising", "sims", "standard", "statistics", "stats", "style", "tw",
  "trace", "workspace" ]
```

(See Section 4.8.) Also, from within a GAP session, you may use GAP's help browser (see Chapter 2 in the GAP Reference Manual); to find out about any particular ACE option, simply type: "`?option` *option*", where *option* is one of the options listed above without any quotes, e.g.

```
gap> ?option echo
```

will display the section in this manual that describes the `echo` option.

We begin this chapter with several sections discussing the nature of the options provided. Please spend some time reading these sections. To continue onto the next section on-line using GAP's help browser, type:

```
gap> ?>
```

4.1 Passing ACE Options

Options are passed to the ACE interface functions in either of the two usual mechanisms provided by GAP, namely:

- options may be set globally using the function `PushOptions` (see Chapter 8 in the GAP Reference Manual); or
- options may be appended to the argument list of any function call, separated by a colon from the argument list (see 4.12 in the GAP Reference Manual), in which case they are then passed on recursively to any subsequent inner function call, which may in turn have options of their own.

In general, if ACE is to be used interactively one should avoid using the global method of passing options. In fact, it is recommended that prior to calling `ACEStart` the `OptionsStack` be empty.

4.2 Warnings regarding Options

As mentioned above, one can set options globally using the function `PushOptions` (see Chapter 8 in the GAP Reference Manual); however, options pushed onto `OptionsStack`, in this way, remain there until an explicit `PopOptions()` call is made. In contrast, options passed in the usual way behind a colon following a function's arguments (see 4.12 in the GAP Reference Manual) are local, and disappear from `OptionsStack` after the function has executed successfully; nevertheless, a function, that is passed options this way, will also see any global options or any options passed down recursively from functions calling that function, unless those options are over-ridden by options passed via the function. Also note that duplication of option names for different programs may lead to misinterpretations. Since a non-empty `OptionsStack` is potentially a mine-field for the unwary user, the function `ResetOptionsStack` (see 8.1.3 in the Reference Manual) is now in the GAP library and

1 ► `FlushOptionsStack()` F

introduced in version 3.001 of the ACE Package to perform the same function, is now a synonym for `ResetOptionsStack`; it simply executes `PopOptions()` until `OptionsStack` is empty.

However, `ResetOptionsStack` (or `FlushOptionsStack`) does not wipe out the options already passed to an **interactive** ACE process. We have provided `GetACEOptions` (see 6.5.7) to keep track of options that the ACE binary process still considers active, which may or may not be still on the `OptionsStack`. There is the interactive `SetACEOptions` (see 6.5.8) to change such options, or, of course, you can always elect to use `ACEQuit` (see 6.1.2) and then start a new interactive ACE process.

Finally, if `ACEIgnoreUnknownDefault := false` (see 4.11.2), there will be situations where an ACE interface function needs to be told explicitly to ignore options passed down recursively to it from calling functions. For this purpose we have provided the options `aceignore` (see 4.11.9) and `aceignoreunknown` (see 4.11.10).

4.3 Abbreviations and mixed case for ACE Options

Except for limitations imposed by GAP e.g. clashes with GAP keywords and blank spaces not allowed in keywords, the options of the ACE interface are the same as for the binary; so, for example, the options can appear in upper or lower case (or indeed, mixed case) and most may be abbreviated. Below we only list the options in all lower case, and in their longest form; where abbreviation is possible we give the shortest abbreviation in the option's description e.g. for the `mendelsohn` option we state that its shortest abbreviation is `mend`, which means `mende`, `mendel` etc., and indeed, `Mend` and `MeND`, are all valid abbreviations of that option. Some options have synonyms e.g. `cfactor` is an alternative for `ct`.

The complete list of ACE options known to the ACE interface functions, their abbreviations and the values that they are known to take may be gleaned from the `KnownACEOptions` record (see Section 4.8).

Options for each of the ACE interface functions `ACECosetTableFromGensAndRels`, `ACECosetTable`, `ACEStats` and `ACEStart` (see Chapter 6), comprise the few non-ACE-binary options (`silent`, `aceinfile`, `aceoutfile`, `aceignore`, `aceignoreunknown`, `acenowarnings`, `aceincomment`, `aceecho` and `echo`) discussed in Section 4.11, (almost) all single-word ACE binary options and `purer` and `purec`. The options `purer` and `purec` give the ACE binary options `pure r` and `pure c`, respectively; (they are the only multiple-word ACE binary options that do not have a single word alternative). The **only** single-word ACE binary options that are **not** available via the ACE interface are abbreviations that clash with GAP keywords (e.g. `fi` for `fill`, `rec` for `recover` and `continu` for `continue`). The detail of this paragraph is probably of little importance to the GAP user; these comments have been included for the user who wishes to reconcile the respective functionalities of the ACE interface and the ACE standalone, and are probably of most value to standalone users.

4.4 Honouring of the order in which ACE Options are passed

Note: Below we describe the intended behaviour, but unfortunately, since GAP 4.5 (approximately when ACE 5.1 was released) the order of options behind the colon is no longer honoured. Until this is fixed, if the order of ACE options needs to be respected, users should use ACE interactively (see 6.1.1).

It is important to realize that ACE's options (even the non-strategy options) are not orthogonal, i.e. the order in which they are put to ACE can be important. For this reason, except for a few options that have no effect on the course of an enumeration, the order in which options are passed to the ACE interface is preserved when those same options are passed to the ACE binary. One of the reasons for the non-orthogonality of options is to protect the user from obtaining invalid enumerations from bad combinations of options; another reason is that commonly one may specify a strategy option and override some of that strategy's defaults; the general rule is that the later option prevails. By the way, it's not illegal to select more than one strategy, but it's not sensible; as just mentioned, the later one prevails.

4.5 What happens if no ACE Strategy Option or if no ACE Option is passed

If an ACE interface function (`ACECosetTableFromGensAndRels`, `ACEStats`, `ACECosetTable` or `ACEStart`) is given no strategy option, the `default` strategy (see Chapter 5) is selected, and a number of options that ACE needs to have a value for are given default values, **prior** to the execution of any user options, if any. This ensures that ACE has a value for all its "run parameters"; three of these are defined from the ACE interface function arguments; and the remaining "run parameters", we denote by "ACE Parameter Options". For user convenience, we have provided the `ACEParameterOptions` record (see 4.12.1), the fields of which are the "ACE Parameter Options". The value of each field (option) of the `ACEParameterOptions` record is either a default value or (in the case of an option that is set by a strategy) a record of default values that ACE assumes when the user does not define a value for the option (either indirectly by selecting a strategy option or directly).

If the `default` strategy does not suffice, most usually a user will select one of the other strategies from among the ones listed in Chapter 5, and possibly modify some of the options by selecting from the options in this chapter. It's not illegal to select more than one strategy, but it's not sensible; as mentioned above, the later one prevails.

4.6 Interpretation of ACE Options

Options may be given a value by an assignment to the name (such as `time := ` *val*); or be passed without assigning a value, in which case GAP treats the option as **boolean** and sets the option to the value `true`, which is then interpreted by the ACE interface functions. Technically speaking the ACE binary itself does not have boolean options, though it does have some options which are declared by passing without a value (e.g. the `hard` strategy option) and others that are boolean in the C-sense (taking on just the values 0 or 1). The behaviour of the ACE interface functions (`ACECosetTableFromGensAndRels`, `ACEStats`, `ACECosetTable` or `ACEStart`) is essentially to restore as much as is possible a behaviour that mimics the ACE standalone; a `false` value is always translated to 0 and `true` may be translated to any of no-value, 0 or 1. Any option passed with an assigned value *val* other than `false` or `true` is passed with the value *val* to the ACE binary. Since this may appear confusing, let's consider some examples.

– The `hard` strategy option (see 5.1.4) should be passed without a value, which in turn is passed to the ACE binary without a value. However, the ACE interface function cannot distinguish the option `hard` being passed without a value, from it being passed via `hard := true`. Passing `hard := false` or `hard := ` *val* for any non-`true` *val* will however produce a warning message (unless the option `acenowarnings` is passed) that the value 0 (for `false`) or *val* is unknown for that option. Nevertheless, despite the warning, in this event, the ACE interface function passes the value to the ACE binary. When the ACE binary sees a line that it doesn't understand it prints a warning and simply ignores it. (So passing `hard := false` will produce warnings, but will have no ill effects.) The reason we still pass **unknown** values to the ACE binary is that it's conceivable a future version of the ACE binary might have several `hard` strategies, in which case the ACE interface function will still complain (until it's made aware of the new possible values) but it will perform in the correct manner if a value expected by the ACE binary is passed.

– The `felsch` strategy option (see 5.1.3) may be passed without a value (which chooses the **felsch 0** strategy) or with the values 0 or 1. Despite the fact that GAP sees this option as **boolean**; it is **not**. There are two Felsch strategies: **felsch 0** and **felsch 1**. To get the **felsch 1** strategy, the user must pass `felsch := 1`. If the user were to pass `felsch := false` the result would be the **felsch 0** strategy (since `false` is always translated to 0), i.e. the same as how `felsch := true` would be interpreted. We could protect the user more from such ideosyncrasies, but we have erred on the side of simplicity in order to make the interface less vulnerable to upgrades of the ACE binary.

The lesson from the two examples is: **check the documentation for an option to see how it will be interpreted**. In general, options documented (in this chapter) as **only** being no-value options can be safely thought of as boolean (i.e. you will get what you expect by assigning `true` or `false`), whereas strategy (no-value) options should **not** be thought of as boolean (a `false` assignment will **not** give you what you might have expected).

Options that are unknown to the ACE interface functions and not ignored (see below), that are passed without a value, are **always** passed to the ACE binary as no-value options (except when the options are ignored); the user can over-ride this behaviour simply by assigning the intended value. Note that it is perfectly safe to allow the ACE binary to be passed unknown options, since ACE simply ignores options it doesn't understand, issues an error message (which is just a warning and is output by GAP unless `acenowarnings` (see 4.11.11) is passed) and continues on with any other options passed in exactly the way it would have if the "unknown" options had not been passed.

An option is ignored if it is unknown to the ACE interface functions and one of the following is true:

– the global variable `ACEIgnoreUnknownDefault` is set to `false` (see 4.11.2) or,
– the `aceignoreunknown` option (see 4.11.10) is passed, or
– the `aceignore` option is passed and the option is an element of the list value of `aceignore` (see 4.11.9).

It is actually **recommended** that the user set `ACEIgnoreUnknownDefault` to `false`, since this will allow the user to see when ACE functions have been passed options that are "unknown" to the ACE package. In this way the user will be informed about misspelt options, for example. So it's a good debugging tool. Also, if the ACE binary is updated with a version with new options then these will not be known by the package (the GAP part) and it will be necessary to set `ACEIgnoreUnknownDefault` to `false` in order for the new options to be passed to the binary. When an ACE function is invoked indirectly by some function that was called with non-ACE options the warning messages may begin to be annoying, and it's then a simple matter to set `ACEIgnoreUnknownDefault` back to the ACE 3.003 default value of `true`.

Warning messages regarding unknown options are printed unless the `acenowarnings` (see 4.11.11) is passed or the option is ignored.

To see how options are interpreted by an ACE interface function, pass the `echo` option.

As mentioned above, any option that the ACE binary doesn't understand is simply ignored and a warning appears in the output from ACE. If this occurs, you may wish to check the input fed to ACE and the output from ACE, which when ACE is run non-interactively are stored in files whose full path names are recorded in the record fields `ACEData.infile` and `ACEData.outfile`, respectively. Alternatively, both interactively and non-interactively one can set the `InfoLevel` of `InfoACE` to 3 (see 1.9.3), to see the output from ACE, or to 4 to also see the commands directed to ACE.

4.7 An Example of passing Options

Continuing with the example of Section 1.2, one could set the `echo` option to be true, use the `hard` strategy option, increase the workspace to 10^7 words and turn messaging on (but to be fairly infrequent) by setting `messages` to a large positive value as follows:

```
gap> ACECosetTable(fgens, rels, [c]
>                   : echo, hard, Wo := 10^7, mess := 10000);;
```

As mentioned in the previous section, `echo` may be thought of as a boolean option, whereas `hard` is a strategy option (and hence should be thought of as a no-value option). Also, observe that two options have been abbreviated: `Wo` is a mixed case abbreviation of `workspace`, and `mess` is an abbreviation of `messages`.

4.8 The KnownACEOptions Record

1 ► `KnownACEOptions` V

is a GAP record whose fields are the ACE options known to the ACE interface; each field (known ACE option) is assigned to a list of the form `[`*i*`,` *ListOrFunction*`]`, where *i* is an integer representing the length of the shortest abbreviation of the option and *ListOrFunction* is either a list of (known) allowed values or a boolean function that may be used to determine if the given value is a (known) valid value e.g.

```
gap> KnownACEOptions.compaction;
[ 3, [ 0 .. 100 ] ]
```

indicates that the option `compaction` may be abbreviated to `com` and the (known) valid values are in the (integer) range 0 to 100; and

```
gap> KnownACEOptions.ct;
[ 2, <Category "IsInt"> ]
```

indicates that there is essentially no abbreviation of `ct` (since its shortest abbreviation is of length 2), and a value of `ct` is known to be valid if `IsInt` returns true for that value.

For user convenience, we provide the function

2 ► `ACEOptionData(` *optname* `)` F

which for a string *optname* representing an ACE option (or a guess of one) returns a record with the following fields:

`name`
: *optname* (unchanged);

`known`
: `true` if *optname* is a valid mixed case abbreviation of a known ACE option, and false otherwise;

`fullname`
: the lower case unabbreviated form of *optname* if the `known` field is set `true`, or *optname* in all lower case, otherwise;

`synonyms`
: a list of known ACE options synonymous with *optname*, in lowercase unabbreviated form, if the `known` field is set `true`, or a list containing just *optname* in all lower case, otherwise;

`abbrev`
: the shortest lowercase abbreviation of *optname* if the `known` field is set `true`, or *optname* in all lower case, otherwise.

For more on synonyms of ACE options, see 4.10.1.

The function `ACEOptionData` provides the user with all the query facility she should ever need; nevertheless, we provide the following functions.

3 ► `IsKnownACEOption(` *optname* `)` F

returns `true` if *optname* is a mixed case abbreviation of a field of `KnownACEOptions`, or `false` otherwise. `IsKnownACEOption(`*optname*`);` is equivalent to

`ACEOptionData(`*optname*`).known;`

4 ► `ACEPreferredOptionName(` *optname* `)` F

returns the lowercase unabbreviated first alternative of *optname* if it is a known ACE option, or *optname* in lowercase, otherwise. `ACEPreferredOptionName(`*optname*`);` is equivalent to

```
ACEOptionData(optname).synonyms[1];
```

5 ► `IsACEParameterOption( optname )` F

returns true if *optname* is an “ACE parameter option”. (ACE Parameter Options are described in Section 4.12.1). `IsACEParameterOption(optname);` is equivalent to

```
ACEPreferredOptionName(optname) in RecNames(ACEParameterOptions);
```

6 ► `IsACEStrategyOption( optname )` F

returns true if *optname* is an “ACE strategy option” (see Section 4.9). `IsACEStrategyOption(optname);` is equivalent to

```
ACEPreferredOptionName(optname) in ACEStrategyOptions;
```

4.9 The ACEStrategyOptions List

1 ► `ACEStrategyOptions` V

is a GAP list that contains the strategy options known to the ACE interface functions:

```
gap> ACEStrategyOptions;
[ "default", "easy", "felsch", "hard", "hlt", "purec", "purer", "sims" ]
```

See Chapter 5 for details regarding the ACE strategy options.

4.10 ACE Option Synonyms

1 ► `ACEOptionSynonyms` V

is a GAP record. A number of known ACE options have synonyms. The fields of the `ACEOptionSynonyms` record are the “preferred” option names and the values assigned to the fields are the lists of synonyms of those option names. What makes an option name “preferred” is somewhat arbitrary (in most cases, it is simply the shortest of a list of synonyms). For a “preferred” option name *optname* that has synonyms, the complete list of synonyms may be obtained by concatenating `[ optname ]` and `ACEOptionSynonyms.(optname)`, e.g.

```
gap> Concatenation( [ "messages" ], ACEOptionSynonyms.("messages") );
[ "messages", "monitor" ]
```

More generally, for an arbitrary option name *optname* its list of synonyms (which may be a list of one element) may be obtained as the `synonyms` field of the record returned by `ACEOptionData(optname)` (see 4.8.2).

4.11 Non-ACE-binary Options

1 ► `NonACEbinOptions` V

is a GAP list of options that have meaning only for the ACE Package interface, i.e. options in `KnownACEOptions` that are **not** ACE binary options; each such option is described in detail below. **Except** for the options listed in `NonACEbinOptions` and those options that are excluded via the `aceignore` and `aceignoreunknown` options (described below), **all** options that are on the `OptionsStack` when an ACE interface function is called, are passed to the ACE binary. Even options that produce the warning message: “`unknown (maybe new) or bad`”, by virtue of not being a field of `KnownACEOptions`, are passed to the ACE binary (except that the options `purer` and `purec` are first translated to `pure r` and `pure c`, respectively). When the ACE binary encounters an option that it doesn’t understand it issues a warning and simply ignores it; so options accidentally passed to ACE are unlikely to pose problems.

We also mention here, since it is related to an option of this section, the following.

2 ▶ `ACEIgnoreUnknownDefault` V

is a global variable (**not** an option) that is initially set by the ACE package to `true`, and is the default action that ACE takes for options that are unknown to the ACE package (but may be new options provided in a new version of the ACE binary). Despite the fact that it is normally set `true`, it is recommended (especially for the novice user of the ACE package) to set `ACEIgnoreUnknownDefault := false`; the worst that can happen is being annoyed by a profusion of warnings of unknown options. For individual functions, the user may use the option `aceignoreunknown` (see 4.11.10) to over-ride the setting of `ACEIgnoreUnknownDefault`.

Here now, are the few options that are available to the GAP interface to ACE that have no counterpart in the ACE standalone:

3 ▶ `silent`

> Inhibits an `Error` return when generating a coset table.

If a coset enumeration that invokes `ACECosetTableFromGensAndRels` does not finish within the preset limits, an error is raised by the interface to GAP, unless the option `silent` or `incomplete` (see 4.11.6) has been set; in the former case, `fail` is returned. This option is included to make the behaviour of `ACECosetTableFromGensAndRels` compatible with that of the function `CosetTableFromGensAndRels` it replaces. If the option `incomplete` is also set, it overrides option `silent`.

4 ▶ `lenlex`

> Ensures that `ACECosetTable` and `ACECosetTableFromGensAndRels` output a coset table that is `lenlex` standardised.

The `lenlex` scheme, numbers cosets in such a way that their "preferred" (coset) representatives, in an alphabet consisting of the user-submitted generators and their inverses, are ordered first according to `length` and then according to a `lexical` ordering. In order to describe what the `lenlex` scheme's `lexical` ordering is, let us consider an example. Suppose the generators submitted by the user are, in user-supplied order, `[x, y, a, b]`, and represent the inverses of these generators by the corresponding uppercase letters: `[X, Y, A, B]`, then the `lexical` ordering of `lenlex` is that derived from defining `x < X < y < Y < a < A < b < B`.

Notes: In some circumstances, ACE prefers to swap the first two generators; such cases are detected by the function `IsACEGeneratorsInPreferredOrder` (see 1.2.3). In such cases, special action is taken to avoid ACE swapping the first two generators; this action is described in the notes for `ACEStandardCosetNumbering` (see 6.7.2). When this special action is invoked, a side-effect is that any setting of the `asis` (see 4.13.1) option by the user is ignored.

The `lenlex` standardisation scheme is the default coset table standardisation scheme of GAP. Alternatively, `semilenlex` can be used. Both schemes are described in detail in Section 3.4.

5 ▶ `semilenlex`

> Ensures that `ACECosetTable` and `ACECosetTableFromGensAndRels` output a coset table that is `semilenlex` standardised.

The `semilenlex` scheme, numbers cosets in such a way that their "preferred" (coset) representatives, in an alphabet consisting of only the user-submitted generators, are ordered first according to `length` and then according to a `lexical` ordering.

6 ▶ `incomplete`

> Allows the return of an `incomplete` coset table, when a coset enumeration does not finish within preset limits.

If a coset enumeration that invokes `ACECosetTableFromGensAndRels` or `ACECosetTable` does not finish within the preset limits, an error is raised by the interface to GAP, unless the option `silent` (see 4.11.3) or `incomplete` has been set; in the latter case, a partial coset table, that is a valid GAP list of lists, is returned. Each position of the table without a valid coset number entry is filled with a zero. If the option `silent` is also set, `incomplete` prevails. An incomplete table is returned reduced (i.e. with insignificant coset numbers — those appearing only in their place of definition — removed) and `lenlex` standardised (regardless of whether the `semilenlex` option is in force). When an incomplete table is returned, a warning is emitted at `InfoACE` or `InfoWarning` level 1.

7 ► `aceinfile:=`*filename*

Creates an ACE input file *filename* for use with the standalone only; *filename* should be a string. (Shortest abbreviation: `acein`.)

This option is only relevant to `ACECosetTableFromGensAndRels` and is ignored if included as an option for invocations of `ACEStats` and `ACEStart`. If this option is used, GAP creates an input file with filename *filename* only, and then exits (i.e. the ACE binary is not called). This option is provided for users who wish to work directly with the ACE standalone. The full path to the input file normally used by ACE (i.e. when option `aceinfile` is not used) is stored in `ACEData.infile`.

8 ► `aceoutfile:=`*filename*

Redirects ACE output to file *filename*; *filename* should be a string. (Shortest abbreviation: `aceo`.)

This is actually a synonym for the `ao` option. Please refer to 4.20.1, for further discussion of this option.

9 ► `aceignore:=`*optionList*

Directs an ACE function to ignore the options in *optionList*; *optionList* should be a list of strings. (Shortest abbreviation: `aceig`.)

If a function called with its own options, in turn calls an ACE function for which those options are not intended, the ACE function will pass those options to the ACE binary. If those options are unknown to the ACE interface (and `ACEIgnoreUnknownDefault := false` and `aceignoreunknown` is not passed; see 4.11.2 and 4.11.10) a warning is issued. Options that are unknown to the ACE binary are simply ignored by ACE (and a warning that the option was ignored appears in the ACE output, which the user will not see unless the `InfoLevel` of `InfoACE` or `InfoWarning` is set to 1). This option enables the user to avoid such options being passed at all, thus avoiding the warning messages and also any options that coincidentally are ACE options but are not intended for the ACE function being called.

10 ► `aceignoreunknown`

Directs an ACE function to ignore any options not known to the ACE interface. (Shortest abbreviation: `aceignoreu`.)

This option is provided for similar reasons to `aceignore`. Normally, it is safe to include it, to avoid aberrant warning messages from the ACE interface. However, fairly obviously, it should not be passed without a value (or set to `true`) in the situation where a new ACE binary has been installed with new options that are not listed among the fields of `KnownACEOptions`, which you intend to use. Omitting the `aceignoreunknown` option is equivalent to setting it to the value of `ACEIgnoreUnknownDefault` (see 4.11.2); i.e. it is superfluous if `ACEIgnoreUnknownDefault := true` unless `aceignoreunknown` is set to `false`.

11 ► `acenowarnings`

Inhibits the warning message "`unknown (maybe new) or bad option`" for options not listed in `KnownACEOptions`. (Shortest abbreviation: `acenow`.)

This option suppresses the warning messages for unknown options (to the ACE interface), but unlike `aceignore` and `aceignoreunknown` still allows them to be passed to the ACE binary.

12 ► `echo`
► `echo:=2`

Echoes arguments and options (and indicates how options were handled).

Unlike the previous options of this section, there **is** an ACE binary option `echo`. However, the `echo` option is handled by the ACE interface and is not passed to the ACE binary. (If you wish to put `echo` in a standalone script use the `aceecho` option following.) If `echo` is passed with the value 2 then a list of the options (together with their values) that are set via ACE defaults are also echoed to the screen.

13 ► `aceecho`

The ACE binary's `echo` command.

This option is only included so that a user **can** put an `echo` statement in an ACE standalone script. Otherwise, use `echo` (above).

14 ► `aceincomment:=`*string*

Print comment *string* in the ACE input; *string* must be a string. (Shortest abbreviation: `aceinc`.)

This option prints the comment *string* behind a sharp sign (#) in the input to ACE. Only useful for adding comments (that ACE ignores) to standalone input files.

15 ► `aceexampleoptions`

An **internal** option for `ACEExample`.

This option is passed **internally** by `ACEExample` to the ACE interface function it calls, when one invokes `ACEExample` with options. Its purpose is to provide a mechanism for the over-riding of an example's options by the user. The option name is deliberately long and has no abbreviation to discourage user use.

4.12 ACE Parameter Options

1 ► `ACEParameterOptions` V

is a GAP record, whose fields are the "ACE Parameter Options". The "ACE Parameter Options" are options which, if not supplied a value by the user, are supplied a default value by ACE. In fact, the "ACE Parameter Options" are those options that appear (along with `Group Generators`, `Group Relators` and `Subgroup Generators`, which are defined from ACE interface function arguments) in the "Run Parameters" block of ACE output, when, for example, the `messages` option is non-zero.

For each field (ACE parameter option) of the `ACEParameterOptions` record, the value assigned is the default value (or a record of default values) that are supplied by ACE when the option is not given a value by the user (either indirectly by selecting a strategy option or directly).

In the cases where the value of a field of the `ACEParameterOptions` record is itself a record, the fields of that record are `default` and strategies for which the value assigned by that strategy differs from the `default` strategy. A "strategy", here, is the strategy option itself, if it is only a no-value option, or the strategy option concatenated with any of its integer values (as strings), otherwise (e.g. `felsch0` and `sims9` are strategies, and `hlt` is both a strategy and a strategy option). As an exercise, the reader might like to try to reproduce the table at the beginning of Chapter 5 using the `ACEParameterOptions` record. (Hint: you first need to select those fields of the `ACEParameterOptions` record whose values are records with at least two fields.)

Note: Where an "ACE Parameter Option" has synonyms, only the "preferred" option name (see 4.10.1) appears as a field of `ACEParameterOptions`. The complete list of "ACE Parameter Options" may be obtained by

```
gap> Concatenation( List(RecNames(ACEParameterOptions),
>                         optname -> ACEOptionData(optname).synonyms) );
[ "path", "subgroup", "max", "time", "fill", "ffactor", "asis", "ct",
  "cfactor", "compaction", "dmode", "dsize", "enumeration", "hole",
  "lookahead", "loop", "mendelsohn", "messages", "monitor", "no", "pmode",
  "psize", "rt", "rfactor", "row", "workspace" ]
```

We describe the "ACE Parameter Options" in the Sections 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, and 4.19, following.

4.13 General ACE Parameter Options that Modify the Enumeration Process

1 ► `asis`

Do not reduce relators. (Shortest abbreviation: `as`.)

By default, ACE freely and cyclically reduces the relators, freely reduces the subgroup generators, and sorts relators and subgroup generators in length-increasing order. If you do not want this, you can switch it off by setting the `asis` option.

Notes: As well as allowing you to use the presentation **as** it **is** given, this is useful for forcing definitions to be made in a prespecified order, by introducing dummy (i.e., freely trivial) subgroup generators. (Note that the exact form of the presentation can have a significant impact on the enumeration statistics.) For some fine points of the influence of `asis` being set on the treatment of involutory generators see the ACE standalone manual.

2 ► `ct:=`*val*
► `cfactor:=`*val*

Number of C style definitions per pass; *val* should be an integer. (Shortest abbreviation of `cfactor` is `c`.)

The absolute value of *val* sets the number of C style definitions per pass through the enumerator's main loop. The sign of *val* sets the style. The possible combinations of the values of `ct` and `rt` (described below) are given in the table of enumeration styles in Section 3.1.

3 ► `rt:=`*val*
► `rfactor:=`*val*

Number of R style definitions per pass; *val* should be an integer. (Shortest abbreviation of `rfactor` is `r`.)

The absolute value of *val* sets the number of R style definitions per pass through the enumerator's main loop. The sign of *val* sets the style. The possible combinations of the values of `ct` (described above) and `rt` are given in the table of enumeration styles in Section 3.1.

4 ► `no:=`*val*

The number of group relators to include in the subgroup; *val* should be an integer greater than or equal to -1.

It is sometimes helpful to include the group relators into the list of the subgroup generators, in the sense that they are applied to coset number 1 at the start of an enumeration. A value of 0 for this option turns this feature off and the (default) argument of -1 includes all the relators. A positive argument includes the specified number of relators, in order. The `no` option affects only the C style procedures.

5 ► `mendelsohn`

Turns on mendelsohn processing. (Shortest abbreviation: `mend`.)

Mendelsohn style processing during relator scanning/closing is turned on by giving this option. Off is the default, and here relators are scanned only from the start (and end) of a relator. Mendelsohn "on" means that all (different) cyclic permutations of a relator are scanned.

The effect of Mendelsohn style processing is case-specific. It can mean the difference between success or failure, or it can impact the number of coset numbers required, or it can have no effect on an enumeration's statistics.

Note: Processing all cyclic permutations of the relators can be very time-consuming, especially if the presentation is large. So, all other things being equal, the Mendelsohn flag should normally be left off.

4.14 ACE Parameter Options Modifying C Style Definitions

The next three options are relevant only for making **C style definitions** (see Section 3.1). Making definitions in C style, that is filling the coset table line by line, it can be very advantageous to switch to making definitions from the preferred definition stack. Possible definitions can be extracted from this stack in various ways and the two options `pmode` and `psize` (see 4.14.2 and 4.14.3 respectively) regulate this. However it should be clearly understood that making all definitions from a preferred definition stack one may violate the condition of Mendelsohn's theorem, and the option `fill` (see 4.14.1) can be used to avoid this.

1 ► `fill:=`*val*
► `ffactor:=`*val*

> Controls the preferred definition strategy by setting the fill factor; *val* must be a non-negative integer. (Shortest abbreviation of `fill` is `fil`, and shortest abbreviation of `ffactor` is `f`.)

Unless prevented by the fill factor, gaps of length one found during deduction testing are preferentially filled (see [Hav91]). However, this potentially violates the formal requirement that all rows in the coset table are eventually filled (and tested against the relators). The fill factor is used to ensure that some constant proportion of the coset table is always kept filled. Before defining a coset number to fill a gap of length one, the enumerator checks whether `fill` times the completed part of the table is at least the total size of the table and, if not, fills coset table rows in standard order (i.e. C style; see Section 3.1) instead of filling gaps.

An argument of 0 selects the default value of $\lfloor 5(n+2)/4 \rfloor$, where n is the number of columns in the table. This default fill factor allows a moderate amount of gap-filling. If `fill` is 1, then there is no gap-filling. A large value of `fill` can cause what is in effect infinite looping (resolved by the coset enumeration failing). However, in general, a large value does work well. The effects of the various gap-filling strategies vary widely. It is not clear which values are good general defaults or, indeed, whether any strategy is always "not too bad".

This option is identified as `Fi` in the "Run Parameters" block (obtained when `messages` is non-zero) of the ACE output, since for the ACE binary, `fi` is an allowed abbreviation of `fill`. However, `fi` is a GAP keyword and so the shortest abbreviation of `fill` allowed by the interface functions is `fil`.

2 ► `pmode:=`*val*

> Option for preferred definitions; *val* should be in the integer range 0 to 3. (Shortest abbreviation: `pmod`.)

The value of the `pmode` option determines which definitions are preferred. If the argument is 0, then Felsch style definitions are made using the next empty table slot. If the argument is non-zero, then gaps of length one found during relator scans in Felsch style are preferentially filled (subject to the value of `fill`). If the argument is 1, they are filled immediately, and if it is 2, the consequent deduction is also made immediately (of course, these are also put on the deduction stack). If the argument is 3, then the gaps of length one are noted in the preferred definition queue.

Provided such a gap survives (and no coincidence occurs, which causes the queue to be discarded) the next coset number will be defined to fill the oldest gap of length one. The default value is either 0 or 3, depending on the strategy selected (see Chapter 5). If you want to know more details, read the code.

3 ► `psize:=`*val*

> Size of preferred definition queue; *val* **must** be 0 or 2^n, for some integer $n > 0$. (Shortest abbreviation: `psiz`.)

The preferred definition queue is implemented as a ring, dropping earliest entries. An argument of 0 selects the default size of 256. Each queue slot takes two words (i.e., 8 bytes), and the queue can store up to $2^n - 1$ entries.

4.15 ACE Parameter Options for R Style Definitions

1 ▶ `row:=`*val*

Set the "row filling" option; *val* is either 0 or 1.

By default, "row filling" is on (i.e. `true` or 1). To turn it off set `row` to `false` or 0 (both are translated to 0 when passed to the ACE binary). When making HLT style (i.e. R style; see Section 3.1) definitions, rows of the coset table are scanned for holes after its coset number has been applied to all relators, and definitions are made to fill any holes encountered. This will, in particular, guarantee fulfilment of the condition of Mendelsohn's Theorem. Failure to do so can cause even simple enumerations to overflow.

2 ▶ `lookahead:=`*val*

Lookahead; *val* should be in the integer range 0 to 4. (Shortest abbreviation: `look`.)

Although HLT style strategies are fast, they are local, in the sense that the implications of any definitions/deductions made while applying coset numbers may not become apparent until much later. One way to alleviate this problem is to perform lookaheads occasionally; that is, to test the information in the table, looking for deductions or concidences. ACE can perform a lookahead when the table overflows, before the compaction routine is called. Lookahead can be done using the entire table or only that part of the table above the current coset number, and it can be done R style (scanning coset numbers from the beginning of relators) or C style (testing all definitions in all essentially different positions).

The following are the effects of the possible values of `lookahead`:

- 0 disables lookahead;
- 1 does a partial table lookahead, R style;
- 2 does a whole table lookahead, C style;
- 3 does a whole table lookahead, R style; and
- 4 does a partial table lookahead, C style.

The default is 1 if the `hlt` strategy is used and 0 otherwise; see Chapter 5.

Section 3.1 describes the various enumeration styles, and, in particular, R style and C style.

4.16 ACE Parameter Options for Deduction Handling

1 ▶ `dmode:=`*val*

Deduction mode; *val* should be in the integer range 0 to 4. (Shortest abbreviation: `dmod`.)

A completed table is only valid if every table entry has been tested in all essentially different positions in all relators. This testing can either be done directly (`felsch` strategy; see 5.1.3) or via relator scanning (`hlt` strategy; see 5.1.5). If it is done directly, then more than one deduction (i.e., table entry) can be waiting to be processed at any one time. So the untested deductions are stored in a stack. Normally this stack is fairly small but, during a collapse, it can become very large.

This command allows the user to specify how deductions should be handled. The value *val* has the following interpretations:

- 0: discard deductions if there is no stack space left;
- 1: as for 0, but purge any redundant coset numbers on the top of the stack at every coincidence;
- 2: as for 0, but purge all redundant coset numbers from the stack at every coincidence;
- 3: discard the entire stack if it overflows; and
- 4: if the stack overflows, double the stack size and purge all redundant coset numbers from the stack.

The default deduction mode is either 0 or 4, depending on the strategy selected (see Chapter 5), and it is recommended that you stay with the default. If you want to know more details, read the well-commented C code.

Notes: If deductions are discarded for any reason, then a final relator check phase will be run automatically at the end of the enumeration, if necessary, to check the result.

2 ► `dsize:=`*val*

Deduction stack size; *val* should be a non-negative integer. (Shortest abbreviation: `dsiz`.)

Sets the size of the (initial) allocation for the deduction stack. The size is in terms of the number of deductions, with one deduction taking two words (i.e., 8 bytes). The default size, of 1000, can be selected by a value of 0. See the `dmode` entry for a (brief) discussion of deduction handling.

4.17 Technical ACE Parameter Options

The following options do not affect how the coset enumeration is done, but how it uses the computer's resources. They might thus affect the runtime as well as the range of problems that can be tackled on a given machine.

1 ► `workspace:=`*val*

Workspace size in words (default 10^8); *val* should be an expression that evaluates to a positive integer, or a string of digits ending in a `k`, `M` or `G` representing a multiplication factor of 10^3, 10^6 or 10^9, respectively e.g. both `workspace := 2 * 10^6` and `workspace := "2M"` specify a workspace of 2×10^6 words. Actually, if the value of `workspace` is entered as a string, each of `k`, `M` or `G` will be accepted in either upper or lower case. (Shortest abbreviation: `wo`.)

By default, ACE has a physical table size of 10^8 words (i.e., 4×10^8 bytes in the default 32-bit environment). The number of coset numbers in the table is the table size divided by the number of columns. Although the number of coset numbers is limited to $2^{31} - 1$ (if the C `int` type is 32 bits), the table size can exceed the 4GByte 32-bit limit if a suitable machine is used.

2 ► `time:=`*val*

Maximum execution time in seconds; *val* must be an integer greater than or equal to -1. (Shortest abbreviation: `ti`.)

The `time` command puts a time limit (in seconds) on the length of a run. The default is -1 which is no time limit. If the argument is ≥ 0 then the total elapsed time for this call is checked at the end of each pass through the enumerator's main loop, and if it's more than the limit the run is stopped and the current table returned.

Note that a limit of 0 performs exactly one pass through the main loop, since $0 \geq 0$.

The time limit is approximate, in the sense that the enumerator may run for a longer, but never a shorter, time. So, if there is, e.g., a big collapse (so that the time round the loop becomes very long), then the run may run over the limit by a large amount.

Notes: The time limit is CPU-time, not wall-time. As in all timing under UNIX, the clock's granularity (usually 10 milliseconds) and the system load can affect the timing; so the number of main loop iterations in a given time may vary.

3 ► `loop:=`*val*

Loop limit; *val* should be a non-negative integer.

The core enumerator is organised as a state machine, with each step performing an "action" (i.e., lookahead, compaction) or a block of actions (i.e., $|$`ct`$|$ coset number definitions, $|$`rt`$|$ coset number applications). The number of passes through the main loop (i.e., steps) is counted, and the enumerator can make an early return when this count hits the value of `loop`. A value of 0, the default, turns this feature off.

Guru Notes: You can do lots of really neat things using this feature, but you need some understanding of the internals of ACE to get real benefit from it.

4 ▶ `path`

Turns on path compression.

To correctly process multiple concidences, a union-find must be performed. If both path compression and weighted union are used, then this can be done in essentially linear time (see, e.g., [CLR90]). Weighted union alone, in the worst-case, is worse than linear, but is subquadratic. In practice, path compression is expensive, since it involves many coset table accesses. So, by default, path compression is turned off; it can be turned on by `path`. It has no effect on the result, but may affect the running time and the internal statistics.

Guru Notes: The whole question of the best way to handle large coincidence forests is problematic. Formally, ACE does not do a weighted union, since it is constrained to replace the higher-numbered of a coincident pair. In practice, this seems to amount to much the same thing! Turning path compression on cuts down the amount of data movement during coincidence processing at the expense of having to trace the paths and compress them. In general, it does not seem to be worthwhile.

5 ▶ `compaction:=`*val*

Percentage of dead coset numbers to trigger compaction; *val* should be an integer (percentage) in the integer range 0 to 100. (Shortest abbreviation: `com`.)

The option `compaction` sets the percentage of **dead** coset numbers needed to trigger compaction of the coset table, during an enumeration. A **dead** coset (number) is a coset number found to be coincident with a smaller coset number. The default is 10 or 100, depending on the strategy used (see Chapter 5).

Compaction recovers the space allocated to coset numbers which are flagged as dead. It results in a table where all the active coset numbers are numbered contiguously from 1, and with the remainder of the table available for new coset numbers.

The coset table is compacted when a definition of a coset number is required, there is no space for a new coset number available, and provided that the given percentage of the coset table contains dead coset numbers. For example, if `compaction` = 20 then compaction will occur only if 20% or more of the coset numbers in the table are dead. An argument of 100 means that compaction is never performed, while an argument of 0 means always compact, no matter how few dead coset numbers there are (provided there is at least one, of course).

Compaction may be performed multiple times during an enumeration, and the table that results from an enumeration may or may not be compact, depending on whether or not there have been any coincidences since the last compaction (or from the start of the enumeration, if there have been no compactions).

Notes: In some strategies (e.g., `hlt`; see 5.1.5) a lookahead phase may be run before compaction is attempted. In other strategies (e.g., `sims := 3`; see 5.1.8) compaction may be performed while there are outstanding deductions; since deductions are discarded during compaction, a final lookahead phase will (automatically) be performed.

Compacting a table "destroys" information and history, in the sense that the coincidence list is deleted, and the table entries for any dead coset numbers are deleted.

6 ▶ `max:=`*val*

Sets the maximum coset number that can be defined; *val* should be 0 or an integer greater than or equal to 2.

By default (which is the case `max` = 0), all of the workspace is used, if necessary, in building the coset table. So the table size (in terms of the number of rows) is an upper bound on how many coset numbers can be alive at any one time. The `max` option allows a limit to be placed on how much physical table space is made available to the enumerator. Enough space for at least two coset numbers (i.e., the subgroup and one other) must be made available.

Notes: If the `easy` strategy (see 5.1.2) is selected, so that `compaction` (see 4.17.5) is off (i.e. set to 100) and `lookahead` (see 4.15.2) is off (i.e. set to 0), and `max` is set to a positive integer, then coset numbers are not reused, and hence `max` bounds the **total** number `totcosets` (see Section 3.5) of coset numbers defined during an enumeration.

On the other hand, if one (or both) of `compaction` or `lookahead` is not off, then some reuse of coset numbers may occur, so that, for the case where `max` is a positive integer, the value of `totcosets` may be greater than `max`.

However, whenever `max` is set to a positive integer, both **activecosets** (the number of **alive** coset numbers at the end of an enumeration) and **maxcosets** (the maximum number of alive coset numbers at any point of an enumeration) are bounded by `max`. See Section 3.5, for a discussion of the terminology: **activecosets** and **maxcosets**.

7 ► `hole:=`*val*

Maximum percentage of holes allowed during an enumeration; *val* should be an integer in the range -1 to 100. (Shortest abbreviation: `ho`.)

This is an experimental feature which allows an enumeration to be terminated when the percentage of holes in the table exceeds a given value. In practice, calculating this is very expensive, and it tends to remain constant or decrease throughout an enumeration. So the feature doesn't seem very useful. The default value of -1 turns this feature off. If you want more details, read the source code.

4.18 ACE Parameter Options controlling ACE Output

1 ► `messages:=`*val*
► `monitor:=`*val*

Sets the verbosity of output from ACE; *val* should be an integer. (Shortest abbreviation of `messages` is `mess`, and shortest abbreviation of `monitor` is `mon`.)

By default, *val* = 0, for which ACE prints out only a single line of information, giving the result of each enumeration. If *val* is non-zero then the presentation and the parameters are echoed at the start of the run, and messages on the enumeration's status as it progresses are also printed out. The absolute value of *val* sets the frequency of the progress messages, with a negative sign turning hole monitoring on. Note that, hole monitoring is expensive, so don't turn it on unless you really need it.

Note that, ordinarily, one will not see these messages: non-interactively, these messages are directed to file `ACE-Data.outfile` (or *filename*, if option `aceoutfile := `*filename*, or `ao := `*filename*, is used), and interactively these messages are simply not displayed. However, one can change this situation both interactively and non-interactively by setting the `InfoLevel` of `InfoACE` to 3 via

```
gap> SetInfoACELevel(3);
```

Then ACE's messages are displayed prepended with "`#I `". Please refer to Appendix A, where the meanings of ACE's output messages are fully discussed.

4.19 ACE Parameter Options that give Names to the Group and Subgroup

These options may be safely ignored; they only give names to the group or subgroup within the ACE output, and have no effect on the enumeration itself.

1 ► `enumeration:=`*string*

Sets the `Group Name` to *string*; *string*, must of course be a string. (Shortest abbreviation: `enum`.)

The ACE binary has a two-word synonym for this option: `Group Name` and this is how it is identified in the "Run Parameters" block of the ACE output when `messages` has a non-zero value. The default `Group Name` is `"G"`.

2 ► `subgroup:=`*string*

Sets the `Subgroup Name` to *string*; *string* must of course be a string. (Shortest abbreviation: `subg`.)

The default `Subgroup Name` is `"H"`.

4.20 Options for redirection of ACE Output

1 ► `ao:=`*filename*
► `aceoutfile:=`*filename*

Redirects (alters) output to *filename*; *filename* should be a string.

Non-interactively, output from ACE is normally directed to a temporary file whose full path is stored in `ACEData.outfile`, which is parsed to produce a coset table or a list of statistics. This option causes ACE's output to be directed to *filename* instead, presumably because the user wishes to see (and keep) data output by the ACE binary, other than the coset table output from `ACECosetTableFromGensAndRels` or the statistics output by `ACEStats`. Please refer to Appendix A, where we discuss the meaning of the additional data to be found in the ACE binary's output. The option `aceoutfile` is a GAP-introduced synonym for `ao`, that is translated to `ao` before submission to the ACE binary. Do not use option `aceoutfile` when running the standalone directly. Happily, `ao` can also be regarded as mnemonical for `aceoutfile`.

4.21 Other Options

ACE has a number of other options, but the GAP user will not ordinarily need them, since, in most cases, alternative interactive functions exist. These remaining options have been relegated to Appendix D. The options listed there may be used both interactively and non-interactively, but many are probably best used directly via the ACE standalone.

5 Strategy Options for ACE

It can be difficult to select appropriate options when presented with a new enumeration. The problem is compounded by the fact that no generally applicable rules exist to predict, given a presentation, which option settings are "good". To help overcome this problem, ACE contains various commands which select particular enumeration strategies. One or other of these strategies may work and, if not, the results may indicate how the options can be varied to obtain a successful enumeration.

If no strategy option is passed to ACE, the `default` strategy is assumed, which starts out presuming that the enumeration will be easy, and if it turns out not to be so, ACE switches to a strategy designed for more difficult enumerations. The other straightforward options for beginning users are `easy` and `hard`. Thus, `easy` will quickly succeed or fail (in the context of the given resources); `default` may succeed quickly, or if not will try the `hard` strategy; and `hard` will run more slowly, from the beginning trying to succeed.

Strategy options are merely options that set a number of the options seen in Chapter 4, all at once; they are parsed in **exactly** the same way as other options; order **is** important. It is usual to specify one strategy option and possibly follow it with a number of options defined in Chapter 4, some of which may over-ride those options set by the strategy option. Please refer to the introductory sections of Chapter 4, paying particular attention to Sections 4.2, 4.5, and 4.6, which give various warnings, hints and information on the interpretation of options.

There are eight strategy options. Each is passed without a value (see Section 4.6) except for `sims` which expects one of the integer values: 1, 3, 5, 7, or 9; and, `felsch` can accept a value of 0 or 1, where 0 has the same effect as passing `felsch` with no value. Thus the eight strategy options define thirteen standard strategies; these are listed in the table below, along with all but three of the options (of Chapter 4) that they set. Additionally, each strategy sets `path = 0`, `psize = 256`, and `dsize = 1000`. Recall `mend`, `look` and `com` abbreviate `mendelsohn` (see 4.13.5), `lookahead` (see 4.15.2) and `compaction` (see 4.17.5), respectively.

	option									
strategy	row	mend	no	look	com	ct	rt	fill	pmode	dmode
default	1	0	-1	0	10	0	0	0	3	4
easy	1	0	0	0	100	0	1000	1	0	0
felsch := 0	0	0	0	0	10	1000	0	1	0	4
felsch := 1	0	0	-1	0	10	1000	0	0	3	4
hard	1	0	-1	0	10	1000	1	0	3	4
hlt	1	0	0	1	10	0	1000	1	0	0
purec	0	0	0	0	100	1000	0	1	0	4
purer	0	0	0	0	100	0	1000	1	0	0
sims := 1	1	0	0	0	10	0	1000	1	0	0
sims := 3	1	0	0	0	10	0	-1000	1	0	4
sims := 5	1	1	0	0	10	0	1000	1	0	0
sims := 7	1	1	0	0	10	0	-1000	1	0	4
sims := 9	0	0	0	0	10	1000	0	1	0	4

Note that we explicitly (re)set all of the listed enumerator options in all of the predefined strategies, even though some of them have no effect. For example, the `fill` value is irrelevant in HLT-type enumeration (see Section 3.1). The idea behind this is that, if you later change some options individually, then the enumeration retains the "flavour" of the last selected predefined strategy.

Note also that other options which may effect an enumeration are left untouched by setting one of the predefined strategies; for example, the values of `max` (see 4.17.6) and `asis` (see 4.13.1). These options have an effect which is independent of the selected strategy.

Note that, apart from the `felsch := 0` and `sims := 9` strategies, all of the strategies are distinct, although some are very similar.

5.1 The Strategies in Detail

Please note that the strategies are based on various **enumeration styles**: **C style**, **Cr style**, **CR style**, **R style**, **R* style**, **Rc style**, **R/C style** and **defaulted R/C style**, all of which are described in detail in Section 3.1.

1 ► `default`

Selects the default strategy. (Shortest abbreviation: `def`.)

This strategy is based on the **defaulted R/C style** (see Section 3.1). The idea here is that we assume that the enumeration is "easy" and start out in **R style**. If it turns out not to be easy, then we regard it as "hard", and switch to **CR style**, after performing a `lookahead` (see 4.15.2) on the entire table.

2 ► `easy`

Selects an "easy" R style strategy.

If this strategy is selected, we follow a HLT-type enumeration style, i.e. **R style** (see Section 3.1), but turn `lookahead` (see 4.15.2) and `compaction` (see 4.17.5) off. For small and/or easy enumerations, this strategy is likely to be the fastest.

3 ► `felsch`
► `felsch:=`*val*

Selects a Felsch strategy; *val* should be 0 or 1. (Shortest abbreviation: `fel`.)

Here a **C style** (see Section 3.1) enumeration is selected. Assigning `felsch` 0 or no value selects a pure Felsch strategy, and a value of 1 selects a Felsch strategy with all relators in the subgroup, i.e. `no` $= -1$ (see 4.13.4), and turns gap-`filling` (see 4.14.1) on.

4 ► `hard`

Selects a mixed **R style** and **C style** strategy.

In many "hard" enumerations, a mixture of **R style** and **C style** (see Section 3.1) definitions, all tested in all essentially different positions, is appropriate. This option selects such a mixed strategy. The idea here is that most of the work is done C style (with the relators in the subgroup, i.e. `no` $= -1$ (see 4.13.4), and with gap-`filling` (see 4.14.1) on), but that every 1000 C style definitions a further coset number is applied to all relators.

Guru Notes: The $1000/1$ mix is not necessarily optimal, and some experimentation may be needed to find an acceptable balance (see, for example, [HR01]). Note also that, the longer the total length of the presentation, the more work needs to be done for each coset number application to the relators; one coset number application can result in more than 1000 definitions, reversing the balance between R style and C style definitions.

5 ► `hlt`

Selects ACE's standard HLT strategy.

Unlike Sims' [Sim94] default HLT strategy, `hlt` sets the `lookahead` option (see 4.15.2). However, the option sequence "`hlt, lookahead := 0`" easily achieves Sims' default HLT strategy (recall, the ordering of options is respected; see Section 4.4).

This is an **R style** (see Section 3.1) strategy.

6 ▶ `purec`

Sets the strategy to basic **C style** (see Section 3.1).

In this strategy there is no `compaction` (see 4.17.5), no gap-`filling` (see 4.14.1) and no relators in subgroup, i.e. `no` = 0 (see 4.13.4).

7 ▶ `purer`

Sets the strategy to basic **R style** (see Section 3.1).

In this strategy there is no `mendelsohn` (see 4.13.5), no `compaction` (see 4.17.5), no `lookahead` (see 4.15.2) and no `row`-filling (see 4.15.1).

8 ▶ `sims:=`*val*

Sets a Sims strategy; *val* should be one of 1, 3, 5, 7 or 9.

In his book [Sim94], Sims discusses (and lists in Table 5.5.1) ten standard enumeration strategies. The Sims' strategies are effectively `hlt` (see 5.1.5) without `lookahead` (see 4.15.2), with or without `mendelsohn` (see 4.13.5) set, in R (`rt` positive, `ct := 0`) or R* style (`rt` negative, `ct := 0`); and `felsch` (see 5.1.3); all either with or without (`lenlex`) table standardisation (see Section 3.4 and 6.7.2 or D.6.2) as the enumeration proceeds. ACE does not implement table standardisation during an enumeration, and so only provides the odd-numbered strategies of Sims (ACE's numbering coincides with that of Sims).

With care, it is often possible to duplicate the statistics given in [Sim94] for his odd-numbered strategies and it is also possible (using the interactive facilities) to approximate his even-numbered strategies. Examples and a more detailed exposition of the Sims strategies are given in Section C.2.

6 Functions for Using ACE Interactively

The user will probably benefit most from interactive use of ACE by setting the `InfoLevel` of `InfoACE` to at least 3 (see 1.9.3), particularly if she uses the `messages` option with a non-zero value.

Have you read the various options warnings yet? If not, please take the time to read Section 1.7 which directs you to various important sections of Chapter 4.

We describe in this chapter the functions that manipulate and initiate interactive ACE processes.

An interactive ACE process is initiated by `ACEStart` and terminated via `ACEQuit`; these functions are described in Section 6.1. `ACEStart` also has forms that manipulate an already started interactive ACE process. `ACEStart` always returns a positive integer *i*, which identifies the interactive ACE process that was initiated or manipulated.

Most functions (there is one `ACEStart` exception), that manipulate an already started interactive ACE process, have a form where the first argument is the integer *i* returned by the initiating `ACEStart` command, and a second form with one argument fewer (where the integer *i* is discovered by a default mechanism, namely by determining the least integer *i* for which there is a currently active interactive ACE process). We will thus commonly say that "for the *i*th (or default) interactive ACE process" a certain function performs a given action. In each case, it is an error, if *i* is not the index of an active interactive process, or there are no current active interactive processes.

Notes: The global method of passing options (via `PushOptions`), should not be used with any of the interactive functions. In fact, the `OptionsStack` should be empty at the time any of the interactive functions is called.

On `quitting` GAP, `ACEQuitAll();` is executed, which terminates all active interactive ACE processes. If GAP is killed without `quitting`, before all interactive ACE processes are terminated, **zombie** processes (still living **child** processes whose **parents** have died), will result. Since zombie processes do consume resources, in such an event, the responsible computer user should seek out and kill the still living `ace` children (e.g. by piping the output of a `ps` with appropriate options, usually `aux` or `ef`, to `grep ace`, to find the process ids, and then using `kill`; try `man ps` and `man kill` if these hints are unhelpful).

6.1 Starting and Stopping Interactive ACE Processes

1 ▶ `ACEStart(` *fgens*`,` *rels*`,` *sgens* `[:`*options*`] )` F
▶ `ACEStart(` *i* `[:`*options*`] )` F
▶ `ACEStart( [:`*options*`] )` F
▶ `ACEStart(` *i*`,` *fgens*`,` *rels*`,` *sgens* `[:`*options*`] )` F
▶ `ACEStart( 0 [:`*options*`] )` F
▶ `ACEStart( 0,` *fgens*`,` *rels*`,` *sgens* `[:`*options*`] )` F
▶ `ACEStart( 0,` *i* `[:`*options*`] )` F
▶ `ACEStart( 0,` *i*`,` *fgens*`,` *rels*`,` *sgens* `[:`*options*`] )` F

The variables are: *i*, a positive integer numbering from 1 that represents (indexes) an already running interactive ACE process; *fgens*, a list of free generators; *rels*, a list of words in the generators *fgens* giving relators for a finitely presented group; and *sgens*, a list of subgroup generators, again expressed as words in the free generators *fgens*. Each of *fgens*, *rels* and *sgens* are given in the standard GAP format for finitely presented groups (See Chapter 47 of the GAP Reference Manual).

All forms of `ACEStart` accept options described in Chapters 4 and 5, and Appendix D, which are listed behind a colon in the usual way (see 4.12 in the GAP Reference Manual). The reader is strongly encouraged to read the introductory sections of Chapter 4, with regard to options. The global mechanism (via `PushOptions`) of passing options is **not** recommended for use with the interactive ACE interface functions; please ensure the `OptionsStack` is empty before calling an interactive ACE interface function.

The return value (for all forms of `ACEStart`) is an integer (numbering from 1) which represents the running process. It is possible to have more than one interactive process running at once. The integer returned may be used to index which of these processes an interactive ACE interface function should be applied to.

The first four forms of `ACEStart` insert a `start` (see D.3.2) directive after the user's options to invoke an enumeration. The last four forms, with `0` as first argument, do not insert a `start` directive. Moreover, the last 3 forms of `ACEStart`, with `0` as first argument only differ from the corresponding forms of `ACEStart` without the `0` argument, in that they do not insert a `start` directive. `ACEStart(0)`, however, is special; unlike the no-argument form of `ACEStart` it invokes a new interactive ACE process. We will now further describe each form of `ACEStart`, in the order they appear above.

The first form of `ACEStart` (on three arguments) is the usual way to start an interactive ACE process.

When `ACEStart` is called with one positive integer argument *i* it starts a new enumeration on the *i*th running process, i.e. it scrubs a previously generated table and starts from scratch with the same parameters (i.e. the same arguments and options); except that if new options are included these will modify those given previously. The only reason for doing such a thing, without new options, is to perhaps compare timings of runs (a second run is quicker because memory has already been allocated). If you are interested in this sort of information, however, you may be better off dealing directly with the standalone.

When `ACEStart` is called with no arguments it finds the least positive integer *i* for which an interactive process is running and applies `ACEStart(`*i*`)`. (Most users will only run one interactive process at a time. Hence, `ACEStart()` will be a useful shortcut for `ACEStart(1)`.)

The fourth form of `ACEStart` on four arguments, invokes a new enumeration on the *i*th running process, with new generators *fgens*, relators *rels* and subgroup generators *sgens*. This is provided so a user can re-use an already running process, rather than start a new process. This may be useful when pseudo-ttys are a scarce resource. See the notes for the non-interactive `ACECosetTable` (1.2.1) which demonstrates an application of a usage of this and the following form of `ACEStart` in a loop.

The fifth form of `ACEStart` (on the one argument: `0`) initiates an interactive ACE process, processes any user options, but does not insert a `start` (see D.3.2) directive. This form is mainly for gurus who are familiar with the ACE standalone and who wish, at least initially, to communicate with ACE using the primitive read/write tools of Section 6.8. In this case, after the group generators, relators, and subgroup generators have been set in the ACE process, invocations of any of `ACEGroupGenerators` (see 6.5.1), `ACERelators` (see 6.5.2), `ACESubgroupGenerators` (see 6.5.3), or `ACEParameters` (see 6.5.10) will establish the corresponding GAP values. Be warned, though, that unless one of the general ACE modes (see Section 6.2): `ACEStart` (without a zero argument), `ACERedo` (see 6.2.3) or `ACEContinue` (see 6.2.2), or one of the mode options: `start` (see D.3.2), `redo` (see D.3.3) or `continu` (see D.3.4), has been invoked since the last change of any parameter options (see 4.12.1), some of the values reported by `ACEParameters` may well be **incorrect**.

The sixth form of `ACEStart` (on four arguments), is like the first form of `ACEStart` (on three arguments), except that it does not insert a `start` (see D.3.2) directive. It initiates an interactive ACE process, with a presentation defined by its last 3 arguments.

The seventh form of `ACEStart` (on two arguments), is like the second form of `ACEStart` (on one argument), except that it does not insert a `start` (see D.3.2) directive. It processes any new options for the *i*th interactive ACE process. `ACEStart(0, `*i*` [: `*options*` ] )`, is similar to `SetACEOptions(`*i*` [: `*options*` ] )` (see 6.5.8), but unlike the latter does not invoke a general mode (see Section 6.2).

The last form of `ACEStart` (on five arguments), is like the fourth form of `ACEStart` (on four arguments), except that it does not insert a `start` (see D.3.2) directive. It re-uses an existing interactive ACE process, with a new presentation. There is no form of `ACEStart` with the same functionality as this form, where the *i* argument is omitted.

Note: When an interactive ACE process is initiated by `ACEStart` a process number *i* is assigned to it (the integer *i* returned by the `ACEStart` command), an ACE (binary) process (in the UNIX sense) is started up, a GAP iostream is assigned to communicate with the ACE (binary) process, and the essential "defining" data associated with the interactive ACE process is saved in `ACEData.io[`*i*`]` (see 1.8.1 for precisely what is saved).

2 ► `ACEQuit( i )` F
► `ACEQuit()` F

terminate an interactive ACE process, where *i* is the integer returned by `ACEStart` when the process was started. If the second form is used (i.e. without arguments) then the interactive process of least index that is still running is terminated.

Note: `ACEQuit(`*i*`)` terminates the ACE (binary) process of interactive ACE process *i*, and closes its GAP iostream, and unbinds the record `ACEData.io[`*i*`]` (see 6.1.1 note).

It can happen that the ACE (binary) process, and hence the GAP iostream assigned to communicate with it, can die, e.g. by the user typing a `Ctrl-C` while the ACE (binary) process is engaged in a long calculation. `IsACEProcessAlive` (see 6.3.3) is provided to check the status of the GAP iostream (and hence the status of the ACE (binary) process it was communicating with); in the case that it is indeed dead, `ACEResurrectProcess` (see 6.3.4) may be used to start a new ACE (binary) process and assign a new GAP iostream to communicate with it, by using the "defining" data of the interactive ACE process saved in `ACEData.io[`*i*`]`.

3 ► `ACEQuitAll()` F

is provided as a convenience, to terminate all active interactive ACE processes with a single command. It is equivalent to executing `ACEQuit(`*i*`)` for all active interactive ACE processes *i* (see 6.1.2).

6.2 General ACE Modes

For our purposes, we define an interactive ACE interface command to be an ACE mode, if it delivers an enumeration result (see Section A.2). Thus, `ACEStart` (see 6.1.1), except when called with the argument 0, and the commands `ACERedo` and `ACEContinue` which we describe below are ACE modes; we call these **general** ACE **modes**. Additionally, there are two other commands which deliver enumeration results: `ACEAllEquivPresentations` (see 6.4.1) and `ACERandomEquivPresentations` (see 6.4.2); we call these "experimentation" ACE modes and describe them in Section 6.4.

Guru Note: The ACE standalone uses the term `mode` in a slightly different sense. There, the commands: `start`, `redo` or `continue`, put the ACE enumerator in `start mode`, `redo mode` or `continue mode`. In this manual, we have used the term to mean the command itself, and generalised it to include any command that produces enumeration results.

After changing any of ACE's parameters, one of three **general modes** is possible: one may be able to "continue" via `ACEContinue` (see 6.2.2), or "redo" via `ACERedo` (see 6.2.3), or if neither of these is possible one may have to re-"start" the enumeration via `ACEStart` (see 6.1.1). Generally, the appropriate mode is invoked automatically when options are changed; so most users should be able to ignore the following three functions.

1 ► `ACEModes( i )` F
► `ACEModes()` F

for the *i*th (or default) interactive ACE process, return a record whose fields are the modes `ACEStart`, `ACEContinue` and `ACERedo`, and whose values are `true` if the mode is possible for the process and `false` otherwise.

2 ► `ACEContinue( i [:`*options*`] )` F
► `ACEContinue( [:`*options*`] )` F

for the *i*th (or default) interactive ACE process, apply any *options* and then "continue" the current enumeration, building upon the existing table. If a previous run stopped without producing a finite index you can, in principle, change any of the options and continue on. Of course, if you make any changes which invalidate the current table, you won't be allowed to `ACEContinue` and an error will be raised. However, after `quit`ting the `break`-loop, the interactive ACE process should normally still be active; after doing so, run `ACEModes` (see 6.2.1) to see which of `ACERedo` or `ACEStart` is possible.

3 ► `ACERedo( i [:options] )` F
► `ACERedo( [:options] )` F

for the *i*th (or default) interactive ACE process, apply any *options* and then "redo" the current enumeration from coset 1 (i.e., the subgroup), keeping any existing information in the table.

Notes: The difference between `ACEContinue` and `ACERedo` is somewhat technical, and the user should regard it as a mode that is a little more expensive than `ACEContinue` but cheaper than `ACEStart`. `ACERedo` is really intended for the case where additional relators and/or subgroup generators have been introduced; the current table, which may be incomplete or exhibit a finite index, is still "valid". However, the new data may allow the enumeration to complete, or cause a collapse to a smaller index. In some cases, `ACERedo` may not be possible and an error will be raised; in this case, `quit` the `break`-loop, and try `ACEStart`, which will discard the current table and re-"start" the enumeration.

6.3 Interactive ACE Process Utility Functions and Interruption of an Interactive ACE Process

1 ► `ACEProcessIndex( i )` F
► `ACEProcessIndex()` F

With argument *i*, which must be a positive integer, `ACEProcessIndex` returns *i* if it corresponds to an active interactive process, or raises an error. With no arguments it returns the default active interactive process or returns `fail` and emits a warning message to `Info` at `InfoACE` or `InfoWarning` level 1.

Note: Essentially, an interactive ACE process *i* is "active" if `ACEData.io[i]` is bound (i.e. we still have some data telling us about it). Also see 6.1.1 note.

2 ► `ACEProcessIndices()` F

returns the list of integer indices of all active interactive ACE processes (see 6.3.1 for the meaning of "active").

3 ► `IsACEProcessAlive( i )` F
► `IsACEProcessAlive()` F

return `true` if the GAP iostream of the *i*th (or default) interactive ACE process started by `ACEStart` is alive (i.e. can still be written to), or `false`, otherwise. (See the notes for 6.1.1 and 6.1.2.)

If the user does not yet have a `gap>` prompt then usually ACE is still away doing something and an ACE interface function is still waiting for a reply from ACE. Typing a `Ctrl-C` (i.e. holding down the `Ctrl` key and typing `c`) will stop the waiting and send GAP into a `break`-loop, from which one has no option but to `quit;`. The typing of `Ctrl-C`, in such a circumstance, usually causes the stream of the interactive ACE process to die; to check this we provide `IsACEProcessAlive` (see 6.3.3). If the stream of an interactive ACE process, indexed by *i*, say, has died, it may still be possible to recover enough of the state, before the death of the stream, from the information stored in the `ACEData.io[i]` record (see Section 1.8). For such a purpose, we have provided `ACEResurrectProcess` (see 6.3.4).

The GAP iostream of an interactive ACE process will also die if the ACE binary has a segmentation fault. We do hope that this never happens to you, but if it does and the failure is reproducible, then it's a bug and we'd like to know about it. Please read the file `README.md` that comes with the ACE package to find out what to include in a bug report and who to email it to.

4 ► `ACEResurrectProcess( i [: options] )` F
► `ACEResurrectProcess( [: options] )` F

re-generate the GAP iostream of the *i*th (or default) interactive ACE process started by `ACEStart` (see 6.1.1, the final note, in particular), and try to recover as much as possible of the previous state from saved values of the process's arguments and parameter options. The possible *options* here are `use` and `useboth` which are described in detail below.

The arguments of the *i*th interactive ACE process are stored in `ACEData.io[`*i*`].args`, a record with fields `fgens`, `rels` and `sgens`, which are the GAP group generators, relators and subgroup generators, respectively (see Section 1.8). Option information is saved in `ACEData.io[`*i*`].options` when a user uses an interactive ACE interface function with options or uses `SetACEOptions` (see 6.5.8). Parameter option information is saved in `ACEData.io[`*i*`].parameters` if `ACEParameters` (see 6.5.10) is used to extract from ACE the current values of the ACE parameter options (this is generally less reliable unless one of the general ACE modes (see Section 6.2), has been run previously).

By default, `ACEResurrectProcess` recovers parameter option information from `ACEData.io[`*i*`].options` if it is bound, or from `ACEData.io[`*i*`].parameters` if it is bound, otherwise. The `ACEData.io[`*i*`].options` record, however, is first filtered for parameter and strategy options (see Sections 4.12.1 and 4.9) and the `echo` option (see 4.11.12). To alter this behaviour, the user is provided two options:

`use := ` *useList*
: *useList* may contain one or both of `"options"` and `"parameters"`. By default, `use = ["options", "parameters"]`.

`useboth`
: (A boolean option). By default, `useboth = false`.

If `useboth = true`, `SetACEOptions` (see 6.5.8) is applied to the *i*th interactive ACE process with each `ACEData.io[`*i*`].(`*field*`)` for each *field* (`"options"` or `"parameters"`) that is bound and in *useList*, in the order implied by *useList*. If `useboth = false`, `SetACEOptions` is applied with `ACEData.io[`*i*`].(`*field*`)` for only the first *field* that is bound in *useList*. The current value of the `echo` option is also preserved (no matter what values the user chooses for the `use` and `useboth` options).

Notes: Do not use general ACE options with `ACEResurrectProcess`; they will only be superseded by those options recovered from `ACEData.io[`*i*`].options` and/or `ACEData.io[`*i*`].parameters`. Instead, call `SetACEOptions` first (or afterwards). When called prior to `ACEResurrectProcess`, `SetACEOptions` will emit a warning that the stream is dead; despite this, the `ACEData.io[`*i*`].options` **will** be updated.

`ACEResurrectProcess` does **not** invoke an ACE mode (see Section 6.2). This leaves the user free to use `SetACEOptions` (which does invoke an ACE mode) to further modify options afterwards.

5 ► `ToACEGroupGenerators(` *fgens* `)` F

This function produces, from a GAP list *fgens* of free group generators, the ACE directive string required by the `group` (see D.2.1) option. (The `group` option may be used to define the equivalent of *fgens* in an ACE process.)

6 ► `ToACEWords(` *fgens*`,` *words* `)` F

This function produces, from a GAP list *words* in free group generators *fgens*, a string that represents those *words* as an ACE list of words. `ToACEWords` may be used to provide suitable values for the options `relators` (see D.2.2), `generators` (see D.2.3), `sg` (see D.2.4), and `rl` (see D.2.5).

6.4 Experimentation ACE Modes

Now we describe the two **experimentation modes**. The term **mode** was defined in Section 6.2.

1 ► `ACEAllEquivPresentations(` *i*`,` *val* `)` F
► `ACEAllEquivPresentations(` *val* `)` F

for the *i*th (or default) interactive ACE process, generates and tests an enumeration for combinations of relator ordering, relator rotations, and relator inversions; *val* is in the integer range 1 to 7.

The argument *val* is considered as a binary number. Its three bits are treated as flags, and control relator rotations (the 2^0 bit), relator inversions (the 2^1 bit) and relator orderings (the 2^2 bit), respectively; where 1 means “active” and 0 means “inactive”. (See below for an example).

Before we describe the GAP output of `ACEAllEquivPresentations` let us spend some time considering what happens before the ACE binary output is parsed.

The `ACEAllEquivPresentations` command first performs a "priming run" using the options as they stand. In particular, the `asis` and `messages` options are honoured.

It then turns `asis` (see 4.13.1) on and `messages` (see 4.18.1) off (i.e. sets `messages` to 0), and generates and tests the requested equivalent presentations. The maximum and minimum values attained by `m` (the maximum number of coset numbers defined at any stage) and `t` (the total number of coset numbers defined) are tracked, and each time the statistics are better than what we've already seen, the ACE binary emits a summary result line for the relators used. See Appendix A for a discussion of the statistics `m` and `t`. To observe these messages set the `InfoLevel` of `InfoACE` to 3; and it's **recommended** that you do this so that you get some idea of what the ACE binary is doing.

The order in which the equivalent presentations are generated and tested has no particular significance, but note that the presentation as given **after** the initial priming run) is the **last** presentation to be generated and tested, so that the group's relators are left "unchanged" by running the `ACEAllEquivPresentations` command.

As discussed by Cannon, Dimino, Havas and Watson [CDHW73] and Havas and Ramsay [HR01] such equivalent presentations can yield large variations in the number of coset numbers required in an enumeration. For this command, we are interested in this variation.

After the final presentation is run, some additional status information messages are printed to the ACE output:

- the number of runs which yielded a finite index;
- the total number of runs (excluding the priming run); and
- the range of values observed for `m` and `t`.

As an example (drawn from the discussion in [HR99]) consider the enumeration of the 448 coset numbers of the subgroup $\langle a^2, a^{-1}b\rangle$ of the group

$$(8,7 \mid 2,3) = \langle a,b \mid a^8 = b^7 = (ab)^2 = (a^{-1}b)^3 = 1\rangle\cdot$$

There are $4! = 24$ relator orderings and $2^4 = 16$ combinations of relator or inverted relator. Exponents are taken into account when rotating relators, so the relators given give rise to 1, 1, 2 and 2 rotations respectively, for a total of $1\cdot 1\cdot 2\cdot 2 = 4$ combinations. So, for *val* $= 7$ (resp. 3), $24\cdot 16\cdot 4 = 1536$ (resp. $16\cdot 4 = 64$) equivalent presentations are tested.

Now we describe the output of `ACEAllEquivPresentations`; it is a record with fields:

`primingResult`
: the ACE enumeration result message (see Section A.2) of the priming run;

`primingStats`
: the enumeration result of the priming run as a GAP record with fields `index`, `cputime`, `cputimeUnits`, `activecosets`, `maxcosets` and `totcosets`, exactly as for the record returned by `ACEStats` (see 1.3.1);

`equivRuns`
: a list of data records, one for each progressively "best" run, where each record has fields:

  `rels`
  : the relators in the order used for the run,

  `enumResult`
  : the ACE enumeration result message (see Section A.2) of the run, and

  `stats`
  : the enumeration result as a GAP record exactly like the record returned by `ACEStats` (see 1.3.1);

`summary`
: a record with fields:

`successes`
the total number of successful (i.e. having finite enumeration index) runs,

`runs`
the total number of equivalent presentation runs executed,

`maxcosetsRange`
the range of values as a GAP list inside which each `equivRuns[`*i*`].maxcosets` lies, and

`totcosetsRange`
the range of values as a GAP list inside which each `equivRuns[`*i*`].totcosets` lies.

Notes: In general, the length of the `equivRuns` field list will be less than the number of runs executed.

There is no way to stop the `ACEAllEquivPresentations` command before it has completed, other than killing the task. So do a reality check beforehand on the size of the search space and the time for each enumeration. If you are interested in finding a "good" enumeration, it can be very helpful, in terms of running time, to put a tight limit on the number of coset numbers via the `max` option. You may also have to set `compaction` = 100 to prevent time-wasting attempts to recover space via compaction. This maximises throughput by causing the "bad" enumerations, which are in the majority, to overflow quickly and abort. If you wish to explore a very large search-space, consider firing up many copies of ACE, and starting each with a "random" equivalent presentation. Alternatively, you could use the `ACERandomEquivPresentations` command.

2 ► `ACERandomEquivPresentations(` *i*`,` *val* `)` F
► `ACERandomEquivPresentations(` *val* `)` F
► `ACERandomEquivPresentations(` *i*`, [`*val*`] )` F
► `ACERandomEquivPresentations( [`*val*`] )` F
► `ACERandomEquivPresentations(` *i*`, [`*val*`,` *Npresentations*`] )` F
► `ACERandomEquivPresentations( [`*val*`,` *Npresentations*`] )` F

for the *i*th (or default) interactive ACE process, generates and tests up to *Npresentations* (or 8, in the first 4 forms) random presentations; *val*, an integer in the range 1 to 7, acts as for `ACEAllEquivPresentations` and *Npresentations*, when given, should be a positive integer.

The routine first turns `asis` (see 4.13.1) on and `messages` (see 4.18.1) off (i.e. sets `messages` to 0), and then generates and tests the requested number of random equivalent presentations. For each presentation, the relators used and the summary result line are printed by ACE. To observe these messages set the `InfoLevel` of `InfoACE` to at least 3.

`ACERandomEquivPresentations` parses the ACE messages, translating them to GAP, and thus returns a list of records (similar to the field `equivRuns` of the returned record of `ACEAllEquivPresentations`). Each record of the returned list is the data derived from a presentation run and has fields:

`rels`
the relators in the order used for the run,

`enumResult`
the ACE enumeration result message (see Section A.2) of the run, and

`stats`
the enumeration result as a GAP record exactly like the record returned by `ACEStats` (see 1.3.1).

Notes: The relator inversions and rotations are "genuinely" random. The relator permuting is a little bit of a kludge, with the "quality" of the permutations tending to improve with successive presentations. When the `ACERandomEquivPresentations` command completes, the presentation active is the **last** one generated.

Guru Notes: It might appear that neglecting to restore the original presentation is an error. In fact, it is a useful feature! Suppose that the space of equivalent presentations is too large to exhaustively test. As noted in the entry for `ACEAllEquivPresentations`, we can start up multiple copies of `ACEAllEquivPresentations` at random points in the search-space. Manually generating `random` equivalent presentations to serve as starting-points is tedious and error-prone. The `ACERandomEquivPresentations` command provides a simple solution; simply run `ACERandomEquivPresentations(`*i*`, 7);` before `ACEAllEquivPresentations(`*i*`, 7);`.

6.5 Interactive Query Functions and an Option Setting Function

1 ▶ `ACEGroupGenerators( i )` F
▶ `ACEGroupGenerators()` F

return the GAP group generators of the *i*th (or default) interactive ACE process. If no generators have been saved for the interactive ACE process, possibly because the process was started via `ACEStart(0);` (see 6.1.1), the ACE process is interrogated, and the equivalent in GAP is saved and returned. Essentially, `ACEGroupGenerators(`*i*`)` interrogates ACE and establishes `ACEData.io[`*i*`].args.fgens`, if necessary, and returns `ACEData.io[`*i*`].args.fgens`. As a side-effect, if any of the remaining fields of `ACEData.io[`*i*`].args` or `ACEData.io[`*i*`].acegens` are unset, they are also set. Note that GAP provides `GroupWithGenerators` (see 39.2.3 in the GAP Reference Manual) to establish a free group on a given set of already-defined generators.

2 ▶ `ACERelators( i )` F
▶ `ACERelators()` F

return the GAP relators of the *i*th (or default) interactive ACE process. If no relators have been saved for the interactive ACE process, possibly because the process was started via `ACEStart(0);` (see 6.1.1), the ACE process is interrogated, the equivalent in GAP is saved and returned. Essentially, `ACERelators(`*i*`)` interrogates ACE and establishes `ACEData.io[`*i*`].args.rels`, if necessary, and returns `ACEData.io[`*i*`].args.rels`. As a side-effect, if any of the remaining fields of `ACEData.io[`*i*`].args` or `ACEData.io[`*i*`].acegens` are unset, they are also set.

3 ▶ `ACESubgroupGenerators( i )` F
▶ `ACESubgroupGenerators()` F

return the GAP subgroup generators of the *i*th (or default) interactive ACE process. If no subgroup generators have been saved for the interactive ACE process, possibly because the process was started via `ACEStart(0);` (see 6.1.1), the ACE process is interrogated, the equivalent in GAP is saved and returned. Essentially, `ACESubgroupGenerators(`*i*`)` interrogates ACE and establishes `ACEData.io[`*i*`].args.sgens`, if necessary, and returns `ACEData.io[`*i*`].args.sgens`. As a side-effect, if any of the remaining fields of `ACEData.io[`*i*`].args` or `ACEData.io[`*i*`].acegens` are unset, they are also set.

4 ▶ `DisplayACEArgs( i )` F
▶ `DisplayACEArgs()` F

display the arguments (i.e. *fgens*, *rels* and *sgens*) of the *i*th (or default) process started by `ACEStart`. In fact, `DisplayACEArgs(`*i*`)` is just a pretty-printer of the `ACEData.io[`*i*`].args` record. Use `GetACEArgs` (see 6.5.7) in assignments. Unlike `ACEGroupGenerators` (see 6.5.1), `ACERelators` (see 6.5.2) and `ACESubgroupGenerators` (see 6.5.3), `DisplayACEArgs` does not have the side-effect of setting any of the fields of `ACEData.io[`*i*`].args` if they are unset.

5 ▶ `GetACEArgs( i )` F
▶ `GetACEArgs()` F

return a record of the current arguments (i.e. *fgens*, *rels* and *sgens*) of the *i*th (or default) process started by `ACEStart`. In fact, `GetACEOptions(`*i*`)` simply returns the `ACEData.io[`*i*`].args` record, or an empty record if that record is unbound. Unlike `ACEGroupGenerators` (see 6.5.1), `ACERelators` (see 6.5.2) and `ACESubgroupGenerators` (see 6.5.3), `GetACEOptions` does not have the side-effect of setting any of the fields of `ACEData.io[`*i*`].args` if they are unset.

6 ▶ `DisplayACEOptions( i )` F
▶ `DisplayACEOptions()` F

display the options, explicitly set by the user, of the *i*th (or default) process started by `ACEStart`. In fact, `DisplayACEOptions(`*i*`)` is just a pretty-printer of the `ACEData.io[`*i*`].options` record. Use `GetACEOptions` (see 6.5.7) in assignments. Please note that no-value ACE options will appear with the assigned value `true` (see Section 4.6 for how the ACE interface functions interpret such options).

Notes: Any options set via `ACEWrite` (see 6.8.1) will **not** be displayed. Also, recall that if ACE is not given any options it uses the `default` strategy (see Section 4.5). To discover the various settings of the ACE Parameter Options (see 4.12.1) in vogue for the ACE process, use `ACEParameters` (see 6.5.10).

7 ► `GetACEOptions(` *i* `)` F
► `GetACEOptions()` F

return a record of the current options (those that have been explicitly set by the user) of the *i*th (or default) process started by `ACEStart`. Please note that no-value ACE options will appear with the assigned value `true` (see Section 4.6 for how the ACE interface functions interpret such options). The notes applying to `DisplayACEOptions` (see 6.5.6) also apply here.

8 ► `SetACEOptions(` *i* `[:`*options*`] )` F
► `SetACEOptions( [:`*options*`] )` F

modify the current options of the *i*th (or default) process started by `ACEStart`. Please ensure that the `OptionsStack` is empty before calling `SetACEOptions`, otherwise the options already present on the `OptionsStack` will also be "seen". All interactive ACE interface functions that accept options, actually call an internal version of `SetACEOptions`; so, it is generally important to keep the `OptionsStack` clear while working with ACE interactively.

After setting the options passed, the first mode of the following: `ACEContinue` (see 6.2.2), `ACERedo` (see 6.2.3) or `ACEStart` (see 6.1.1), that may be applied, is automatically invoked.

Since a user will sometimes have options in the form of a record (e.g. via `GetACEOptions`), we provide an alternative to the behind-the-colon syntax, in a manner .like `PushOptions`, for the passing of options via `SetACEOptions`:

9 ► `SetACEOptions(` *i*`,` *optionsRec* `)` F
► `SetACEOptions(` *optionsRec* `)` F

In this form, the record *optionsRec* is used to update the current options of the *i*th (or default) process started by `ACEStart`. Note that since *optionsRec* is a record each field must have an assigned value; in particular, no-value ACE options should be assigned the value `true` (see Section 4.6). Please don't mix these two forms of `SetACEOptions` with the previous two forms; i.e. do **not** pass both a record argument and options, since this will lead to options appearing in the wrong order; if you want to do this, make two separate calls to `SetACEOptions`, e.g. let's say you have a process like that started by:

```
gap> ACEExample("A5", ACEStart);
```

then the following demonstrates both usages of `SetACEOptions`:

```
gap> SetACEOptions( rec(echo := 2) );
gap> SetACEOptions( : hlt);
```

Each of the three commands above generates output; for brevity it has not been included.

Notes:

When `ACECosetTableFromGensAndRels` enters a `break`-loop (see 1.2.1), local versions of the second form of each of `DisplayACEOptions` and `SetACEOptions` become available. (Even though the names are similar and their function is analogous they are in fact different functions.)

10 ► `ACEParameters(` *i* `)` F
► `ACEParameters()` F

return a record of the current values of the ACE Parameter Options (see 4.12.1) of the *i*th (or default) process started by `ACEStart`, according to ACE. Please note that some options may be reported with incorrect values if they have been changed recently without following up with one of the modes `ACEContinue`, `ACERedo` or `ACEStart`. Together the commands `ACEGroupGenerators`, `ACERelators`, `ACESubgroupGenerators` and `ACEParameters` give the equivalent GAP information that is obtained in ACE with `sr := 1` (see D.5.7), which is the "Run Parameters" block obtained in the messaging output (observable when the `InfoLevel` of `InfoACE` is set to at least 3), when `messages` (see 4.18.1) is set a non-zero value.

Notes: One use for this function might be to determine the options required to replicate a previous run, but be sure that, if this is your purpose, any recent change in the parameter option values has been followed by an invocation of one of `ACEContinue` (see 6.2.2), `ACERedo` (see 6.2.3) or `ACEStart` (see 6.1.1).

As a side-effect, for ACE process *i*, any of the fields of `ACEData.io[`*i*`].args` or `ACEData.io[`*i*`].acegens` that are unset, are set.

11 ▶ `IsCompleteACECosetTable(` *i* `)` F
▶ `IsCompleteACECosetTable()` F

return, for the *i*th (or default) process started by `ACEStart`, `true` if ACE's current coset table is complete, or `false` otherwise.

Note: The completeness of the coset table of the *i*th interactive ACE process is determined by checking whether `ACEData.io[`*i*`].stats.index` is positive; a value of zero indicates the last enumeration failed to complete. The record `ACEData.io[`*i*`].stats` is what is returned by `ACEStats(`*i*`)` (see 1.3.1).

12 ▶ `ACEDisplayCosetTable(` *i* `)` F
▶ `ACEDisplayCosetTable()` F
▶ `ACEDisplayCosetTable(` *i*`,` `[`*val*`]` `)` F
▶ `ACEDisplayCosetTable(` `[`*val*`]` `)` F
▶ `ACEDisplayCosetTable(` *i*`,` `[`*val*`,` *last*`]` `)` F
▶ `ACEDisplayCosetTable(` `[`*val*`,` *last*`]` `)` F
▶ `ACEDisplayCosetTable(` *i*`,` `[`*val*`,` *last*`,` *by*`]` `)` F
▶ `ACEDisplayCosetTable(` `[`*val*`,` *last*`,` *by*`]` `)` F

compact and display the (possibly incomplete) coset table of the *i*th (or default) process started by `ACEStart`; *val* must be an integer, and *last* and *by* must be positive integers. In the first two forms of the command, the entire coset table is displayed, without orders or coset representatives. In the third and fourth forms, the absolute value of *val* is taken to be the last line of the table to be displayed (and 1 is taken to be the first); in the fifth and sixth forms, |*val*| is taken to be the first line of the table to be displayed, and *last* is taken to be the number of the last line to be displayed. In the last two forms, the table is displayed from line |*val*| to line *last* in steps of *by*. If *val* is negative, then the orders modulo the subgroup (if available) and coset representatives are displayed also.

Note: The coset table displayed will normally only be `lenlex` standardised if the call to `ACEDisplayCosetTable` is preceded by `ACEStandardCosetNumbering` (see 6.7.2). The options `lenlex` (see 4.11.4) and `semilenlex` (see 4.11.5) are only executed by `ACECosetTable` (see 1.2.1). The ACE binary does not provide `semilenlex` standardisation, and hence `ACEDisplayCosetTable` will never display a `semilenlex` standard coset table.

13 ▶ `ACECosetRepresentative(` *i*`,` *n* `)` F
▶ `ACECosetRepresentative(` *n* `)` F

return, for the *i*th (or default) process started by `ACEStart`, the coset representative of coset *n* of the current coset table held by ACE, where *n* must be a positive integer.

14 ▶ `ACECosetRepresentatives(` *i* `)` F
▶ `ACECosetRepresentatives()` F

return, for the *i*th (or default) process started by `ACEStart`, the list of coset representatives of the current coset table held by ACE.

15 ▶ `ACETransversal(` *i* `)` F
▶ `ACETransversal()` F

return, for the *i*th (or default) process started by `ACEStart`, the list of coset representatives of the current coset table held by ACE, if the current table is complete, and `fail` otherwise. Essentially, `ACETransversal(`*i*`) = ACECosetRepresentatives(`*i*`)` for a complete table.

16 ► `ACECycles( i )` F
► `ACECycles()` F
► `ACEPermutationRepresentation( i )` F
► `ACEPermutationRepresentation()` F

return, for the *i*th (or default) process started by `ACEStart`, a list of permutations corresponding to the group generators, (i.e., the permutation representation), if the current coset table held by ACE is complete or `fail`, otherwise. In the event of failure a message is emitted to `Info` at `InfoACE` or `InfoWarning` level 1.

17 ► `ACETraceWord( i, n, word )` F
► `ACETraceWord( n, word )` F

for the *i*th (or default) interactive ACE process started by `ACEStart`, trace *word* through ACE's coset table, starting at coset *n*, and return the final coset number if the trace completes, and `fail` otherwise. In Group Theory terms, if the cosets of a subgroup *H* in a group *G* are the subject of interactive ACE process *i* and the coset identified by that process by the integer *n* corresponds to some coset *Hx*, for some *x* in *G*, and *word* represents the element *g* of *G*, then, providing the current coset table is complete enough, `ACETraceWord( i, n, word )` returns the integer identifying the coset *Hxg*.

Notes: You may wish to compact ACE's coset table first, either explicitly via `ACERecover` (see 6.7.1), or, implicitly, via any function call that invokes ACE's compaction routine (see 6.7.1 note).

If you actually wanted ACE's coset representative, then, for a **compact** table, feed the output of `ACETraceWord` to `ACECosetRepresentative` (see 6.5.13).

18 ► `ACEOrders( i )` F
► `ACEOrders()` F
► `ACEOrders( i : suborder := suborder )` F
► `ACEOrders(: suborder := suborder )` F

for the *i*th (or default) interactive ACE process started by `ACEStart`, search for all coset numbers whose representatives' orders (modulo the subgroup) are either finite, or, if invoked with the `suborder` option, are multiples of *suborder*, where *suborder* should be a positive integer. `ACEOrders` returns a (possibly empty) list of records, each with fields `coset`, `order` and `rep`, which are respectively, the coset number, its order modulo the subgroup, and a representative for each coset number satisfying the criteria of the search.

Note: You may wish to compact ACE's coset table first, either explicitly via `ACERecover` (see 6.7.1), or, implicitly, via any function call that invokes ACE's compaction routine (see 6.7.1 note).

19 ► `ACEOrder( i, suborder )` F
► `ACEOrder( suborder )` F

for the *i*th (or default) interactive ACE process started by `ACEStart`, search for coset number(s) whose coset representatives have order modulo the subgroup a multiple of *suborder*. When *suborder* is a positive integer, `ACEOrder` returns just one record with fields `coset`, `order` and `rep`, which are respectively, the coset number, its order modulo the subgroup, and a representative for the first coset number satisfying the criteria of the search, or `fail` if there is no such coset number. The value of *suborder* may also be a negative integer, in which case, `ACEOrder( i, suborder )` is equivalent to `ACEOrders( i : suborder := |suborder|)`; or *suborder* may be zero, in which case, `ACEOrder( i, 0 )` is equivalent to `ACEOrders( i )`.

Note: You may wish to compact ACE's coset table first, either explicitly via `ACERecover` (see 6.7.1), or, implicitly, via any function call that invokes ACE's compaction routine (see 6.7.1 note).

20 ► `ACECosetOrderFromRepresentative( i, cosetrep )` F
► `ACECosetOrderFromRepresentative( cosetrep )` F

for the *i*th (or default) interactive ACE process return the order (modulo the subgroup) of the coset with representative *cosetrep*, a word in the free group generators.

Note: `ACECosetOrderFromRepresentative` calls `ACETraceWord` to determine the coset (number) to which *cosetrep* belongs, and then scans the output of `ACEOrders` to determine the order of the coset (number).

21 ► `ACECosetsThatNormaliseSubgroup(` *i*, *n* `)` F
► `ACECosetsThatNormaliseSubgroup(` *n* `)` F

for the *i*th (or default) interactive ACE process started by `ACEStart`, determine non-trivial (i.e. other than coset 1) coset numbers whose representatives normalise the subgroup.

- If $n > 0$, the list of the first *n* non-trivial coset numbers whose representatives normalise the subgroup is returned.
- If $n < 0$, a list of records with fields `coset` and `rep` which represent the coset number and a representative, respectively, of the first *n* non-trivial coset numbers whose representatives normalise the subgroup is returned.
- If $n = 0$, a list of records with fields `coset` and `rep` which represent the coset number and a representative, respectively, of all non-trivial coset numbers whose representatives normalise the subgroup is returned.

Note: You may wish to compact ACE's coset table first, either explicitly via `ACERecover` (see 6.7.1), or, implicitly, via any function call that invokes ACE's compaction routine (see 6.7.1 note).

22 ► `ACEStyle(` *i* `)` F
► `ACEStyle()` F

returns the current enumeration style as one of the strings: `"C"`, `"Cr"`, `"CR"`, `"R"`, `"R*"`, `"Rc"`, `"R/C"`, or `"R/C (defaulted)"` (see Section 3.1).

The next two functions of this section are really intended for ACE standalone gurus. To fully understand their output you will need to consult the standalone manual and the C source code.

23 ► `ACEDumpVariables(` *i* `)` F
► `ACEDumpVariables()` F
► `ACEDumpVariables(` *i*, [*level*] `)` F
► `ACEDumpVariables(` [*level*] `)` F
► `ACEDumpVariables(` *i*, [*level*, *detail*] `)` F
► `ACEDumpVariables(` [*level*, *detail*] `)` F

dump the internal variables of ACE of the *i*th (or default) process started by `ACEStart`; *level* should be one of 0, 1, or 2, and *detail* should be 0 or 1.

The value of *level* determines which of the three levels of ACE to dump. (You will need to read the standalone manual to understand what Levels 0, 1 and 2 are all about.) The value of *detail* determines the amount of detail (*detail* `= 0` means less detail). The first two forms of `ACEDumpVariables` (with no list argument) selects *level* `= 0`, *detail* `= 0`. The third and fourth forms (with a list argument containing the integer *level*) makes *detail* `= 0`. This command is intended for gurus; the source code should be consulted to see what the output means.

24 ► `ACEDumpStatistics(` *i* `)` F
► `ACEDumpStatistics()` F

dump ACE's internal statistics accumulated during the most recent enumeration of the *i*th (or default) process started by `ACEStart`, provided the ACE binary was built with the statistics package (which it is by default). Use `ACEBinaryVersion();` (see 6.5.25) to check for the inclusion of the statistics package. See the `enum.c` source file for the meaning of the variables.

25 ► `ACEBinaryVersion(` *i* `)` F
► `ACEBinaryVersion()` F

for the *i*th (or default) process started by `ACEStart`, print, via `Info` (at `InfoACE` level 1), version details of the ACE binary you are currently running, including what compiler flags were set when the executable was built, and also returns the version number of the binary as a string. Essentially the information obtained is what is obtained via ACE's options option (see D.5.5), and the returned value is what is stored in ACEData.version (see 1.8.1). A typical output, illustrating the default build, is:

```
gap> ACEBinaryVersion();
#I  No interactive ACE sessions are currently active
#I  ACE Binary Version: 3.001
#I  ACE 3.001        Sat Feb 27 11:27:15 2016
#I  =========================================
#I  Host information:
#I    name = banksia
#I  ACE 3.001 executable built:
#I    Wed Feb 24 15:25:26 AWST 2016
#I  Level 0 options:
#I    statistics package = on
#I    coinc processing messages = on
#I    dedn processing messages = on
#I  Level 1 options:
#I    workspace multipliers = decimal
#I  Level 2 options:
#I    host info = on
"3.001"
```

Notes: The ACE binary's banner may also appear in the output (if it has not already appeared). Unlike other ACE interface functions, the information obtained via `ACEBinaryVersion();` is absolutely independent of any enumeration. For this reason, we make it permissible to run `ACEBinaryVersion();` when there are no currently active interactive ACE processes; and, in such a case, `ACEBinaryVersion();` emits a warning that there are no interactive ACE sessions currently active and initiates (and closes again) its own stream to obtain the information from the ACE binary. For the current version of the ACE package (the GAP code component) use `ACEPackageVersion();` (see 1.11.1).

6.6 Interactive Versions of Non-interactive ACE Functions

1 ▶ `ACECosetTable(` *i* `[:`*options*`] )` F
▶ `ACECosetTable( [:`*options*`] )` F

return a coset table as a GAP object, in standard form (for GAP). These functions perform the same function as `ACECosetTableFromGensAndRels` and `ACECosetTable` on three arguments (see 1.2.1), albeit interactively, on the *i*th (or default) process started by `ACEStart`. If options are passed then an internal version of `ACEModes` is run to determine which of the general ACE modes (see Section 6.2) `ACEContinue`, `ACERedo` or `ACEStart` is possible; and (an internal version of) the first mode of these that is allowed is executed, to ensure the resultant table is correct for the current options.

2 ▶ `ACEStats(` *i* `[:`*options*`] )` F
▶ `ACEStats( [:`*options*`] )` F

perform the same function as `ACEStats` on three arguments (see 1.3.1 — non-interactive version), albeit interactively, on the *i*th (or default) process started by `ACEStart`. If options are passed then an internal version of `ACEModes` is run to determine which of the general ACE modes (see Section 6.2) `ACEContinue`, `ACERedo` or `ACEStart` is possible; and (an internal version of) the first mode of these that is allowed is executed, to ensure the resultant statistics are correct for the current options.

See Section B.3 for an example demonstrating both these functions within an interactive process.

3 ▶ `IsACEGeneratorsInPreferredOrder(` *i* `)` F
▶ `IsACEGeneratorsInPreferredOrder()` F

for the *i*th (or default) interactive ACE process started by `ACEStart`, return `true` if the group generators of that process, are in an order that will not be changed by ACE, and `false` otherwise. This function has greatest relevance to users who call `ACECosetTable` (see 6.6.1), with the `lenlex` option (see 4.11.4). For more details, see the discussion for the non-interactive version of `IsACEGeneratorsInPreferredOrder` (1.2.3), which is called with two arguments.

6.7 Steering ACE Interactively

1 ▶ `ACERecover(` *i* `)` F
▶ `ACERecover()` F

invoke the compaction routine on the coset table of the *i*th (or default) interactive ACE process started by `ACEStart`, in order to recover the space used by the dead coset numbers. A `CO` message line is printed if any rows of the coset table were recovered, and a `co` line if none were. (See Appendix A for the meanings of these messages.)

Note: The compaction routine is called automatically when any of `ACEDisplayCosetTable` (see 6.5.12), `ACECosetRepresentative` (see 6.5.13), `ACECosetRepresentatives` (see 6.5.14), `ACETransversal` (see 6.5.15), `ACECycles` (see 6.5.16), `ACEStandardCosetNumbering` (see 6.7.2), `ACECosetTable` (see 1.2.1) or `ACEConjugatesForSubgroupNormalClosure` (see 6.7.10), is invoked.

2 ▶ `ACEStandardCosetNumbering(` *i* `)` F
▶ `ACEStandardCosetNumbering()` F

compact and then do a `lenlex` standardisation (see Section 3.4) of the numbering of cosets in the coset table of the *i*th (or default) interactive ACE process started by `ACEStart`. That is, for a given ordering of the generators in the columns of the table, they produce a canonic table. A table that includes a column for each generator inverse immediately following the column for the corresponding generator, standardised according to the `lenlex` scheme, has the property that a row-major scan (i.e. a scan of the successive rows of the **body** of the table row by row, from left to right) encounters previously unseen cosets in numeric order. This function does not display the new table; use `ACEDisplayCosetTable` (see 6.5.12) for that.

Notes: In a `lenlex` canonic table, the coset representatives are ordered first according to length and then the lexicographic order defined by the order the generators and their inverses head the columns. Note that, unless special action is taken, ACE avoids having an involutory generator in the first column (by swapping the first two generators), except when there is only one generator, or when the second generator is also an involution; so the lexicographic order used by ACE need not necessarily correspond with the order in which the generators were first put to ACE. (We have used the term “involution” above; what we really mean is a generator `x` for which there is a relator `x*x` or `x^2`. Such a generator may, of course, turn out to actually be the identity.) The function `IsACEGeneratorsInPreferredOrder` (see 1.2.3) detects cases when ACE would swap the first two generators.

Standardising the coset numbering within ACE does **not** affect the GAP coset table obtained via `ACECosetTable` (see 1.2.1). If `ACECosetTable` is called without the `lenlex` option GAP’s default standardisation is applied after conversion of ACE’s output, which undoes an ACE standardisation. On the other hand, if `ACECosetTable` is called with the `lenlex` option then after a check and special action, if required, the equivalent of a call to `ACEStandardCosetNumbering` is invoked, irrespective of whether it has been done by the user beforehand. The check that is done is a call to `IsACEGeneratorsInPreferredOrder` (see 1.2.3) to ensure that ACE has not swapped the first two generators. The special action taken when the call to `IsACEGeneratorsInPreferredOrder` returns `false`, is the setting of the `asis` option (see 4.13.1) to 1 and the resubmission of the relators to ACE taking care not to submit the relator that determines the first generator as an involution as that generator squared (these two actions together avert ACE’s swapping of the first two generators), followed by the re-`start`ing of the enumeration.

Guru Notes: In five of the ten standard enumeration strategies of Sims [Sim94] (i.e. the five Sims strategies not provided by ACE), the table is standardised repeatedly. This is expensive computationally, but can result in fewer cosets being necessary. The effect of doing this can be investigated in ACE by (repeatedly) halting the enumeration (via restrictive options), standardising the coset numbering, and continuing (see Section C.2 for an example).

3 ► `ACEAddRelators( i, wordlist )` F
► `ACEAddRelators( wordlist )` F

add, for the *i*th (or default) interactive ACE process started by `ACEStart`, the words in the list *wordlist* to any relators already present, and automatically invoke `ACERedo`, if it can be applied, or otherwise `ACEStart`. Note that ACE sorts the resultant relator list, unless the `asis` option (see 4.13.1) has been set to 1; don't assume, unless `asis = 1`, that the new relators have been appended in user-provided order to the previously existing relator list. `ACEAddRelators` also returns the new relator list. Use `ACERelators` (see 6.5.2) to determine the current relator list.

4 ► `ACEAddSubgroupGenerators( i, wordlist )` F
► `ACEAddSubgroupGenerators( wordlist )` F

add, for the *i*th (or default) interactive ACE process started by `ACEStart`, the words in the list *wordlist* to any subgroup generators already present, and automatically invoke `ACERedo`, if it can be applied, or otherwise `ACEStart`. Note that ACE sorts the resultant subgroup generator list, unless the `asis` option (see 4.13.1) has been set to 1; don't assume, unless `asis = 1`, that the new subgroup generators have been appended in user-provided order to the previously existing subgroup generator list. `ACEAddSubgroupGenerators` also returns the new subgroup generator list. Use `ACESubgroupGenerators` (see 6.5.3) to determine the current subgroup generator list.

5 ► `ACEDeleteRelators( i, list )` F
► `ACEDeleteRelators( list )` F

for the *i*th (or default) interactive ACE process started by `ACEStart`, delete *list* from the current relators, if list is a list of words in the group generators, or those current relators indexed by the integers in *list*, if *list* is a list of positive integers, and automatically invoke `ACEStart`. `ACEDeleteRelators` also returns the new relator list. Use `ACERelators` (see 6.5.2) to determine the current relator list.

6 ► `ACEDeleteSubgroupGenerators( i, list )` F
► `ACEDeleteSubgroupGenerators( list )` F

for the *i*th (or default) interactive ACE process started by `ACEStart`, delete *list* from the current subgroup generators, if list is a list of words in the group generators, or those current subgroup generators indexed by the integers in *list*, if *list* is a list of positive integers, and automatically invoke `ACEStart`. `ACEDeleteSubgroupGenerators` also returns the new subgroup generator list. Use `ACESubgroupGenerators` (see 6.5.3) to determine the current subgroup generator list.

7 ► `ACECosetCoincidence( i, n )` F
► `ACECosetCoincidence( n )` F

for the *i*th (or default) interactive ACE process started by `ACEStart`, return the representative of coset *n*, where *n* must be a positive integer, and add it to the subgroup generators; i.e., equates this coset with coset 1, the subgroup. `ACERedo` is automatically invoked.

8 ► `ACERandomCoincidences( i, subindex )` F
► `ACERandomCoincidences( subindex )` F
► `ACERandomCoincidences( i, [subindex] )` F
► `ACERandomCoincidences( [subindex] )` F
► `ACERandomCoincidences( i, [subindex, attempts] )` F
► `ACERandomCoincidences( [subindex, attempts] )` F

for the *i*th (or default) interactive ACE process started by `ACEStart`, attempt up to *attempts* (or, in the first four forms, 8) times to find nontrivial subgroups with index a multiple of *subindex* by repeatedly making random coset numbers coincident with coset 1 and seeing what happens. The starting coset table must be non-empty, but must **not** be complete (use `ACERandomlyApplyCosetCoincidence` (see 6.7.9) if your table is already complete). For each attempt, by applying ACE's `rc` option (see D.2.9) random coset representatives are repeatedly added to the subgroup and the enumeration redone. If the table becomes too small, the attempt is aborted, the original subgroup generators restored, and another attempt made. If an attempt succeeds, then the new set of subgroup generators is retained. `ACERandomCoincidences` returns the list of new subgroup generators added. Use `ACESubgroupGenerators` (see 6.5.3) to determine the current subgroup generator list.

Notes: `ACERandomCoincidences` may add subgroup generators even if it failed to determine a nontrivial subgroup with index a multiple of *subindex*; in such a case, the original status may be restored by applying `ACEDeleteSubgroupGenerators` (see 6.7.6) with the list returned by `ACERandomCoincidences`.

`ACERandomCoincidences` applies the `rc` option (see D.2.9) of ACE which takes the line that if an enumeration has already obtained a finite index then either, *subindex* is already a divisor of that finite index, or the request is impossible. Thus an invocation of `ACERandomCoincidences`, in the case where the coset table is already complete, is an error.

Guru Notes: A coset can have many different representatives. Consider running `ACEStandardCosetNumbering` (see 6.7.2) before `ACERandomCoincidences`, to canonicise the table and the representatives.

9 ► `ACERandomlyApplyCosetCoincidence(` *i* `[:` *controlOptions*`])` F
► `ACERandomlyApplyCosetCoincidence( [:` *controlOptions*`])` F

for the *i*th (or default) interactive ACE process started by `ACEStart`, try to find a larger proper subgroup (i.e. a subgroup of smaller but nontrivial index), by repeatedly applying `ACECosetCoincidence` (see 6.7.7) and seeing what happens; `ACERandomlyApplyCosetCoincidence` returns the (possibly empty) list of new subgroup generators added. The starting coset table must already be complete (use `ACERandomCoincidences` (see 6.7.8) if your table is not already complete). `ACERandomlyApplyCosetCoincidence` provides the following four options (*controlOptions*).

`subindex :=` *subindex*
: Sets the restriction that the final index should be a multiple of *subindex*; *subindex* must be a positive integer divisor of the initial subgroup index.

`hibound :=` *hibound*
: Sets the restriction that the final index should be (strictly) less than *hibound*; *hibound* must be an integer that is greater than 1 and at most the initial subgroup index.

`lobound :=` *lobound*
: Sets the restriction that the final index should be (strictly) greater than *lobound*; *lobound* must be an integer that is at least 1 and (strictly) less than the initial subgroup index.

`attempts :=` *attempts*
: Sets the restriction that the number of applications of `ACECosetCoincidence` should be at most *attempts*; *attempts* must be a positive integer.

By default, *subindex* `= 1`, *hibound* is the existing subgroup index, *lobound* `= 1` and *attempts* `= 8`. If after an attempt the new index is a multiple of *subindex*, less than *hibound* and greater than *lobound* then the process terminates (and the list of new subgroup generators is returned). Otherwise, if an attempt reaches a stage where the criteria cannot be satisfied, the attempt is aborted, the original subgroup generators restored, and another attempt made. If no attempt is successful an empty list is returned. Use `ACESubgroupGenerators` (see 6.5.3) to determine the current subgroup generator list.

10 ► `ACEConjugatesForSubgroupNormalClosure(` *i* `)` F
► `ACEConjugatesForSubgroupNormalClosure()` F
► `ACEConjugatesForSubgroupNormalClosure(` *i* `: add )` F
► `ACEConjugatesForSubgroupNormalClosure(: add )` F

for the *i*th (or default) interactive ACE process started by `ACEStart`, test each conjugate of a subgroup generator by a group generator for membership in the subgroup, and return the (possibly empty) list of conjugates that were determined to not belong to the subgroup (coset 1); and, if called with the add option, these conjugates are also added to the existing list of subgroup generators.

Notes: A conjugate of a subgroup generator is tested for membership of the subgroup, by checking whether it can be traced from coset 1 to coset 1 (see ACETraceWord: 6.5.17). For an **incomplete** coset table, such a trace may not complete, in which case ACEConjugatesForSubgroupNormalClosure may return an empty list even though the subgroup is **not** normally closed within the group.

The add option does **not** guarantee that the resultant subgroup is normally closed. It is still possible that some conjugates of the newly added subgroup generators will not be elements of the subgroup.

Example: To demonstrate the usage and features of ACEConjugatesForSubgroupNormalClosure, let us consider an example where we know pretty well what to expect.

Let G be the group, isomorphic to the symmetric group S_6, with the presentation

$$\{a,b \mid a^2, b^6, (ab^{-1}ab)^3, (ab^{-1}ab^2)^4, (ab)^5, (ab^{-2}ab^2)^2\}$$

(from [CM72]), and let H be the subgroup $\langle ab^3 \rangle$. There is an isomorphism ϕ from G to S_6 mapping a onto $(1,2)$ and ab^3 onto $(1,2,3,4,5,6)$. It follows that ϕ maps ab^3 into A_6. So we know that the normal closure of H has index 2 in G. Let us observe this via ACE.

First we start an enumeration with max set to 80.

```
gap> F := FreeGroup( "a", "b" );;
gap> a := F.1;; b := F.2;;
gap> fgens := GeneratorsOfGroup( F );;
gap> rels := [ a^2, b^6, (a*b^-1*a*b)^3, (a*b^-1*a*b^2)^4, (a*b)^5,
>              (a*b^-2*a*b^2)^2 ];;
gap> sgens := [ a*b^3 ];;
gap> i := ACEStart( fgens, rels, sgens : max := 80 );;
gap> IsCompleteACECosetTable( i );
false
gap> ACEConjugatesForSubgroupNormalClosure( i : add );
#I  ACEConjugatesForSubgroupNormalClosure: All (traceable) conjugates in subgp
[  ]
```

Though we know that H is not equal to its normal closure, we did not get any new elements (we had warned above of such a possibility). Apparently our incomplete table is too small. So let us increase max to 100 and continue.

```
gap> ACEContinue( i : max := 100 );;
gap> IsCompleteACECosetTable( i );
false
gap> ACEConjugatesForSubgroupNormalClosure( i : add );
[ b^-1*a*b^4 ]
gap> IsCompleteACECosetTable( i );
true
gap> ACEStats( i ).index;
20
```

This time we got a new element, and after adding it to the subgroup generators we obtained a complete table. However the resulting subgroup need not yet be the normal closure of H (and in fact we know that it is not). So we continue with another call to the function ACEConjugatesForSubgroupNormalClosure.

```
gap> ACEConjugatesForSubgroupNormalClosure( i : add );
[ b^-2*a*b^5, b*a*b^2 ]
gap> ACEStats( i ).index;
2
```

Now we have the index that we expected. Another call to the function `ACEConjugatesForSubgroupNormalClosure` should not yield any more conjugates. We ensure that this is indeed the case and then display the resulting list of subgroup generators.

```
gap> ACEConjugatesForSubgroupNormalClosure( i : add );
#I  ACEConjugatesForSubgroupNormalClosure: All (traceable) conjugates in subgp
[  ]
gap> ACESubgroupGenerators( i );
[ a*b^3, b*a*b^2, b^-1*a*b^4, b^-2*a*b^5 ]
```

6.8 Primitive ACE Read/Write Functions

For those familiar with the workings of the ACE standalone we provide primitive read/write tools to communicate directly with an interactive ACE process, started via `ACEStart` (possibly with argument 0, but this is not essential). For the most part, it is up to the user to translate the output strings from ACE into a form useful in GAP. However, after the group generators, relators, and subgroup generators have been set in the ACE process, via `ACEWrite`, invocations of any of `ACEGroupGenerators` (see 6.5.1), `ACERelators` (see 6.5.2), `ACESubgroupGenerators` (see 6.5.3), or `ACEParameters` (see 6.5.10) will establish the corresponding GAP values. Be warned though, that unless one of the modes `ACEStart` (without a zero argument; see 6.1.1), `ACERedo` (see 6.2.3) or `ACEContinue` (see 6.2.2), or their equivalent for the standalone ACE (`start;`, `redo;`, or `continue;`), has been invoked since the last change of any parameter options (see 4.12.1), some of the values reported by `ACEParameters` may well be **incorrect**.

1 ► `ACEWrite(` *i*, *string* `)` F
► `ACEWrite(` *string* `)` F

write *string* to the *i*th or default interactive ACE process; *string* must be in exactly the form the ACE standalone expects. The command is echoed via `Info` at `InfoACE` level 4 (with a "`ToACE> `" prompt); i.e. `SetInfoACELevel(4);` will allow you to see what is transmitted to the ACE binary. `ACEWrite` returns `true` if successful in writing to the stream of the interactive ACE process, and `fail` otherwise.

Note: If `ACEWrite` returns `fail` (which means that the ACE process has died), you may like to try resurrecting the interactive ACE process via `ACEResurrectProcess` (see 6.3.4).

2 ► `ACERead(` *i* `)` F
► `ACERead()` F

read a complete line of ACE output, from the *i*th or default interactive ACE process, if there is output to be read and returns `fail` otherwise. When successful, the line is returned as a string complete with trailing newline character. Please note that it is possible to be "too quick" (i.e. the return can be `fail` purely because the output from ACE is not there yet), but if `ACERead` finds any output at all, it waits for a complete line.

3 ► `ACEReadAll(` *i* `)` F
► `ACEReadAll()` F

read and return as many **complete** lines of ACE output, from the *i*th or default interactive ACE process, as there are to be read, **at the time of the call**, as a list of strings with the trailing newlines removed and returns the empty list otherwise. `ACEReadAll` also writes each line read via `Info` at `InfoACE` level 3. Whenever `ACEReadAll` finds only a partial line, it waits for the complete line, thus increasing the probability that it has captured all the output to be had from ACE.

4 ► `ACEReadUntil(` *i*`,` *IsMyLine* `)` F
► `ACEReadUntil(` *IsMyLine* `)` F
► `ACEReadUntil(` *i*`,` *IsMyLine*`,` *Modify* `)` F
► `ACEReadUntil(` *IsMyLine*`,` *Modify* `)` F

read complete lines of ACE output, from the *i*th or default interactive ACE process, "chomps" them (i.e. removes any trailing newline character), emits them to `Info` at `InfoACE` level 3, and applies the function *Modify* (where *Modify* is just the identity map/function for the first two forms) until a "chomped" line *line* for which *IsMyLine*`(` *Modify*`(`*line*`)` `)` is true. `ACEReadUntil` returns the list of *Modify*-ed "chomped" lines read.

Notes: When provided by the user, *Modify* should be a function that accepts a single string argument.

IsMyLine should be a function that is able to accept the output of *Modify* (or take a single string argument when *Modify* is not provided) and should return a boolean.

If *IsMyLine*`(` *Modify*`(`*line*`)` `)` is never true, `ACEReadUntil` will wait indefinitely.

A The Meanings of ACE's output messages

In this chapter, we discuss the meanings of the messages that appear in output from the ACE binary, the verbosity of which is determined by the `messages` option (see 4.18.1). Actually our aim here is to concentrate on describing those "messages" that are controlled by the `messages` option, namely the **progress** and **results messages**; other output from ACE is fairly self-explanatory. (We describe some other output to give points of reference.)

Both interactively and non-interactively, ACE output is `Info`-ed at `InfoACE` level 3. To see the `Info`-ed ACE output, set the `InfoLevel` of `InfoACE` to at least 3, e.g.

```
gap> SetInfoACELevel(3);
```

This causes each line of ACE output to be prepended with "`#I `". Below, we describe the `Info`-ed output observed when each of `ACECosetTableFromGensAndRels`, `ACECosetTable`, `ACEStats` or `ACEStart` is called with three arguments, and presume that users will be able to extend the description to explain output observed from other ACE interface functions. The first-observed (`Info`-ed) output from ACE, is ACE's banner, e.g.

```
#I  ACE 3.001        Sat Feb 27 11:44:12 2016
#I  =========================================
#I  Host information:
#I    name = banksia
#I  ***
```

If there were any errors in the directives put to ACE, or output from the options described in Appendix D, they will appear next. Then, the next observed output is a row of three asterisks:

```
#I  ***
```

Guru note: The three asterisks are generated by a "`text:***`" (see D.7.1) directive, emitted to ACE, so that ACE's response can be used as a sentinel to flush out any user errors, or any output from a user's use of Appendix D options.

Next, if the `messages` option (see 4.18.1) is set to a non-zero value, what is observed is a heading like:

```
#I    #--- ACE 3.001: Run Parameters ---
```

(where `3.001` may be replaced be some later version number of the ACE binary) followed by the "input parameters" developed from the arguments and options passed to `ACECosetTableFromGensAndRels`, `ACEStats` or `ACEStart`. After these appears a separator:

```
#I    #---------------------------------
```

followed by any **progress messages** (progress messages only occur if `messages` is non-zero; recall that by default `messages` = 0), followed by a **results message**.

In the case of `ACECosetTableFromGensAndRels`, the **results message** is followed by a compaction (`CO` or `co`) progress message and a coset table.

ACE's exit banner which should look something like:

```
=========================================
ACE 3.001        Sat Feb 27 11:44:17 2016
```

is only seen when running ACE as a standalone.

Both **progress messages** and the **results message** consist of an initial tag followed by a list of statistics. All messages have values for the statistics `a`, `r`, `h`, `n`, `h`, `l` and `c` (excepting that the second `h`, the one following the `n` statistic, is only given if hole monitoring has been turned on by setting `messages` < 0, which as noted above is expensive and should be avoided unless really needed). Additionally, there may appear the statistics: `m` and `t` (as for the results message); `d`; or `s`, `d` and `c` (as for the `DS` progress message). The meanings of the various statistics and tags will follow later. The following is a sample progress message:

```
#I  AD: a=2 r=1 h=1 n=3; h=66.67% l=1 c=+0.00; m=2 t=2
```

with tag `AD` and values for the statistics `a`, `r`, `h`, `n`, `h` (appears because `messages` < 0), `l`, `c`, `m` and `t`. The following is a sample results message:

```
#I  INDEX = 12 (a=12 r=16 h=1 n=16; l=3 c=0.01; m=14 t=15)
```

which, in this case, declares a successful enumeration of the coset numbers of a subgroup of index 12 within a group, and, as it turns out, values for the same statistics as the sample progress message.

You should see all the above (and a little more), except that your dates and host information will no doubt differ, by running:

```
gap> ACEExample("A5-C5" : echo:=0, messages:=-1);
```

In the following table we list the statistics that can follow a progress or results message tag, in order:

```
--------------------------------------------------------------------
statistic   meaning
--------------------------------------------------------------------
a           number of active coset numbers
r           number of applied coset numbers
h           first (potentially) incomplete row
n           next coset number definition
h           percentage of holes in the table (if ‘messages’$ \< 0$)
l           number of main loop passes
c           total CPU time
m           maximum number of active coset numbers
t           total number of coset numbers defined
s           new deduction stack size (with DS tag)
d           current deduction stack size, or
              no. of non-redundant deductions retained (with DS tag)
c           no. of redundant deductions discarded (with DS tag)
--------------------------------------------------------------------
```

Now that we have discussed the various meanings of the statistics, it's time to discuss the various types of progress and results messages possible. First we describe progress messages.

A.1 Progress Messages

A progress message (and its tag) indicates the function just completed by the enumerator. In the following table, the possible message `tags` appear in the first column. In the `action` column, a `y` indicates the function is aggregated and counted. Every time this count reaches the value of `messages`, a message line is printed and the count is zeroed. Those tags flagged with a `y*` are only present if the appropriate option was included when the ACE binary was compiled (a default compilation includes the appropriate options; so normally read `y*` as `y`). Tags with an `n` in the `action` column indicate the function is not counted, and cause a message line to be output every time they occur. They also cause the action count to be reset.

tag	action	meaning
AD	y	coset 1 application definition (‘SG’/‘RS’ phase)
RD	y	R-style definition
RF	y	row-filling definition
CG	y	immediate gap-filling definition
CC	y*	coincidence processed
DD	y*	deduction processed
CP	y	preferred list gap-filling definition
CD	y	C-style definition
Lx	n	lookahead performed (type ‘x’)
CO	n	table compacted
co	n	compaction routine called, but nothing recovered
CL	n	complete lookahead (table as deduction stack)
UH	n	updated completed-row counter
RA	n	remaining coset numbers applied to relators
SG	n	subgroup generator phase
RS	n	relators in subgroup phase
DS	n	stack overflowed (compacted and doubled)

A.2 Results Messages

The possible results are given in the following table; any result not listed represents an internal error and should be reported to the ACE authors.

result tag	meaning
INDEX = x	finite index of ‘x’ obtained
OVERFLOW	out of table space
SG PHASE OVERFLOW	out of space (processing subgroup generators)
ITERATION LIMIT	‘loop’ limit triggered
TIME LIMT	‘ti’ limit triggered
HOLE LIMIT	‘ho’ limit triggered
INCOMPLETE TABLE	all coset numbers applied, but table has holes
MEMORY PROBLEM	out of memory (building data structures)

Notes: Recall that hole monitoring is switched on by setting a negative value for the `messages` (see 4.18.1) option, but note that hole monitoring is expensive, so don’t turn it on unless you really need it. If you wish to print out the presentation and the options, but not the progress messages, then set `messages` non-zero, but very large. (You’ll still get the `SG`, `DS`, etc. messages, but not the `RD`, `DD`, etc. ones.) You can set `messages` to 1, to monitor all enumerator actions, but be warned that this can yield very large output files.

B Examples

In this chapter we collect together a number of examples which illustrate the various ways in which the ACE Package may be used, and give some interactions with the `ACEExample` function. In a number of cases, we have set the `InfoLevel` of `InfoACE` to 3, so that all output from ACE is displayed, prepended by "`#I `". Recall that to also see the commands directed **to** ACE (behind a "`ToACE> `" prompt), you will need to set the `InfoACE` level to 4. We have omitted the line

```
gap> LoadPackage("ace");
true
```

which is, of course, required at the beginning of any session requiring ACE.

B.1 Example where ACE is made the Standard Coset Enumerator

If ACE is made the standard coset enumerator, one simply uses the method of passing arguments normally used with those commands that invoke `CosetTableFromGensAndRels`, but one is able to use all options available via the ACE interface. As an example we use ACE to compute the permutation representation of a perfect group from the data library (GAP's perfect group library stores for each group a presentation together with generators of a subgroup as words in the group generators such that the permutation representation on the cosets of this subgroup will be a (nice) faithful permutation representation for the perfect group). The example we have chosen is an extension of a group of order 16 by the simple alternating group A_5.

```
gap> TCENUM:=ACETCENUM;; # Make ACE the standard coset enumerator
gap> G := PerfectGroup(IsPermGroup, 16*60, 1   # Arguments ... as per usual
>                      : max := 50, mess := 10 # ... but we use ACE options
>                      );
A5 2^4
gap> GeneratorsOfGroup(G); # Just to show we indeed have a perm'n rep'n
[ (2,4)(3,5)(7,15)(8,14)(10,13)(12,16), (2,6,7)(3,11,12)(4,14,5)(8,9,13)(10,
    15,16), (1,2)(3,8)(4,9)(5,10)(6,7)(11,15)(12,14)(13,16),
  (1,3)(2,8)(4,13)(5,6)(7,10)(9,16)(11,12)(14,15),
  (1,4)(2,9)(3,13)(5,14)(6,15)(7,11)(8,16)(10,12),
  (1,5)(2,10)(3,6)(4,14)(7,8)(9,12)(11,16)(13,15) ]
gap> Order(G);
960
```

The call to `PerfectGroup` produced an output string that identifies the group `G`, but we didn't see how ACE became involved here. Let's redo that part of the above example after first setting the `InfoLevel` of `InfoACE` to 3, so that we may get to glimpse what's going on behind the scenes.

```
gap> SetInfoACELevel(3); # Just to see what's going on behind the scenes
gap> # Recall that we did: TCENUM:=ACETCENUM;;
gap> G := PerfectGroup(IsPermGroup, 16*60, 1   # Arguments ... as per usual
>                      : max := 50, mess := 10 # ... but we use ACE options
>                      );
#I  ACE 3.001        Sun Sep 30 22:08:11 2001
#I  =========================================
#I  Host information:
#I    name = rigel
#I  ***
#I    #--- ACE 3.001: Run Parameters ---
#I  Group Name: G;
#I  Group Generators: abstuv;
#I  Group Relators: (a)^2, (s)^2, (t)^2, (u)^2, (v)^2, (b)^3, (st)^2, (uv)^2,
#I    (su)^2, (sv)^2, (tu)^2, (tv)^2, asau, atav, auas, avat, Bvbu, Bsbvt,
#I    Bubvu, Btbvuts, (ab)^5;
#I  Subgroup Name: H;
#I  Subgroup Generators: a, b;
#I  Wo:1000000; Max:50; Mess:10; Ti:-1; Ho:-1; Loop:0;
#I  As:0; Path:0; Row:1; Mend:0; No:21; Look:0; Com:10;
#I  C:0; R:0; Fi:11; PMod:3; PSiz:256; DMod:4; DSiz:1000;
#I    #--------------------------------
#I  SG: a=1 r=1 h=1 n=2; l=1 c=+0.00; m=1 t=1
#I  RD: a=11 r=1 h=1 n=12; l=2 c=+0.00; m=11 t=11
#I  RD: a=21 r=2 h=1 n=22; l=2 c=+0.00; m=21 t=21
#I  CC: a=29 r=4 h=1 n=31; l=2 c=+0.00; d=0
#I  CC: a=19 r=4 h=1 n=31; l=2 c=+0.00; d=0
#I  CC: a=19 r=6 h=1 n=36; l=2 c=+0.00; d=0
#I  INDEX = 16 (a=16 r=36 h=1 n=36; l=3 c=0.00; m=30 t=35)
#I  CO: a=16 r=17 h=1 n=17; c=+0.00
#I   coset |      b      B      a      s      t      u      v
#I  -------+-------------------------------------------------
#I       1 |      1      1      1      2      3      4      5
#I       2 |     11     14      4      1      6      8      9
#I       3 |     13     15      5      6      1     10     11
#I       4 |      7      5      2      8     10      1      7
#I       5 |      4      7      3      9     11      7      1
#I       6 |      8     10      7      3      2     12     14
#I       7 |      5      4      6     15     16      5      4
#I       8 |     10      6      8      4     12      2     15
#I       9 |     16     12     10      5     14     15      2
#I      10 |      6      8      9     12      4      3     16
#I      11 |     14      2     11     14      5     16      3
#I      12 |      9     16     15     10      8      6     13
#I      13 |     15      3     13     16     15     14     12
#I      14 |      2     11     16     11      9     13      6
#I      15 |      3     13     12      7     13      9      8
#I      16 |     12      9     14     13      7     11     10
A5 2^4
```

B.2 Example of Using ACECosetTableFromGensAndRels

The following example calls ACE for up to 800 coset numbers (`max := 800`) using Mendelsohn style relator processing (`mendelsohn`) and sets progress messages to be printed every 500 iterations (`messages :=500`); we do "`SetInfoACELevel(3);`" so that we may see these messages. The value of `table`, i.e. the GAP coset table, immediately follows the last ACE message ("`#I `") line, but both the coset table from ACE and the GAP coset table have been abbreviated. A slightly modified version of this example, which includes the `echo` option is available on-line via `table := ACEExample("perf602p5");`. You may wish to peruse the notes in the `ACEExample` index first, however, by executing `ACEExample();`. (Note that the final table output here is `lenlex` standardised.)

```
gap> SetInfoACELevel(3);           # So we can see the progress messages
gap> G := PerfectGroup(IsSubgroupFpGroup,2^5*60, 2);;# See previous example:
gap>                               # "Example where ACE is made the
gap>                               #  Standard Coset Enumerator"
gap> fgens := FreeGeneratorsOfFpGroup(G);;
gap> table := ACECosetTableFromGensAndRels(
>                # arguments
>                fgens, RelatorsOfFpGroup(G), fgens{[1]}
>                # options
>                : mendelsohn, max:=800, mess:=500);
#I  ACE 3.001       Sun Sep 30 22:10:10 2001
#I  =========================================
#I  Host information:
#I    name = rigel
#I  ***
#I    #--- ACE 3.001: Run Parameters ---
#I  Group Name: G;
#I  Group Generators: abstuvd;
#I  Group Relators: (s)^2, (t)^2, (u)^2, (v)^2, (d)^2, aad, (b)^3, (st)^2,
#I    (uv)^2, (su)^2, (sv)^2, (tu)^2, (tv)^2, Asau, Atav, Auas, Avat, Bvbu,
#I    dAda, dBdb, (ds)^2, (dt)^2, (du)^2, (dv)^2, Bubvu, Bsbdvt, Btbvuts,
#I    (ab)^5;
#I  Subgroup Name: H;
#I  Subgroup Generators: a;
#I  Wo:1000000; Max:800; Mess:500; Ti:-1; Ho:-1; Loop:0;
#I  As:0; Path:0; Row:1; Mend:1; No:28; Look:0; Com:10;
#I  C:0; R:0; Fi:13; PMod:3; PSiz:256; DMod:4; DSiz:1000;
#I    #--------------------------------
#I  SG: a=1 r=1 h=1 n=2; l=1 c=+0.00; m=1 t=1
#I  RD: a=321 r=68 h=1 n=412; l=5 c=+0.00; m=327 t=411
#I  CC: a=435 r=162 h=1 n=719; l=9 c=+0.00; d=0
#I  CL: a=428 r=227 h=1 n=801; l=13 c=+0.00; m=473 t=800
#I  DD: a=428 r=227 h=1 n=801; l=14 c=+0.00; d=33
#I  CO: a=428 r=192 h=243 n=429; l=15 c=+0.00; m=473 t=800
#I  INDEX = 480 (a=480 r=210 h=484 n=484; l=18 c=0.00; m=480 t=855)
#I  CO: a=480 r=210 h=481 n=481; c=+0.00
#I   coset |      a      A      b      B      s      t      u      v      d
#I  -------+---------------------------------------------------------------
#I       1 |      1      1      7      6      2      3      4      5      1
#I       2 |      4      4     22     36      1      8     10     11      2
... 476 lines omitted here ...
#I     479 |    479    479    384    383    475    468    470    471    479
```

```
#I     480 |    480    480    421    420    470    469    475    476    480
[ [ 1, 8, 13, 6, 7, 4, 5, 2, 34, 35, 32, 33, 3, 48, 49, 46, 47, 57, 59, 28,
      21, 25, 62, 64, 22, 26, 66, 20, 67, 69, 74, 11, 12, 9, 10, 89, 65, 87,
... 30 lines omitted here ...
      477, 438, 478, 446, 475, 479, 471, 473, 476, 469 ],
  [ 1, 8, 13, 6, 7, 4, 5, 2, 34, 35, 32, 33, 3, 48, 49, 46, 47, 57, 59, 28,
      21, 25, 62, 64, 22, 26, 66, 20, 67, 69, 74, 11, 12, 9, 10, 89, 65, 87,
... 30 lines omitted here ...
      477, 438, 478, 446, 475, 479, 471, 473, 476, 469 ],
... 363 lines omitted here ...
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
      21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
... 30 lines omitted here ...
      472, 473, 474, 475, 476, 477, 478, 479, 480 ] ]
```

B.3 Example of Using ACE Interactively (Using ACEStart)

Now we illustrate a simple interactive process, with an enumeration of an index 12 subgroup (isomorphic to C_5) within A_5. Observe that we have relied on the default level of messaging from ACE (`messages` = 0) which gives a result line (the "`#I INDEX`" line here) only, without parameter information. The result line is visible in the `Info`-ed component of the output below because we set the `InfoLevel` of `InfoACE` to a value of at least 2 (in fact we set it to 3; doing "`SetInfoACELevel(2);`" would make **only** the result line visible). We have however used the option `echo`, so that we can see how the interface handled the arguments and options. On-line try: `SetInfoACELevel(3); ACEExample("A5-C5", ACEStart);` (this is nearly equivalent to the sequence following, but the variables `F`, `a`, `b`, `G` are not accessible, being "local" to `ACEExample`).

```
gap> SetInfoACELevel(3); # So we can see output from ACE binary
gap> F := FreeGroup("a","b");; a := F.1;;  b := F.2;;
gap> G := F / [a^2, b^3, (a*b)^5 ];
<fp group on the generators [ a, b ]>
gap> ACEStart(FreeGeneratorsOfFpGroup(G), RelatorsOfFpGroup(G), [a*b]
>           # Options
>           : echo, # Echo handled by GAP (not ACE)
>             enum := "A_5",  # Give the group G a meaningful name
>             subg := "C_5"); # Give the subgroup a meaningful name
ACEStart called with the following arguments:
 Group generators : [ a, b ]
 Group relators : [ a^2, b^3, a*b*a*b*a*b*a*b*a*b ]
 Subgroup generators : [ a*b ]
#I  ACE 3.001        Sun Sep 30 22:11:42 2001
#I  =========================================
#I  Host information:
#I    name = rigel
ACEStart called with the following options:
 echo := true (not passed to ACE)
 enum := A_5
 subg := C_5
#I  ***
#I  INDEX = 12 (a=12 r=16 h=1 n=16; l=3 c=0.00; m=14 t=15)
1
```

The return value on the last line is an "index" that identifies the interactive process; we use this "index" with functions that need to interact with the interactive ACE process; we now demonstrate this with the interactive version of ACEStats:

```
gap> ACEStats(1);
rec( index := 12, cputime := 0, cputimeUnits := "10^-2 seconds",
  activecosets := 12, maxcosets := 14, totcosets := 15 )
gap> # Actually, we didn't need to pass an argument to ACEStats()
gap> # ... we could have relied on the default:
gap> ACEStats();
rec( index := 12, cputime := 0, cputimeUnits := "10^-2 seconds",
  activecosets := 12, maxcosets := 14, totcosets := 15 )
```

Similarly, we may use ACECosetTable with 0 or 1 arguments, which is the interactive version of ACECosetTableFromGensAndRels, and which returns a standard table.

```
gap> ACECosetTable(); # Interactive version of ACECosetTableFromGensAndRels()
#I  CO: a=12 r=13 h=1 n=13; c=+0.00
#I   coset |      b      B      a
#I  -------+---------------------
#I       1 |      3      2      2
#I       2 |      1      3      1
#I       3 |      2      1      4
#I       4 |      8      5      3
#I       5 |      4      8      6
#I       6 |      9      7      5
#I       7 |      6      9      8
#I       8 |      5      4      7
#I       9 |      7      6     10
#I      10 |     12     11      9
#I      11 |     10     12     12
#I      12 |     11     10     11
[ [ 2, 1, 4, 3, 7, 8, 5, 6, 10, 9, 12, 11 ],
  [ 2, 1, 4, 3, 7, 8, 5, 6, 10, 9, 12, 11 ],
  [ 3, 1, 2, 5, 6, 4, 8, 9, 7, 11, 12, 10 ],
  [ 2, 3, 1, 6, 4, 5, 9, 7, 8, 12, 10, 11 ] ]
gap> # To terminate the interactive process we do:
gap> ACEQuit(1); # Again, we could have omitted the 1
gap> # If we had more than one interactive process we could have
gap> # terminated them all in one go with:
gap> ACEQuitAll();
```

B.4 Fun with ACEExample

First let's see the ACEExample index (obtained with no argument, with "index" as argument, or with a non-existent example as argument):

```
gap> ACEExample();
#I                   ACEExample Index (Table of Contents)
#I                   ------------------------------------
#I  This table of possible examples is displayed when calling ACEExample
#I  with no arguments, or with the argument: "index" (meant in the sense
#I  of 'list'), or with a non-existent example name.
```

```
#I
#I  The following ACE examples are available (in each case, for a subgroup
#I  H of a group G, the cosets of H in G are enumerated):
#I
#I    Example          G                       H               strategy
#I    -------          -                       -               --------
#I    "A5"             A_5                     Id              default
#I    "A5-C5"          A_5                     C_5             default
#I    "C5-fel0"        C_5                     Id              felsch := 0
#I    "F27-purec"      F(2,7) = C_29           Id              purec
#I    "F27-fel0"       F(2,7) = C_29           Id              felsch := 0
#I    "F27-fel1"       F(2,7) = C_29           Id              felsch := 1
#I    "M12-hlt"        M_12 (Matthieu group)   Id              hlt
#I    "M12-fel1"       M_12 (Matthieu group)   Id              felsch := 1
#I    "SL219-hard"     SL(2,19)                |G : H| = 180   hard
#I    "perf602p5"      PerfectGroup(60*2^5,2) |G : H| = 480   default
#I  * "2p17-fel1"      |G| = 2^17              Id              felsch := 1
#I    "2p17-fel1a"     |G| = 2^17              |G : H| = 1     felsch := 1
#I    "2p17-2p3-fel1"  |G| = 2^17              |G : H| = 2^3   felsch := 1
#I    "2p17-2p14-fel1" |G| = 2^17              |G : H| = 2^14 felsch := 1
#I    "2p17-id-fel1"   |G| = 2^17              Id              felsch := 1
#I  * "2p18-fel1"      |G| = 2^18              |G : H| = 2     felsch := 1
#I  * "big-fel1"       |G| = 2^18.3            |G : H| = 6     felsch := 1
#I  * "big-hard"       |G| = 2^18.3            |G : H| = 6     hard
#I
#I  Notes
#I  -----
#I  1. The example (first) argument of ACEExample() is a string; each
#I     example above is in double quotes to remind you to include them.
#I  2. By default, ACEExample applies ACEStats to the chosen example. You
#I     may alter the ACE function used, by calling ACEExample with a 2nd
#I     argument; choose from: ACECosetTableFromGensAndRels (or, equival-
#I     ently ACECosetTable), or ACEStart, e.g. `ACEExample("A5", ACEStart);'
#I  3. You may call ACEExample with additional ACE options (entered after a
#I     colon in the usual way for options), e.g. `ACEExample("A5" : hlt);'
#I  4. Try the *-ed examples to explore how to modify options when an
#I     enumeration fails (just follow the instructions you get within the
#I     break-loop, or see Notes 2. and 3.).
#I  5. Try `SetInfoACELevel(3);' before calling ACEExample, to see the
#I     effect of setting the "mess" (= "messages") option.
#I  6. To suppress a long output, use a double semicolon (`;;') after the
#I     ACEExample command. (However, this does not suppress Info-ed output.)
#I  7. Also, try `SetInfoACELevel(2);' or `SetInfoACELevel(4);' before
#I     calling ACEExample.
gap>
```

Notice that the example we first met in Section 1.2, the Fibonacci group F(2,7), is available via examples `"F27-purec"`, `"F27-fel0"`, and `"F27-fel1"` (with 2nd argument `ACECosetTableFromGensAndRels` to produce a coset table), except that each of these enumerate the cosets of its trivial subgroup (of index 29). Let's experiment with the first of these F(2,7) examples; since this example uses the `messages` option, we ought to set the `InfoLevel` of `InfoACE` to 3, first, but since the coset table is quite long, we will be content for the moment with applying the default function `ACEStats` to the example.

Before exhibiting the example we list a few observations that should be made. Observe that the first group of `Info` lines list the commands that are executed; these lines are followed by the result of the `echo` option (see 4.11.12); which in turn are followed by `Info` messages from ACE courtesy of the non-zero value of the `messages` option (and we see these because we first set the `InfoLevel` of `InfoACE` to 3); and finally, we get the output (record) of the `ACEStats` command.

Observe also that ACE uses the same generators as are input; this will always occur if you stick to single lowercase letters for your generator names. Note, also that capitalisation is used by ACE as a short-hand for inverses, e.g. `C = c^-1` (see `Group Relators` in the ACE "Run Parameters" block).

```
gap> SetInfoACELevel(3);
gap> ACEExample("F27-purec");
#I  # ACEExample "F27-purec" : enumeration of cosets of H in G,
#I  # where G = F(2,7) = C_29, H = Id, using purec strategy.
#I  #
#I  # F, G, a, b, c, d, e, x, y are local to ACEExample
#I  # We define F(2,7) on 7 generators
#I  F := FreeGroup("a","b","c","d","e", "x", "y");
#I       a := F.1;  b := F.2;  c := F.3;  d := F.4;
#I       e := F.5;  x := F.6;  y := F.7;
#I  G := F / [a*b*c^-1, b*c*d^-1, c*d*e^-1, d*e*x^-1,
#I            e*x*y^-1, x*y*a^-1, y*a*b^-1];
#I  ACEStats(
#I      FreeGeneratorsOfFpGroup(G),
#I      RelatorsOfFpGroup(G),
#I      [] # Generators of identity subgroup (empty list)
#I      # Options that don't affect the enumeration
#I      : echo, enum := "F(2,7), aka C_29", subg := "Id",
#I      # Other options
#I      wo := "2M", mess := 25000, purec);
ACEStats called with the following arguments:
 Group generators : [ a, b, c, d, e, x, y ]
 Group relators : [ a*b*c^-1, b*c*d^-1, c*d*e^-1, d*e*x^-1, e*x*y^-1,
  x*y*a^-1, y*a*b^-1 ]
 Subgroup generators : [  ]
#I  ACE 3.001        Sun Sep 30 22:16:08 2001
#I  =========================================
#I  Host information:
#I    name = rigel
ACEStats called with the following options:
 echo := true (not passed to ACE)
 enum := F(2,7), aka C_29
 subg := Id
 wo := 2M
 mess := 25000
 purec (no value, passed to ACE via option: pure c)
#I  ***
#I    #--- ACE 3.001: Run Parameters ---
#I  Group Name: F(2,7), aka C_29;
#I  Group Generators: abcdexy;
#I  Group Relators: abC, bcD, cdE, deX, exY, xyA, yaB;
#I  Subgroup Name: Id;
#I  Subgroup Generators: ;
```

```
#I  Wo:2M; Max:142855; Mess:25000; Ti:-1; Ho:-1; Loop:0;
#I  As:0; Path:0; Row:0; Mend:0; No:0; Look:0; Com:100;
#I  C:1000; R:0; Fi:1; PMod:0; PSiz:256; DMod:4; DSiz:1000;
#I    #--------------------------------
#I  DD: a=5290 r=1 h=1050 n=5291; l=8 c=+0.00; d=2
#I  CD: a=10410 r=1 h=2149 n=10411; l=13 c=+0.01; m=10410 t=10410
#I  DD: a=15428 r=1 h=3267 n=15429; l=18 c=+0.01; d=0
#I  DD: a=20430 r=1 h=4386 n=20431; l=23 c=+0.02; d=1
#I  DD: a=25397 r=1 h=5519 n=25399; l=28 c=+0.01; d=1
#I  CD: a=30313 r=1 h=6648 n=30316; l=33 c=+0.01; m=30313 t=30315
#I  DS: a=32517 r=1 h=7326 n=33240; l=36 c=+0.01; s=2000 d=997 c=4
#I  DS: a=31872 r=1 h=7326 n=33240; l=36 c=+0.00; s=4000 d=1948 c=53
#I  DS: a=29077 r=1 h=7326 n=33240; l=36 c=+0.00; s=8000 d=3460 c=541
#I  DS: a=23433 r=1 h=7326 n=33240; l=36 c=+0.01; s=16000 d=5940 c=2061
#I  DS: a=4163 r=1 h=7326 n=33240; l=36 c=+0.03; s=32000 d=447 c=15554
#I  INDEX = 29 (a=29 r=1 h=33240 n=33240; l=37 c=0.15; m=33237 t=33239)
rec( index := 29, cputime := 15, cputimeUnits := "10^-2 seconds",
  activecosets := 29, maxcosets := 33237, totcosets := 33239 )
```

Now let's see that we can add some new options, even ones that over-ride the example's options; but first we'll reduce the output a bit by setting the `InfoLevel` of `InfoACE` to 2 and since we are not going to observe any progress messages from ACE with that `InfoACE` level we'll set `messages := 0`; also we'll use the function `ACECosetTableFromGensAndRels` and so it's like our first encounter with F(2,7) we'll add the subgroup generator `c` via `sg := ["c"]` (see D.2.4). Observe that `"c"` is a string not a GAP group generator; to convert a list of GAP words *sgens* in generators *fgens*, suitable for an assignment of the `sg` option use the construction: `ToACEWords(`*fgens*`, `*sgens*`)` (see 6.3.6). Note again that if only single lowercase letter strings are used to identify the GAP group generators, the same strings are used to identify those generators in ACE. (It's actually fortunate that we could pass the value of `sg` as a string here, since the generators of each `ACEExample` example are **local** variables and so are not accessible, though we could call `ACEExample` with 2nd argument `ACEStart` and use `ACEGroupGenerators` to get at them.) For good measure, we also change the string identifying the subgroup (since it will no longer be the trivial group), via the `subgroup` option (see 4.19.2).

In considering the example following, observe that in the `Info` block all the original example options are listed along with our new options `sg := [ "c" ], messages := 0` after the tag "`# User Options`". Following the `Info` block there is a block due to `echo`; in its listing of the options first up there is `aceexampleoptions` alerting us that we passed some `ACEExample` options; observe also that in this block `subg := Id` and `mess := 25000` disappear (they are over-ridden by `subgroup := "< c >"`, `messages := 0`, but the quotes for the value of `subgroup` are not visible); note that we don't have to use the same abbreviations for options to over-ride them. Also observe that our new options are **last**.

```
gap> SetInfoACELevel(2);
gap> ACEExample("F27-purec", ACECosetTableFromGensAndRels
>               : sg := ["c"], subgroup := "< c >", messages := 0);
#I  # ACEExample "F27-purec" : enumeration of cosets of H in G,
#I  # where G = F(2,7) = C_29, H = Id, using purec strategy.
#I  #
#I  # F, G, a, b, c, d, e, x, y are local to ACEExample
#I  # We define F(2,7) on 7 generators
#I  F := FreeGroup("a","b","c","d","e", "x", "y");
#I       a := F.1;  b := F.2;  c := F.3;  d := F.4;
#I       e := F.5;  x := F.6;  y := F.7;
#I  G := F / [a*b*c^-1, b*c*d^-1, c*d*e^-1, d*e*x^-1,
#I            e*x*y^-1, x*y*a^-1, y*a*b^-1];
```

```
#I  ACECosetTableFromGensAndRels(
#I      FreeGeneratorsOfFpGroup(G),
#I      RelatorsOfFpGroup(G),
#I      [] # Generators of identity subgroup (empty list)
#I      # Options that don't affect the enumeration
#I      : echo, enum := "F(2,7), aka C_29", subg := "Id",
#I      # Other options
#I      wo := "2M", mess := 25000, purec,
#I      # User Options
#I        sg := [ "c" ],
#I        subgroup := "< c >",
#I        messages := 0);
ACECosetTableFromGensAndRels called with the following arguments:
 Group generators : [ a, b, c, d, e, x, y ]
 Group relators : [ a*b*c^-1, b*c*d^-1, c*d*e^-1, d*e*x^-1, e*x*y^-1,
  x*y*a^-1, y*a*b^-1 ]
 Subgroup generators : [  ]
ACECosetTableFromGensAndRels called with the following options:
 aceexampleoptions := true (inserted by ACEExample, not passed to ACE)
 echo := true (not passed to ACE)
 enum := F(2,7), aka C_29
 wo := 2M
 purec (no value, passed to ACE via option: pure c)
 sg := [ "c" ] (brackets are not passed to ACE)
 subgroup := < c >
 messages := 0
#I  INDEX = 1 (a=1 r=2 h=2 n=2; l=4 c=0.00; m=332 t=332)
[ [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ],
  [ 1 ], [ 1 ], [ 1 ], [ 1 ] ]
```

Now following on from our last example we shall demonstrate how one can recover from a `break`-loop (see Section 1.2). To force the `break`-loop we pass `max := 2` (see 4.17.6), while using the ACE interface function `ACECosetTableFromGensAndRels` with `ACEExample`; in this way, ACE will not be able to complete the enumeration, and hence enters a `break`-loop when it tries to provide a complete coset table. While we're at it we'll pass the `hlt` (see 5.1.5) strategy option (which will over-ride `purec`). (The `InfoACE` level is still 2.) To avoid getting a trace-back during the `break`-loop (which can look a little scary to the unitiated) we will set `OnBreak` (see 6.4.3) as follows:

```
gap> NormalOnBreak := OnBreak;; # Save the old value to restore it later
gap> OnBreak := function() Where(0); end;;
```

Note that there are some "user-input" comments inserted at the `brk>` prompt.

```
gap> ACEExample("F27-purec", ACECosetTableFromGensAndRels
>               : sg := ["c"], subgroup := "< c >", max := 2, hlt);
#I  # ACEExample "F27-purec" : enumeration of cosets of H in G,
#I  # where G = F(2,7) = C_29, H = Id, using purec strategy.
#I  #
#I  # F, G, a, b, c, d, e, x, y are local to ACEExample
#I  # We define F(2,7) on 7 generators
#I  F := FreeGroup("a","b","c","d","e", "x", "y");
#I       a := F.1;  b := F.2;  c := F.3;  d := F.4;
#I       e := F.5;  x := F.6;  y := F.7;
#I  G := F / [a*b*c^-1, b*c*d^-1, c*d*e^-1, d*e*x^-1,
```

```
#I           e*x*y^-1, x*y*a^-1, y*a*b^-1];
#I  ACECosetTableFromGensAndRels(
#I      FreeGeneratorsOfFpGroup(G),
#I      RelatorsOfFpGroup(G),
#I      [] # Generators of identity subgroup (empty list)
#I      # Options that don't affect the enumeration
#I      : echo, enum := "F(2,7), aka C_29", subg := "Id",
#I      # Other options
#I      wo := "2M", mess := 25000, purec,
#I      # User Options
#I        sg := [ "c" ],
#I        subgroup := "< c >",
#I        max := 2,
#I        hlt := true);
ACECosetTableFromGensAndRels called with the following arguments:
 Group generators : [ a, b, c, d, e, x, y ]
 Group relators : [ a*b*c^-1, b*c*d^-1, c*d*e^-1, d*e*x^-1, e*x*y^-1,
  x*y*a^-1, y*a*b^-1 ]
 Subgroup generators : [  ]
ACECosetTableFromGensAndRels called with the following options:
 aceexampleoptions := true (inserted by ACEExample, not passed to ACE)
 echo := true (not passed to ACE)
 enum := F(2,7), aka C_29
 wo := 2M
 mess := 25000
 purec (no value, passed to ACE via option: pure c)
 sg := [ "c" ] (brackets are not passed to ACE)
 subgroup := < c >
 max := 2
 hlt (no value)
#I  OVERFLOW (a=2 r=1 h=1 n=3; l=4 c=0.00; m=2 t=2)
Error, no coset table ...
 the `ACE' coset enumeration failed with the result:
 OVERFLOW (a=2 r=1 h=1 n=3; l=4 c=0.00; m=2 t=2)
Entering break read-eval-print loop ...
 try relaxing any restrictive options
 e.g. try the `hard' strategy or increasing `workspace'
 type: '?strategy options' for info on strategies
 type: '?options for ACE' for info on options
 type: 'DisplayACEOptions();' to see current ACE options;
 type: 'SetACEOptions(:<option1> := <value1>, ...);'
 to set <option1> to <value1> etc.
 (i.e. pass options after the ':' in the usual way)
 ... and then, type: 'return;' to continue.
 Otherwise, type: 'quit;' to quit to outer loop.
brk> # Let's give ACE enough coset numbers to work with ...
brk> # and while we're at it see the effect of 'echo := 2' :
brk> SetACEOptions(: max := 0, echo := 2);
brk> # Let's check what the options are now:
brk> DisplayACEOptions();
rec(
  enum := "F(2,7), aka C_29",
```

```
    wo := "2M",
    mess := 25000,
    purec := true,
    sg := [ "c" ],
    subgroup := "< c >",
    hlt := true,
    max := 0,
    echo := 2 )

brk> # That's ok ... so now we 'return;' to escape the break-loop
brk> return;
ACECosetTableFromGensAndRels called with the following arguments:
 Group generators : [ a, b, c, d, e, x, y ]
 Group relators : [ a*b*c^-1, b*c*d^-1, c*d*e^-1, d*e*x^-1, e*x*y^-1,
  x*y*a^-1, y*a*b^-1 ]
 Subgroup generators : [  ]
ACECosetTableFromGensAndRels called with the following options:
 enum := F(2,7), aka C_29
 wo := 2M
 mess := 25000
 purec (no value, passed to ACE via option: pure c)
 sg := [ "c" ] (brackets are not passed to ACE)
 subgroup := < c >
 hlt (no value)
 max := 0
 echo := 2 (not passed to ACE)
Other options set via ACE defaults:
 asis := 0
 compaction := 10
 ct := 0
 dmode := 0
 dsize := 1000
 fill := 1
 hole := -1
 lookahead := 1
 loop := 0
 mendelsohn := 0
 no := 0
 path := 0
 pmode := 0
 psize := 256
 row := 1
 rt := 1000
 time := -1
#I  INDEX = 1 (a=1 r=2 h=2 n=2; l=3 c=0.00; m=2049 t=3127)
[ [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ],
  [ 1 ], [ 1 ], [ 1 ], [ 1 ] ]
```

Observe that `purec` did **not** disappear from the option list; nevertheless, it **is** over-ridden by the `hlt` option (at the ACE level). Observe the "`Other options set via ACE defaults`" list of options that has resulted from having the `echo := 2` option (see 4.11.12). Observe, also, that `hlt` is nowhere near as good, here, as `purec` (refer to Section 1.2):

whereas `purec` completed the same enumeration with a total number of coset numbers of 332, the `hlt` strategy required 3127.

Before we finish this section, let us say something about the examples listed when one calls `ACEExample` with no arguments that have a `*` beside them; these are examples for which the enumeration fails to complete. The first such example listed is `"2p17-fel1"`, where a group of order 2^{17} is enumerated over the identity subgroup with the `felsch := 1` strategy. The enumeration fails after defining a total number of 416664 coset numbers. (In fact, the enumeration can be made to succeed by simply increasing `workspace` to `"4700k"`, but in doing so a total of 783255 coset numbers are defined.) With the example `"2p17-fel1a"` the same group is again enumerated, again with the `felsch := 1` strategy, but this time over the group itself and after tweaking a few options, to see how well we can do. The other `"2p17-`*XXX*`"` examples are again enumerations of the same group but over smaller and smaller subgroups, until we once again enumerate over the identity subgroup but far more efficiently this time (only needing to define a total of 550659 coset numbers, which can be achieved with `workspace` set to `"3300k"`).

The other `*`-ed examples enumerate overgroups of the group of order 2^{17} of the `"2p17-`*XXX*`"` examples. It's recommended that you try these with second argument `ACECosetTableFromGensAndRels` so that you enter a `break`-loop, where you can experiment with modifying the options using `SetACEOptions`. The example `"2p18-fel1"` can be made to succeed with `hard, mend, workspace := "10M"`; why don't you see if you can do better! There are no hints for the other two `*`-ed examples; they are exercises for you to try.

Let's now restore the original value of `OnBreak`:

```
gap> OnBreak := NormalOnBreak;;
```

Of course, running `ACEExample` with `ACEStart` as second argument opens up far more flexibility. Try it! Have fun! Play with as many options as you can.

B.5 Using ACEReadResearchExample

Without an argument, the function `ACEReadResearchExample` reads the file `"pgrelfind.g"` in the `res-examples` directory which defines GAP functions such as `PGRelFind`, that were used in [CHHR02] to show that the group $L_3(5)$ has deficiency 0.

The **deficiency** of a finite presentation $\{X \mid R\}$ of a finite group G is $|R| - |X|$, and the **deficiency** of the group G is the minimum of the deficiencies of all finite presentations of G.

Let us now invoke `ACEReadResearchExample` with no arguments:

```
gap> ACEReadResearchExample();
#I  The following are now defined:
#I
#I  Functions:
#I    PGRelFind, ClassesGenPairs, TranslatePresentation,
#I    IsACEResExampleOK
#I  Variables:
#I    ACEResExample, ALLRELS, newrels, F, a, b, newF, x, y,
#I    L2_8, L2_16, L3_3s, U3_3s, M11, M12, L2_32,
#I    U3_4s, J1s, L3_5s, PSp4_4s, presentations
#I
#I  Also:
#I
#I  TCENUM = ACETCENUM  (Todd-Coxeter Enumerator is now ACE)
#I
#I  For an example of their application, you might like to try:
#I  gap> ACEReadResearchExample("doL28.g" : optex := [1,2,4,5,8]);
#I  (the output is 65 lines followed by a 'gap> ' prompt)
```

```
#I
#I  For information type: ?Using ACEReadResearchExample
gap>
```

The output (`Info`-ed at `InfoACELevel` 1) states that a number of new functions are defined. During a GAP session, you can find out details of these functions by typing:

```
gap> ?Using ACEReadResearchExample
```

C Finer Points with Examples

The examples in this chapter are intended to provide the nearest GAP equivalent of the similarly named sections in Appendix A of `ace3001.ps` (the standalone manual in directory `standalone-doc`). There is a **lot** of detail here, which the novice ACE Package user won't want to know about. Please, despite the name of the first section of this chapter, read the examples in Appendix B first.

C.1 Getting Started

Each of the functions `ACECosetTableFromGensAndRels` (see 1.2.1), `ACEStats` (see 1.3.1 — non-interactive version) and `ACEStart` (see 6.1.1), may be called with three arguments: *fgens* (the group generators), *rels* (the group relators), and *sgens* (the subgroup generators). While it is legal for the arguments *rels* and *sgens* to be empty lists, it is always an error for *fgens* to be empty, e.g.

```
gap> ACEStats([],[],[]);
Error, fgens (arg[1]) must be a non-empty list of group generators ...
 called from
CALL_ACE( "ACEStats", arg[1], arg[2], arg[3] ) called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
 type: 'quit;' to quit to outer loop, or
 type: 'fgens := <val>; return;' to assign <val> to fgens to continue.
brk> fgens := FreeGeneratorsOfFpGroup(FreeGroup("a"));
[ a ]
brk> return;
rec( index := 0, cputime := 13, cputimeUnits := "10^-2 seconds",
  activecosets := 499998, maxcosets := 499998, totcosets := 499998 )
```

The example shows that the ACE package does allow you to recover from the `break`-loop. However, the definition of `fgens` above is local to the `break`-loop, and in any case we shall want two generators for the examples we wish to consider and raise some other points; so let us re-define `fgens` and start again:

```
gap> F := FreeGroup("a", "b");; fgens := FreeGeneratorsOfFpGroup(F);;
```

An `ACEStats` example

By default, the presentation is not echoed; use the `echo` (see 4.11.12) option if you want that. Also, by default, the ACE binary only prints a **results message**, but we won't see that unless we set `InfoACE` to a level of at least 2 (see 1.9.3):

```
gap> SetInfoACELevel(2);
```

Calling `ACEStats` with arguments `fgens`, `[]`, `[]`, defines a free froup with 2 generators, since the second argument defines an empty relator list; and since the third argument is an empty list of generators, the subgroup defined is trivial. So the enumeration overflows:

```
gap> ACEStats(fgens, [], []);
#I  OVERFLOW (a=249998 r=83333 h=83333 n=249999; l=337 c=0.10; m=249998 t=2499\
98)
rec( index := 0, cputime := 10, cputimeUnits := "10^-2 seconds",
  activecosets := 249998, maxcosets := 249998, totcosets := 249998 )
```

The line starting with "`#I` ". is the `Info`-ed **results message** from ACE; see Appendix A for details on what it means. Observe that since the enumeration overflowed, ACE's result message has been translated into a GAP record with `index` field 0.

To dump out the presentation and parameters associated with an enumeration, ACE provides the `sr` (see D.5.7) option. However, you won't see output of this command, unless you set the `InfoACELevel` to at least 3. Also, to ensure the reliability of the output of the `sr` option, an enumeration should **precede** it; for `ACEStats` (and `ACECosetTableFromGensAndRels`) the directive `start` (see D.3.2) required to initiate an enumeration is inserted (automatically) after all the user's options, except if the user herself supplies an option that initiates an enumeration (namely, one of `start` or `begin` (see D.3.2), `aep` (see D.1.1) or `rep` (see D.1.2)). Interactively, the equivalent of the `sr` command is `ACEParameters` (see 6.5.10), which gives an output record that is immediately GAP-usable. With the above in mind let's rerun the enumeration and get ACE's dump of the presentation and parameters:

```
gap> SetInfoACELevel(3);
gap> ACEStats(fgens, [], [] : start, sr := 1);
#I  ACE 3.001        Wed Oct 31 09:36:39 2001
#I  =========================================
#I  Host information:
#I    name = rigel
#I  OVERFLOW (a=249998 r=83333 h=83333 n=249999; l=337 c=0.09; m=249998 t=2499\
98)
#I    #--- ACE 3.001: Run Parameters ---
#I  Group Name: G;
#I  Group Generators: ab;
#I  Group Relators: ;
#I  Subgroup Name: H;
#I  Subgroup Generators: ;
#I  Wo:1000000; Max:249998; Mess:0; Ti:-1; Ho:-1; Loop:0;
#I  As:0; Path:0; Row:1; Mend:0; No:0; Look:0; Com:10;
#I  C:0; R:0; Fi:7; PMod:3; PSiz:256; DMod:4; DSiz:1000;
#I    #---------------------------------
rec( index := 0, cputime := 9, cputimeUnits := "10^-2 seconds",
  activecosets := 249998, maxcosets := 249998, totcosets := 249998 )
```

Observe that at `InfoACE` level 3, one also gets ACE's banner. We could have printed out the first few lines of the coset table if we had wished, using the `print` (see D.5.8) option, but note as with the `sr` option, an enumeration should **precede** it. Here's what happens if you disregard this (recall, we still have the `InfoACE` level set to 3):

```
gap> ACEStats(fgens, [], [] : print := [-1, 12]);
#I  ACE 3.001        Wed Oct 31 09:37:37 2001
#I  =========================================
#I  Host information:
#I    name = rigel
#I  ** ERROR (continuing with next line)
#I     no information in table
#I  ***
#I  ***
#I  OVERFLOW (a=249998 r=83333 h=83333 n=249999; l=337 c=0.09; m=249998 t=2499\
```

```
98)
rec( index := 0, cputime := 9, cputimeUnits := "10^-2 seconds",
  activecosets := 249998, maxcosets := 249998, totcosets := 249998 )
```

Essentially, because ACE had not done an enumeration prior to getting the `print` directive, it complained with an "`** ERROR`", recovered and went on with the `start` directive automatically inserted by the `ACEStats` command: no ill effects at the GAP level, but also no table.

Now, let's do what we should have done (to get those first few lines of the coset table), namely, insert the `start` option before the `print` option (the `InfoACE` level is still 3):

```
gap> ACEStats(fgens, [], [] : start, print := [-1, 12]);
#I  ACE 3.001        Wed Oct 31 09:38:28 2001
#I  =========================================
#I  Host information:
#I    name = rigel
#I  OVERFLOW (a=249998 r=83333 h=83333 n=249999; l=337 c=0.10; m=249998 t=2499\
98)
#I  co: a=249998 r=83333 h=83333 n=249999; c=+0.00
#I   coset |      a      A      b      B   order   rep've
#I  -------+---------------------------------------------
#I       1 |      2      3      4      5
#I       2 |      6      1      7      8       0   a
#I       3 |      1      9     10     11       0   A
#I       4 |     12     13     14      1       0   b
#I       5 |     15     16      1     17       0   B
#I       6 |     18      2     19     20       0   aa
#I       7 |     21     22     23      2       0   ab
#I       8 |     24     25      2     26       0   aB
#I       9 |      3     27     28     29       0   AA
#I      10 |     30     31     32      3       0   Ab
#I      11 |     33     34      3     35       0   AB
#I      12 |     36      4     37     38       0   ba
#I  ***
rec( index := 0, cputime := 10, cputimeUnits := "10^-2 seconds",
  activecosets := 249998, maxcosets := 249998, totcosets := 249998 )
```

The values we gave to the `print` option, told ACE to print the first 12 lines and include coset representatives. Note that, since there are no relators, the table has separate columns for generator inverses. So the default workspace of 1000000 words allows a table of $249998 = 1000000/4 - 2$ cosets. Since row `filling` (see 4.14.1) is on by default, the table is simply filled with cosets in order. Note that a compaction phase is done before printing the table, but that this does nothing here (the lowercase `co:` tag), since there are no dead cosets. The coset representatives are simply all possible freely reduced words, in length plus lexicographic (i.e. `lenlex`; see Section 3.4) order.

Using `ACECosetTableFromGensAndRels`

The essential difference between the functions `ACEStats` and `ACECosetTableFromGensAndRels` is that `ACEStats` parses the **results message** from the ACE binary and outputs a GAP record containing statistics of the enumeration, and `ACECosetTableFromGensAndRels` after parsing the **results message**, goes on to parse ACE's coset table, if it can, and outputs a GAP list of lists version of that table. So, if we had used `ACECosetTableFromGensAndRels` instead of `ACEStats` in our examples above, we would have observed similar output, except that we would have ended up in a `break`-loop (because the enumeration overflows) instead of obtaining a record containing enumeration statistics. We have already seen an example of that in Section 1.2. So, here we will consider two options that prevent one entering a `break`-loop, namely the `silent` (see 4.11.3) and `incomplete` (see 4.11.6) options. Firstly, let's take the last ACEStats example, but use ACECosetTableFromGensAndRels instead and include the silent option. (We still have the InfoACE level set at 3.)

```
gap> ACECosetTableFromGensAndRels(fgens, [], [] : start, print := [-1, 12],
>                                                 silent);
#I  ACE 3.001        Wed Oct 31 09:40:18 2001
#I  ========================================
#I  Host information:
#I    name = rigel
#I  OVERFLOW (a=249998 r=83333 h=83333 n=249999; l=337 c=0.09; m=249998 t=2499\
98)
#I  co: a=249998 r=83333 h=83333 n=249999; c=+0.00
#I   coset |      a      A      b      B   order   rep've
#I  -------+---------------------------------------------
#I       1 |      2      3      4      5
#I       2 |      6      1      7      8       0   a
#I       3 |      1      9     10     11       0   A
#I       4 |     12     13     14      1       0   b
#I       5 |     15     16      1     17       0   B
#I       6 |     18      2     19     20       0   aa
#I       7 |     21     22     23      2       0   ab
#I       8 |     24     25      2     26       0   aB
#I       9 |      3     27     28     29       0   AA
#I      10 |     30     31     32      3       0   Ab
#I      11 |     33     34      3     35       0   AB
#I      12 |     36      4     37     38       0   ba
#I  ***
fail
```

Since, the enumeration overflowed and the silent option was set, ACECosetTableFromGensAndRels simply returned fail. But hang on, ACE at least has a partial table; we should be able to obtain it in GAP format, in a situation like this. We can. We simply use the incomplete option, instead of the silent option. However, if we did that with the example above, the result would be an enormous table (the number of **active cosets** is 249998); so let us also set the max (see 4.17.6) option, in order that we should get a more modestly sized partial table. Finally, we will use print := -12 since it is a shorter equivalent alternative to print := [-1, 12]. Note that the output here was obtained with GAP 4.3 (and is the same with GAP 4.4).

Note: Sinec the order options are passed to ACE behind the colon, has not been honoured since GAP 4.5 (at about the time ACE 5.1 was released in 2012), the behaviour exhibited below is no longer observed. To approximately get the behaviour below, omit the option start. This option is inserted anyway, if a user omits it, and importantly is inserted after the max option is put to the ACE binary.

```
gap> ACECosetTableFromGensAndRels(fgens, [], [] : max := 12,
>                                                 start, print := -12,
>                                                 incomplete);
#I  ACE 3.001        Wed Oct 31 09:41:14 2001
#I  ========================================
#I  Host information:
#I    name = rigel
#I  OVERFLOW (a=12 r=4 h=4 n=13; l=5 c=0.00; m=12 t=12)
#I  co: a=12 r=4 h=4 n=13; c=+0.00
#I   coset |      a      A      b      B   order   rep've
#I  -------+---------------------------------------------
#I       1 |      2      3      4      5
```

```
#I       2 |      6      1      7      8       0   a
#I       3 |      1      9     10     11       0   A
#I       4 |     12      0      0      1       0   b
#I       5 |      0      0      1      0       0   B
#I       6 |      0      2      0      0       0   aa
#I       7 |      0      0      0      2       0   ab
#I       8 |      0      0      2      0       0   aB
#I       9 |      3      0      0      0       0   AA
#I      10 |      0      0      0      3       0   Ab
#I      11 |      0      0      3      0       0   AB
#I      12 |      0      4      0      0       0   ba
#I  ***
#I  co: a=12 r=4 h=4 n=13; c=+0.00
#I   coset |      a      A      b      B
#I  -------+----------------------------
#I       1 |      2      3      4      5
#I       2 |      6      1      7      8
#I       3 |      1      9     10     11
#I       4 |     12      0      0      1
#I       5 |      0      0      1      0
#I       6 |      0      2      0      0
#I       7 |      0      0      0      2
#I       8 |      0      0      2      0
#I       9 |      3      0      0      0
#I      10 |      0      0      0      3
#I      11 |      0      0      3      0
#I      12 |      0      4      0      0
#I  ACECosetTable: Coset table is incomplete, reduced & lenlex standardised.
[ [ 2, 6, 1, 12, 0, 0, 0, 0, 3, 0, 0, 0 ],
  [ 3, 1, 9, 0, 0, 2, 0, 0, 0, 0, 0, 4 ],
  [ 4, 7, 10, 0, 1, 0, 0, 2, 0, 0, 3, 0 ],
  [ 5, 8, 11, 1, 0, 0, 2, 0, 0, 3, 0, 0 ] ]
```

Observe, that despite the fact that ACE is able to define coset representatives for all 12 coset numbers defined, the body of the coset table now contains a 0 at each place formerly occupied by a coset number larger than 12 (0 essentially represents "don't know"). To get a table that is the same in the first 12 rows as before we would have had to set `max` to 38, since that was the largest coset number that appeared in the body of the 12-line table, previously. Also, note that the `max` option **preceded** the `start` option; since the interface respects the order in which options are put by the user, the enumeration invoked by `start` would otherwise have only been restricted by the size of `workspace` (see 4.17.1). The warning that the coset table is incomplete is emitted at `InfoACE` or `InfoWarning` level 1, i.e. by default, you will see it.

Using `ACEStart`

The limitation of the functions `ACEStats` and `ACECosetTableFromGensAndRels` (on three arguments) is that they do not **interact** with ACE; they call ACE with user-defined input, and collect and parse the output for either statistics or a coset table. On the other hand, the `ACEStart` (see 6.1.1) function allows one to start up an ACE process and maintain a dialogue with it. Moreover, via the functions `ACEStats` and `ACECosetTable` (on 1 or no arguments), one is able to extract the same information that we could with the non-interactive versions of these functions. However, we can also do a lot more. Each ACE option that provides output that can be used from within GAP has a corresponding interactive interface function that parses and translates that output into a form usable from within GAP.

Now we emulate our (successful) `ACEStats` exchanges above, using interactive ACE interface functions. We could do this with: `ACEStart(0, fgens, [], [] : start, sr := 1);` where the `0` first argument tells `ACEStart` not to insert start after the options explicitly listed. Alternatively, we may do the following (note that the InfoACE level is still 3):

```
gap> ACEStart(fgens, [], []);
#I  ACE 3.001        Wed Oct 31 09:42:49 2001
#I  =========================================
#I  Host information:
#I    name = rigel
#I  ***
#I  OVERFLOW (a=249998 r=83333 h=83333 n=249999; l=337 c=0.10; m=249998 t=2499\
98)
1
gap> ACEParameters(1);
#I    #--- ACE 3.001: Run Parameters ---
#I  Group Name: G;
#I  Group Generators: ab;
#I  Group Relators: ;
#I  Subgroup Name: H;
#I  Subgroup Generators: ;
#I  Wo:1000000; Max:249998; Mess:0; Ti:-1; Ho:-1; Loop:0;
#I  As:0; Path:0; Row:1; Mend:0; No:0; Look:0; Com:10;
#I  C:0; R:0; Fi:7; PMod:3; PSiz:256; DMod:4; DSiz:1000;
#I    #---------------------------------
rec( enumeration := "G", subgroup := "H", workspace := 1000000,
  max := 249998, messages := 0, time := -1, hole := -1, loop := 0, asis := 0,
  path := 0, row := 1, mendelsohn := 0, no := 0, lookahead := 0,
  compaction := 10, ct := 0, rt := 0, fill := 7, pmode := 3, psize := 256,
  dmode := 4, dsize := 1000 )
```

Observe that the ACEStart call returned an integer (1, here). All 8 forms of the ACEStart function, return an integer that identifies the interactive ACE interface function initiated or communicated with. We may use this integer to tell any interactive ACE interface function which interactive ACE process we wish to communicate with. Above we passed 1 to the ACEParameters command which caused sr := 1 (see D.5.7) to be passed to the interactive ACE process indexed by 1 (the process we just started), and a record containing the parameter options (see 4.12.1) is returned. Note that the "Run Parameters": Group Generators, Group Relators and Subgroup Generators are considered "args" (i.e. arguments) and a record containing these is returned by the GetACEArgs (see 6.5.5) command; or they may be obtained individually via the commands: ACEGroupGenerators (see 6.5.1), ACERelators (see 6.5.2), or ACESubgroupGenerators (see 6.5.3).

We can obtain the enumeration statistics record, via the interactive version of ACEStats (see 6.6.2) :

```
gap> ACEStats(1); # The interactive version of ACEStats takes 1 or no arg's
rec( index := 0, cputime := 10, cputimeUnits := "10^-2 seconds",
  activecosets := 249998, maxcosets := 249998, totcosets := 249998 )
```

To display 12 lines of the coset table with coset representatives without invoking a further enumeration we could do: ACEStart(0, 1 : print := [-1, 12]);. Alternatively, we may use the ACEDisplayCosetTable (see 6.5.12) (the table itself is emitted at InfoACE level 1, since by default we presumably want to see it):

```
gap> ACEDisplayCosetTable(1, [-1, 12]);
#I   co: a=249998 r=83333 h=83333 n=249999; c=+0.00
#I    coset |      a      A      b      B   order   rep've
#I   -------+---------------------------------------------
#I        1 |      2      3      4      5
#I        2 |      6      1      7      8       0   a
#I        3 |      1      9     10     11       0   A
#I        4 |     12     13     14      1       0   b
#I        5 |     15     16      1     17       0   B
#I        6 |     18      2     19     20       0   aa
#I        7 |     21     22     23      2       0   ab
#I        8 |     24     25      2     26       0   aB
#I        9 |      3     27     28     29       0   AA
#I       10 |     30     31     32      3       0   Ab
#I       11 |     33     34      3     35       0   AB
#I       12 |     36      4     37     38       0   ba
#I   ------------------------------------------------------------
```

Still with the same interactive ACE process we can now emulate the `ACECosetTableFromGensAndRels` exchange that gave us an incomplete coset table. Note that it is still necessary to invoke an enumeration after setting the `max` (see 4.17.6) option. We could just call `ACECosetTable` with the argument 1 and the same 4 options we used for `ACECosetTableFromGensAndRels`. Alternatively, we can do the equivalent of the 4 options one (or two) at a time, via their equivalent interactive commands. Note that the `ACEStart` command (without `0` as first argument) inserts a `start` directive after the user option `max`:

```
gap> ACEStart(1 : max := 12);
#I   ***
#I   OVERFLOW (a=12 r=4 h=4 n=13; l=5 c=0.00; m=12 t=12)
1
```

Now the following `ACEDisplayCosetTable` command does the equivalent of the `print := [-1, 12]` option.

```
gap> ACEDisplayCosetTable(1, [-1, 12]);
#I   co: a=12 r=4 h=4 n=13; c=+0.00
#I    coset |      a      A      b      B   order   rep've
#I   -------+---------------------------------------------
#I        1 |      2      3      4      5
#I        2 |      6      1      7      8       0   a
#I        3 |      1      9     10     11       0   A
#I        4 |     12      0      0      1       0   b
#I        5 |      0      0      1      0       0   B
#I        6 |      0      2      0      0       0   aa
#I        7 |      0      0      0      2       0   ab
#I        8 |      0      0      2      0       0   aB
#I        9 |      3      0      0      0       0   AA
#I       10 |      0      0      0      3       0   Ab
#I       11 |      0      0      3      0       0   AB
#I       12 |      0      4      0      0       0   ba
#I   ------------------------------------------------------------
```

Finally, we call `ACECosetTable` with 1 argument (which invokes the interactive version of `ACECosetTableFromGensAndRels`) with the option `incomplete`.

```
gap> ACECosetTable(1 : incomplete);
#I  start = yes, continue = yes, redo = yes
#I  ***
#I  OVERFLOW (a=12 r=4 h=4 n=13; l=4 c=0.00; m=12 t=12)
#I  co: a=12 r=4 h=4 n=13; c=+0.00
#I   coset |      a      A      b      B
#I  -------+---------------------------
#I       1 |      2      3      4      5
#I       2 |      6      1      7      8
#I       3 |      1      9     10     11
#I       4 |     12      0      0      1
#I       5 |      0      0      1      0
#I       6 |      0      2      0      0
#I       7 |      0      0      0      2
#I       8 |      0      0      2      0
#I       9 |      3      0      0      0
#I      10 |      0      0      0      3
#I      11 |      0      0      3      0
#I      12 |      0      4      0      0
#I  ACECosetTable: Coset table is incomplete, reduced & lenlex standardised.
[ [ 2, 6, 1, 12, 0, 0, 0, 0, 3, 0, 0, 0 ],
  [ 3, 1, 9, 0, 0, 2, 0, 0, 0, 0, 0, 4 ],
  [ 4, 7, 10, 0, 1, 0, 0, 2, 0, 0, 3, 0 ],
  [ 5, 8, 11, 1, 0, 0, 2, 0, 0, 3, 0, 0 ] ]
```

Observe the line beginning “`#I start = yes,`” (the first line in the output of `ACECosetTable`). This line appears in response to the option `mode` (see D.3.1) inserted by `ACECosetTable` after any user options; it is inserted in order to check that no user options (possibly made before the `ACECosetTable` call) have invalidated ACE’s coset table. Since the line also says `continue = yes`, the mode `continue` (the least expensive of the three modes; see D.3.4) is directed at ACE which evokes a **results message**. Then `ACECosetTable` extracts the incomplete table via a `print` (see D.5.8) directive. If you wish to see all the options that are directed to ACE, set the `InfoACE` level to 4 (then all such commands are `Info`-ed behind a “`ToACE> `” prompt; see 1.9.3).

Following the standalone manual, we now set things up to do the alternating group A_5, of order 60. (We saw the group A_5 with subgroup C_5 earlier in Section B.3; here we are concerned with observing and remarking on the output from the ACE binary.) We turn messaging on via the `messages` (see 4.18.1) option; setting `messages` to 1 tells ACE to emit a **progress message** on each pass of its main loop. In the example following we set `messages := 1000`, which, for our example, sets the interval between messages so high that we only get the “Run Parameters” block (the same as that obtained with `sr := 1`), no progress messages and the final **results message**. Recall `F` is the free group we defined on generators `fgens`: `"a"` and `"b"`. Here we will be interested in seeing what is transmitted to the ACE binary; so we will set the `InfoACE` level to 4 (what is transmitted to ACE will now appear behind a “`ToACE> `” prompt, and we will still see the messages **from** ACE). Note, that when GAP prints `F.1` (= `fgens[1]`) it displays `a`, but the **variable** `a` is (at the moment) unassigned; so for convenience (in defining relators, for example) we first assign the variable `a` to be `F.1` (and `b` to be `F.2`).

```
gap> SetInfoACELevel(4);
gap> a := F.1;; b := F.2;;
gap> # Enumerating A_5 = < a, b | a^2, b^3, (a*b)^5 >
gap> # over Id (trivial subgp)
gap> ACEStart(1, fgens, [a^2, b^3, (a*b)^5], []
>                                     # 4th arg empty (to define Id)
>            : enumeration := "A_5", # Define the Group Name
>              subgroup := "Id",     # Define the Subgroup Name
```

```
>              max := 0,             # Set ‘max’ back to default (no limit)
>              messages := 1000);    # Progress messages every 1000 iter’ns
#I  ToACE> group:ab;
#I  ToACE> relators:a^2,b^3,a*b*a*b*a*b*a*b*a*b;
#I  ToACE> generators;
#I  ToACE> enumeration:A_5;
#I  ToACE> subgroup:Id;
#I  ToACE> max:0;
#I  ToACE> messages:1000;
#I  ToACE> text:***;
#I  ***
#I  ToACE> text:***;
#I  ***
#I  ToACE> Start;
#I    #--- ACE 3.001: Run Parameters ---
#I  Group Name: A_5;
#I  Group Generators: ab;
#I  Group Relators: (a)^2, (b)^3, (ab)^5;
#I  Subgroup Name: Id;
#I  Subgroup Generators: ;
#I  Wo:1000000; Max:333331; Mess:1000; Ti:-1; Ho:-1; Loop:0;
#I  As:0; Path:0; Row:1; Mend:0; No:3; Look:0; Com:10;
#I  C:0; R:0; Fi:6; PMod:3; PSiz:256; DMod:4; DSiz:1000;
#I    #---------------------------------
#I  INDEX = 60 (a=60 r=77 h=1 n=77; l=3 c=0.00; m=66 t=76)
1
```

Observe that the `fgens` and subgroup generators (the empty list) arguments are transmitted to ACE via the ACE binary’s `group` and `generators` options, respectively. Observe also, that the relator `(a*b)^5` is expanded by GAP to `a*b*a*b*a*b*a*b*a*b` when transmitted to ACE and then ACE correctly deduces that it’s `(a*b)^5`.

Since we did not specify a strategy the `default` (see 5.1.1) strategy was followed and hence coset number definitions were R (i.e. HLT) style, and a total of 76 coset numbers (`t=76`) were defined (if we had tried `felsch` we would have achieved the best possible: `t=60`). Note, that ACE already “knew” the group generators and subgroup generators; so, we could have avoided re-transmitting that information by using the `relators` (see D.2.2) option:

```
gap> ACEStart(1 : relators := ToACEWords(fgens, [a^2, b^3, (a*b)^5]),
>                 enumeration := "A_5",
>                 subgroup := "Id",
>                 max := 0,
>                 messages := 1000);
#I  Detected usage of a synonym of one (or more) of the options:
#I      ‘group’, ‘relators’, ‘generators’.
#I  Discarding current values of args.
#I  (The new args will be extracted from ACE, later).
#I  ToACE> relators:a^2,b^3,a*b*a*b*a*b*a*b*a*b;
#I  ToACE> enumeration:A_5;
#I  ToACE> subgroup:Id;
#I  ToACE> max:0;
#I  ToACE> messages:1000;
#I  No group generators saved. Setting value(s) from ACE ...
#I  ToACE> sr:1;
#I    #--- ACE 3.001: Run Parameters ---
```

```
#I  Group Name: A_5;
#I  Group Generators: ab;
#I  Group Relators: (a)^2, bbb, ababababab;
#I  Subgroup Name: Id;
#I  Subgroup Generators: ;
#I  Wo:1000000; Max:333331; Mess:1000; Ti:-1; Ho:-1; Loop:0;
#I  As:0; Path:0; Row:1; Mend:0; No:3; Look:0; Com:10;
#I  C:0; R:0; Fi:6; PMod:3; PSiz:256; DMod:4; DSiz:1000;
#I    #--------------------------------
#I  ToACE> text:***;
#I  ***
#I  ToACE> Start;
#I    #--- ACE 3.001: Run Parameters ---
#I  Group Name: A_5;
#I  Group Generators: ab;
#I  Group Relators: (a)^2, (b)^3, (ab)^5;
#I  Subgroup Name: Id;
#I  Subgroup Generators: ;
#I  Wo:1000000; Max:333331; Mess:1000; Ti:-1; Ho:-1; Loop:0;
#I  As:0; Path:0; Row:1; Mend:0; No:3; Look:0; Com:10;
#I  C:0; R:0; Fi:6; PMod:3; PSiz:256; DMod:4; DSiz:1000;
#I    #--------------------------------
#I  INDEX = 60 (a=60 r=77 h=1 n=77; l=3 c=0.00; m=66 t=76)
1
```

Note the usage of `ToACEWords` (see 6.3.6) to provide the appropriate string value of the `relators` option. Also, observe the `Info`-ed warning of the action triggered by using the `relators` option, that says that the current values of the “args” (i.e. what would be returned by `GetACEArgs`; see 6.5.5) were discarded, which immediately triggered the action of reinstantiating the value of `ACEData.io[1].args` (which is what the `Info`:

```
#I  No group generators saved. Setting value(s) from ACE ...
```

was all about). Also observe that the “Run Parameters” block was `Info`-ed twice; the first time was due to `ACEStart` emitting `sr` with value `1` to ACE, the response of which is used to re-instantiate `ACEData.io[1].args`, and the second is in response to transmitting `Start` to ACE.

In particular, GAP no longer thinks `fgens` are the group generators:

```
gap> ACEGroupGenerators(1) = fgens;
false
```

Groan! We will just have to re-instantiate everything:

```
gap> fgens := ACEGroupGenerators(1);;
gap> F := GroupWithGenerators(fgens);; a := F.1;; b := F.2;;
```

We now define a non-trivial subgroup, of small enough index, to make the observation of all progress messages, by setting `messages := 1`, a not too onerous proposition. As for defining the relators, we could use the 1-argument version of `ACEStart`, in which case we would use the `subgroup` (see 4.19.2) option with the value `ToACEWords(fgens, [ a*b ])`. However, as we saw, in the end we don't save anything by doing this, since afterwards the variables `fgens`, `a`, `b` and `F` would no longer be associated with `ACEStart` process 1. Instead, we will use the more convenient 4-argument form, and also switch the `InfoACELevel` back to 3:

```
gap> SetInfoACELevel(3);
gap> ACEStart(1, ACEGroupGenerators(1), ACERelators(1), [ a*b ]
>               : messages := 1);
#I  ***
#I  ***
#I    #--- ACE 3.001: Run Parameters ---
#I  Group Name: A_5;
#I  Group Generators: ab;
#I  Group Relators: (a)^2, (b)^3, (ab)^5;
#I  Subgroup Name: Id;
#I  Subgroup Generators: ab;
#I  Wo:1000000; Max:333331; Mess:1; Ti:-1; Ho:-1; Loop:0;
#I  As:0; Path:0; Row:1; Mend:0; No:3; Look:0; Com:10;
#I  C:0; R:0; Fi:6; PMod:3; PSiz:256; DMod:4; DSiz:1000;
#I    #--------------------------------
#I  AD: a=2 r=1 h=1 n=3; l=1 c=+0.00; m=2 t=2
#I  SG: a=2 r=1 h=1 n=3; l=1 c=+0.00; m=2 t=2
#I  RD: a=3 r=1 h=1 n=4; l=2 c=+0.00; m=3 t=3
#I  RD: a=4 r=2 h=1 n=5; l=2 c=+0.00; m=4 t=4
#I  RD: a=5 r=2 h=1 n=6; l=2 c=+0.00; m=5 t=5
#I  RD: a=6 r=2 h=1 n=7; l=2 c=+0.00; m=6 t=6
#I  RD: a=7 r=2 h=1 n=8; l=2 c=+0.00; m=7 t=7
#I  RD: a=8 r=2 h=1 n=9; l=2 c=+0.00; m=8 t=8
#I  RD: a=9 r=2 h=1 n=10; l=2 c=+0.00; m=9 t=9
#I  CC: a=8 r=2 h=1 n=10; l=2 c=+0.00; d=0
#I  RD: a=9 r=5 h=1 n=11; l=2 c=+0.00; m=9 t=10
#I  RD: a=10 r=5 h=1 n=12; l=2 c=+0.00; m=10 t=11
#I  RD: a=11 r=5 h=1 n=13; l=2 c=+0.00; m=11 t=12
#I  RD: a=12 r=5 h=1 n=14; l=2 c=+0.00; m=12 t=13
#I  RD: a=13 r=5 h=1 n=15; l=2 c=+0.00; m=13 t=14
#I  RD: a=14 r=5 h=1 n=16; l=2 c=+0.00; m=14 t=15
#I  CC: a=13 r=6 h=1 n=16; l=2 c=+0.00; d=0
#I  CC: a=12 r=6 h=1 n=16; l=2 c=+0.00; d=0
#I  INDEX = 12 (a=12 r=16 h=1 n=16; l=3 c=0.00; m=14 t=15)
1
```

Observe that we used `ACERelators(1)` (see 6.5.2) to grab the value of the relators we had defined earlier. We also used `ACEGroupGenerators(1)` (see 6.5.1) to get the group generators.

The run ended with 12 active (see Section 3.5) coset numbers (`a=12`) after defining a total number of 15 coset numbers (`t=15`); the definitions occurred at the steps with progress messages tagged by `AD:` (coset 1 application definition) and `SG:` (subgroup generator phase), and the 13 tagged by `RD:` (R style definition). So there must have been 3 coincidences: observe that there were 3 progress messages with a `CC:` tag. (See Appendix A.)

We can dump out the statistics accumulated during the run, using `ACEDumpStatistics` (see 6.5.24), which `Infos` the ACE output of the `statistics` (see D.5.10) at `InfoACE` level 1.

```
gap> ACEDumpStatistics();
#I    #- ACE 3.001: Level 0 Statistics -
#I  cdcoinc=0 rdcoinc=2 apcoinc=0 rlcoinc=0 clcoinc=0
#I    xcoinc=2 xcols12=4 qcoinc=3
#I    xsave12=0 s12dup=0 s12new=0
#I    xcrep=6 crepred=0 crepwrk=0 xcomp=0 compwrk=0
#I  xsaved=0 sdmax=0 sdoflow=0
```

```
#I  xapply=1 apdedn=1 apdefn=1
#I  rldedn=0 cldedn=0
#I  xrdefn=1 rddedn=5 rddefn=13 rdfill=0
#I  xcdefn=0 cddproc=0 cdddedn=0 cddedn=0
#I    cdgap=0 cdidefn=0 cdidedn=0 cdpdl=0 cdpof=0
#I    cdpdead=0 cdpdefn=0 cddefn=0
#I    #---------------------------------
```

The statistic qcoinc=3 states what we had already observed, namely, that there were three coincidences. Of these, two were primary coincidences (rdcoinc=2). Since t=15, there were fourteen non-trivial coset number definitions; one was during the application of coset 1 to the subgroup generator (apdefn=1), and the remainder occurred during applications of the coset numbers to the relators (rddefn=13). For more details on the meanings of the variables you will need to read the C code comments.

Now let us display all 12 lines of the coset table with coset representatives.

```
gap> ACEDisplayCosetTable([-12]);
#I  CO: a=12 r=13 h=1 n=13; c=+0.00
#I   coset |      b      B      a   order   rep've
#I  -------+-------------------------------------
#I       1 |      3      2      2
#I       2 |      1      3      1       3   B
#I       3 |      2      1      4       3   b
#I       4 |      8      5      3       5   ba
#I       5 |      4      8      6       2   baB
#I       6 |      9      7      5       5   baBa
#I       7 |      6      9      8       3   baBaB
#I       8 |      5      4      7       5   bab
#I       9 |      7      6     10       5   baBab
#I      10 |     12     11      9       3   baBaba
#I      11 |     10     12     12       2   baBabaB
#I      12 |     11     10     11       3   baBabab
#I  ------------------------------------------------------------
```

Note how the pre-printout compaction phase now does some work (indicated by the upper-case CO: tag), since there were coincidences, and hence dead coset numbers. Note how b and B head the first two columns, since ACE requires that the first two columns be occupied by a generator/inverse pair or a pair of involutions. The a column is also the A column, as a is an involution.

We now use ACEStandardCosetNumbering to produce a lenlex standard table within ACE, but note that this is only lenlex with respect to the ordering b, a of the generators. Then we call ACEDisplayCosetTable again to see it. Observe that at both the standardisation and coset table display steps a compaction phase is invoked but on both occasions the lowercase co: tag indicates that nothing is done (all the recovery of dead coset numbers that could be done was done earlier).

```
gap> ACEStandardCosetNumbering();
#I  co/ST: a=12 r=13 h=1 n=13; c=+0.00
gap> ACEDisplayCosetTable([-12]);
#I  co: a=12 r=13 h=1 n=13; c=+0.00
#I   coset |      b      B      a   order   rep've
#I  -------+-------------------------------------
#I       1 |      2      3      3
#I       2 |      3      1      4       3   b
#I       3 |      1      2      1       3   B
```

```
#I      4 |      5      6      2       5   ba
#I      5 |      6      4      7       5   bab
#I      6 |      4      5      8       2   baB
#I      7 |      8      9      5       5   baba
#I      8 |      9      7      6       5   baBa
#I      9 |      7      8     10       3   babaB
#I     10 |     11     12      9       3   babaBa
#I     11 |     12     10     12       3   babaBab
#I     12 |     10     11     11       2   babaBaB
#I  ------------------------------------------------------------
```

Of course, the table above is not `lenlex` with respect to the order of the generators we had originally given to ACE; to get that, we would have needed to specify `lenlex` (see 4.11.4) at the enumeration stage. The effect of the `lenlex` option at the enumeration stage is the following: behind the scenes it ensures that the relator `a^2` is passed to ACE as `aa` and then it sets the option `asis` to 1; this bit of skulduggery stops ACE treating `a` as an involution, allowing `a` and `A` (the inverse of `a`) to take up the first two columns of the coset table, effectively stopping ACE from reordering the generators. To see what is passed to ACE, at the enumeration stage, we set the `InfoACELevel` to 4, but since we don't really want to see messages this time we set `messages := 0`.

```
gap> SetInfoACELevel(4);
gap> ACEStart(1, ACEGroupGenerators(1), ACERelators(1), [ a*b ]
>              : messages := 0, lenlex);
#I  ToACE> group:ab;
#I  ToACE> relators:aa, b^3,a*b*a*b*a*b*a*b*a*b;
#I  ToACE> generators:a*b;
#I  ToACE> asis:1;
#I  ToACE> messages:0;
#I  ToACE> text:***;
#I  ***
#I  ToACE> text:***;
#I  ***
#I  ToACE> Start;
#I  INDEX = 12 (a=12 r=17 h=1 n=17; l=3 c=0.00; m=15 t=16)
1
gap> ACEStandardCosetNumbering();
#I  ToACE> standard;
#I  CO/ST: a=12 r=13 h=1 n=13; c=+0.00
gap> # The capitalised 'CO' indicates space was recovered during compaction
gap> ACEDisplayCosetTable([-12]);
#I  ToACE> print:-12;
#I  ToACE> text:------------------------------------------------------------;
#I  co: a=12 r=13 h=1 n=13; c=+0.00
#I   coset |      a      A      b      B   order   rep've
#I  -------+--------------------------------------------
#I      1 |      2      2      3      2
#I      2 |      1      1      1      3       2   a
#I      3 |      4      4      2      1       3   b
#I      4 |      3      3      5      6       5   ba
#I      5 |      7      7      6      4       5   bab
#I      6 |      8      8      4      5       2   baB
#I      7 |      5      5      8      9       5   baba
#I      8 |      6      6      9      7       5   baBa
```

```
#I        9 |     10       10        7        8        3    babaB
#I       10 |      9        9       11       12        3    babaBa
#I       11 |     12       12       12       10        3    babaBab
#I       12 |     11       11       10       11        2    babaBaB
#I   ------------------------------------------------------------
```

You may have noticed the use of ACE's `text` option several times above; this just tells ACE to print the argument given to `text` (as a comment). This is used by the GAP interface as a sentinel; when the string appears in the ACE output, the GAP interface knows not to expect anything else.

C.2 Emulating Sims

Here we consider the various `sims` strategies (see 5.1.8), with respect to duplicating Sims' example statistics of his strategies given in Section 5.5 of [Sim94], and giving approximations of his even-numbered strategies.

In order to duplicate Sims' maximum active coset numbers and total coset numbers statistics, one needs to work with the formal inverses of the relators and subgroup generators from [Sim94], since definitions are made from the front in Sims' routines and from the rear in ACE. Also, in instances where `IsACEGeneratorsInPreferredOrder(`*gens*`,` *rels*`)` returns `false`, for group generators *fgens* and relators *rels*, one will need to apply the `lenlex` option to stop ACE from re-ordering the generators and relators (see 1.2.3 and 4.11.4). In general, we can match Sims' values for the `sims := 1` and `sims := 3` strategies (the R style and R* style Sims strategies with `mendelsohn` off) and for the `sims := 9` (C style) strategy, but sometimes we may not exactly match Sims' statistics for the `sims := 5` and `sims := 7` strategies (the R style and R* style Sims strategies with `mendelsohn` on); Sims does not specify an order for the (Mendelsohn) processing of cycled relators and evidently ACE's processing order is different to the one Sims used in his CHLT algorithm to get his statistics (see 4.13.5).

Note: HLT as it appears in Table 5.5.1 of [Sim94] is achieved in ACE with the sequence "`hlt, lookahead := 0`" and CHLT is (nearly) equivalent to "`hlt, lookahead := 0, mendelsohn`"; also Sims' *save* `= false` equates to R style (`rt` positive, `ct := 0`) in ACE, and *save* `= true`, for Sims' HLT and CHLT, equates to R* style (`rt` negative, `ct := 0`) in ACE. Sims' Felsch strategy coincides with ACE's `felsch := 0` strategy, i.e. `sims := 9` is identical to `felsch := 0`. (See the options 5.1.5, 4.15.2, 4.13.5, 4.13.2, 4.13.3 and 5.1.3.)

The following duplicates the "Total" (`totcosets` in ACE) and "Max. Active" (`maxcosets` in ACE) statistics for Example 5.2 of [Sim94], found in Sims' Table 5.5.3, for the `sims := 3` strategy.

```
gap> SetInfoACELevel(1); # No behind-the-scenes info. please
gap> F := FreeGroup("r", "s", "t");; r := F.1;; s := F.2;; t := F.3;;
gap> ACEStats([r, s, t], [(r^t*r^-2)^-1, (s^r*s^-2)^-1, (t^s*t^-2)^-1], []
>             : sims := 3);
rec( index := 1, cputime := 0, cputimeUnits := "10^-2 seconds",
  activecosets := 1, maxcosets := 673, totcosets := 673 )
```

By replacing `sims := 3` with `sims :=` *i* for *i* equal to 1, 5, 7 or 9, one may verify that for *i* equal to 1 or 9, Sims' statistics are again duplicated, and observe a slight variance with Sims' statistics for *i* equal to 5 or 7.

Now, we show how one can approximate any one of Sims' even-numbered strategies. Essentially, the idea is to start an interactive ACE process using `ACEStart` (see 6.1.1) with `sims :=` *i*, for *i* equal to 1, 3, 5, 7 or 9, and `max` set to some low value *maxstart* so that the enumeration stops after only completing a few rows of the coset table. Then, to approximate Sims' strategy *i* `+ 1`, one alternately applies `ACEStandardCosetNumbering` and `ACEContinue`, progressively increasing the value of `max` by some value *maxstep*. The general algorithm is provided by the `ACEEvenSims` function following.

```
gap> ACEEvenSims := function(fgens, rels, sgens, i, maxstart, maxstep)
>      local j;
>      j := ACEStart(fgens, rels, sgens : sims := i, max := maxstart);
>      while ACEStats(j).index = 0 do
>        ACEStandardCosetNumbering(j);
>        ACEContinue(j : max := ACEParameters(j).max + maxstep);
>      od;
>      return ACEStats(j);
>    end;;
```

It turns out that one can duplicate the Sims' strategy 4 statistics in Table 5.5.3 of [Sim94], with *i* `= 3` (so that *i* `+ 1 = 4`), *maxstart* `= 14` and *maxstep* `= 50`:

```
gap> ACEEvenSims([r, s, t], [(r^t*r^-2)^-1, (s^r*s^-2)^-1, (t^s*t^-2)^-1],
>                []、 3, 14, 50);
rec( index := 1, cputime := 0, cputimeUnits := "10^-2 seconds",
  activecosets := 1, maxcosets := 393, totcosets := 393 )
```

Setting *maxstep* `= 60` (and leaving the other parameters the same) also gives Sims' statistics, but *maxstart* `= 64` with *maxstep* `= 80` does better:

```
gap> ACEEvenSims([r, s, t], [(r^t*r^-2)^-1, (s^r*s^-2)^-1, (t^s*t^-2)^-1],
>                [], 3, 64, 80);
rec( index := 1, cputime := 0, cputimeUnits := "10^-2 seconds",
  activecosets := 1, maxcosets := 352, totcosets := 352 )
```

Even though the (`lenlex`) standardisation steps in the above examples produce a significant improvement over the `sims := 3` statistics, this does not happen universally. Sims [Sim94] gives many examples where the even-numbered strategies fail to show any significant improvement over the odd-numbered strategies, and one example (see Table 5.5.7) where `sims := 2` gives a performance that is very much worse than any of the other Sims strategies. As with any of the strategies, what works well for some groups may not work at all well with other groups. There are **no** general rules. It's a bit of a game. Let's hope you win most of the time.

D Other ACE Options

Here we list all the known ACE options not provided earlier. Most of the options provided here have interactive function alternatives (each such alternative is noted at the end of the section describing the corresponding option and introduced by "**INTERACTIVELY**, use ..."). A few options have only limited usefulness from GAP; many options, users will normally only wish to use if generating an input file, by using the option `aceinfile` (see 4.11.7). However all options here are functional, both interactively and non-interactively.

D.1 Experimentation Options

1 ▶ `aep:=`*val*

Runs the enumeration for all equivalent presentations; *val* is in the integer range 1 to 7.

The `aep` option runs an enumeration for combinations of relator ordering, relator rotations, and relator inversions.

The argument *val* is considered as a binary number. Its three bits are treated as flags, and control relator rotations (the 2^0 bit), relator inversions (the 2^1 bit) and relator orderings (the 2^2 bit), respectively; where 1 means "active" and 0 means "inactive". (See below for an example).

The `aep` option first performs a "priming run" using the options as they stand. In particular, the `asis` and `messages` options are honoured.

It then turns `asis` on and `messages` off (i.e. sets `messages` to 0), and generates and tests the requested equivalent presentations. The maximum and minimum values attained by `m` (the maximum number of coset numbers defined at any stage) and `t` (the total number of coset numbers defined) are tracked, and each time a new "record" is found, the relators used and the summary result line is printed. See Appendix A for a discussion of the statistics `m` and `t`. To observe these messages either set the `InfoLevel` of `InfoACE` to 3 or non-interactively you can peruse the ACE output file (see 4.11.8).

Normally when a non-interactive ACE interface function is called, the option `start` (see D.3.2), is quietly inserted after all the options provided by the user, to initiate a coset enumeration. Since the `aep` option invokes an enumeration, the quiet insertion of the `start` option is neither needed nor done, when a non-interactive ACE interface function is called with the `aep` option.

The order in which the equivalent presentations are generated and tested has no particular significance, but note that the presentation as given **after** the initial priming run) is the **last** presentation to be generated and tested, so that the group's relators are left `unchanged` by running the `aep` option, (not that a non-interactive user cares).

As discussed by Cannon, Dimino, Havas and Watson [CDHW73] and Havas and Ramsay [HR01] such equivalent presentations can yield large variations in the number of coset numbers required in an enumeration. For this command, we are interested in this variation.

After the final presentation is run, some additional status information messages are printed to the ACE output file:

– the number of runs which yielded a finite index;

– the total number of runs (excluding the priming run); and

– the range of values observed for `m` and `t`.

As an example (drawn from the discussion in [HR99]) consider the enumeration of the 448 coset numbers of the subgroup $\langle a^2, a^{-1}b\rangle$ of the group

$$(8,7 \mid 2,3) = \langle a,b \mid a^8 = b^7 = (ab)^2 = (a^{-1}b)^3 = 1\rangle\cdot$$

There are $4! = 24$ relator orderings and $2^4 = 16$ combinations of relator or inverted relator. Exponents are taken into account when rotating relators, so the relators given give rise to 1, 1, 2 and 2 rotations respectively, for a total of $1\cdot 1\cdot 2\cdot 2 = 4$ combinations. So, for `aep` = 7 (resp. 3), $24\cdot 16\cdot 4 = 1536$ (resp. $16\cdot 4 = 64$) equivalent presentations are tested.

Notes: There is no way to stop the `aep` option before it has completed, other than killing the task. So do a reality check beforehand on the size of the search space and the time for each enumeration. If you are interested in finding a "good" enumeration, it can be very helpful, in terms of running time, to put a tight limit on the number of coset numbers via the `max` option. You may also have to set `compaction` = 100 to prevent time-wasting attempts to recover space via compaction. This maximises throughput by causing the "bad" enumerations, which are in the majority, to overflow quickly and abort. If you wish to explore a very large search-space, consider firing up many copies of ACE, and starting each with a "random" equivalent presentation. Alternatively, you could use the `rep` command.

INTERACTIVELY, use `ACEAllEquivPresentations` (see 6.4.1).

2 ► `rep:=`*val*
► `rep:=[`*val*`,` *Npresentations*`]`

Run the enumeration for `random` equivalent presentations; *val* is in the integer range 1 to 7; *Npresentations* must be a positive integer.

The `rep` (random equivalent presentations) option complements the `aep` option. It generates and tests some random equivalent presentations. The argument *val* acts as for `aep`. It is also possible to set the number *Npresentations* of random presentations used (by default, eight are used), by using the extended syntax `rep:=[`*val*`,`*Npresentations*`]`.

The routine first turns `asis` on and `messages` off (i.e. sets `messages` to 0), and then generates and tests the requested number of random equivalent presentations. For each presentation, the relators used and the summary result line are printed. To observe these messages either set the `InfoLevel` of `InfoACE` to at least 3 or non-interactively you can peruse the ACE output file (see 4.11.8).

Normally when a non-interactive ACE interface function is called, the option `start` (see D.3.2), is quietly inserted after all the options provided by the user, to initiate a coset enumeration. Since the `rep` option invokes an enumeration, the quiet insertion of the `start` option is neither needed nor done, when a non-interactive ACE interface function is called with the `rep` option.

Notes: The relator inversions and rotations are "genuinely" random. The relator permuting is a little bit of a kludge, with the "quality" of the permutations tending to improve with successive presentations. When the `rep` command completes, the presentation active is the **last** one generated, (not that the non-interactive user cares).

Guru Notes: It might appear that neglecting to restore the original presentation is an error. In fact, it is a useful feature! Suppose that the space of equivalent presentations is too large to exhaustively test. As noted in the entry for `aep`, we can start up multiple copies of `aep` at random points in the search-space. Manually generating `random` equivalent presentations to serve as starting-points is tedious and error-prone. The `rep` option provides a simple solution; simply run `rep := 7` before `aep := 7`.

INTERACTIVELY, use `ACERandomEquivPresentations` (see 6.4.2).

D.2 Options that Modify a Presentation

1 ► `group:=`*grpgens*

Defines the `group` generators; *grpgens* should be an integer (that is the number of generators) or a string that is the concatenation of, or a list of, single-lowercase-letter group generator names, i.e. it should be in a form suitable for the ACE binary to interpret. (Shortest abbreviation: `gr`.)

The group generators should normally be input as one of the arguments of an ACE interface function, though this option may be useful when `ACEStart` (see 6.1.1) is called with the single argument 0. This option may also be useful for re-using an interactive process for a new enumeration, rather than using `ACEQuit` to kill the process and `ACEStart` to initiate a new process. If the generators each have names that as strings are single lowercase letters, those same strings are used to represent the same generators by ACE; otherwise, ACE will represent each generator by an integer, numbered sequentially from 1.

To convert a GAP list *fgens* of free group generators into a form suitable for the `group` option, use the construction: `ToACEGroupGenerators(`*fgens*`)` (see 6.3.5). It is **strongly recommended** that users of the `group` option use this construction.

Notes: Any use of the `group` command which actually defines generators invalidates any previous enumeration, and stays in effect until the next `group` command. Any words for the group or subgroup must be entered using the nominated generator format, and all printout will use this format. A valid set of generators is the minimum information necessary before ACE will attempt an enumeration.

Guru Notes: The columns of the coset table are allocated in the same order as the generators are listed, insofar as this is possible, given that the first two columns must be a generator/inverse pair or a pair of involutions. The ordering of the columns can, in some cases, affect the definition sequence of cosets and impact the statistics of an enumeration.

2 ► `relators:=`*relators*

Defines the group `relators`; *relators* must be a string or list of strings that the ACE binary can interpret as words in the group generators. (Shortest abbreviation: `rel`.)

The group relators should normally be input as one of the arguments of an ACE interface function, but this option may occasionally be useful with interactive processes (see D.2.1). If *wordList* is an empty list, the group is free.

To convert a GAP list *rels* of relators in the free group generators *fgens* into a form suitable for the `relators` option, use the construction: `ToACEWords(`*fgens*`, `*rels*`)` (see 6.3.6).

3 ► `generators:=`*subgens*

Defines the subgroup `generators`; *subgens* must be a string or list of strings that the ACE binary can interpret as words in the group generators. (Shortest abbreviation: `gen`.)

The subgroup generators should normally be input as one of the arguments of an ACE interface function, but this option may occasionally be useful with interactive processes (see D.2.1). By default, there are no subgroup generators and the subgroup is trivial. This command allows a list of subgroup generating words to be entered.

To convert a GAP list *sgens* of subgroup generators in the free group generators *fgens* into a form suitable for the `generators` option, use the construction: `ToACEWords(`*fgens*`, `*sgens*`)` (see 6.3.6).

4 ► `sg:=`*subgens*

Adds the words in *subgens* to any subgroup generators already present; *subgens* must be a string or list of strings that the ACE binary can interpret as words in the group generators.

The enumeration must be (re)`start`ed or `redone`, it cannot be `continue`d.

To convert a GAP list *sgens* of subgroup generators in the free group generators *fgens* into a form suitable for the `generators` option, use the construction: `ToACEWords(`*fgens*`, `*sgens*`)` (see 6.3.6).

INTERACTIVELY, use `ACEAddSubgroupGenerators` (see 6.7.4).

5 ► `rl:=`*relators*

Appends the relator list *relators* to the existing list of relators present; *relators* must be a string or list of strings that the ACE binary can interpret as words in the group generators.

The enumeration must be (re)`start`ed or `redo`ne, it cannot be `continue`d.

To convert a GAP list *rels* of relators in the free group generators *fgens* into a form suitable for the `rl` option, use the construction: `ToACEWords(`*fgens*`, `*rels*`)` (see 6.3.6).

INTERACTIVELY, use `ACEAddRelators` (see 6.7.3).

6 ► `ds:=`*list*

Deletes subgroup generators; *list* must be a list of positive integers.

This command allows subgroup generators to be deleted from the presentation. If the generators are numbered from 1 in the output of, say, the `sr` command (see D.5.7), then the generators listed in *list* are deleted; *list* must be a strictly increasing sequence.

INTERACTIVELY, use `ACEDeleteSubgroupGenerators` (see 6.7.6).

7 ► `dr:=`*list*

Deletes relators; *list* must be a list of positive integers.

This command allows group relators to be deleted from the presentation. If the relators are numbered from 1 in the output of, say, the `sr` command (see D.5.7), then the relators listed in *list* are deleted; *list* must be a strictly increasing sequence.

INTERACTIVELY, use `ACEDeleteRelators` (see 6.7.5).

8 ► `cc:=`*val*

Makes coset *val* coincide with coset 1; *val* should be a positive integer.

Prints out the representative of coset *val*, and adds it to the subgroup generators; i.e., forces coset *val* to coincide with coset 1, the subgroup.

INTERACTIVELY, use `ACECosetCoincidence` (see 6.7.7).

9 ► `rc:=`*val*
► `rc:=[`*val*`]`
► `rc:=[`*val*`, `*attempts*`]`

Enforce random coincidences; *val* and *attempts* must be positive integers.

This option attempts upto *attempts* (or, in the first and second forms, 8) times to find nontrivial subgroups with index a multiple of *val* by repeatedly making random coset numbers coincident with coset 1 and seeing what happens. The starting coset table must be non-empty, but should not be complete. For each attempt, we repeatedly add random coset representatives to the subgroup and `redo` the enumeration. If the table becomes too small, the attempt is aborted, the original subgroup generators restored, and another attempt made. If an attempt succeeds, then the new set of subgroup generators is retained.

Guru Notes: A coset number can have many different coset representatives. Consider running `standard` before `rc`, to canonicise the table and hence the coset representatives.

INTERACTIVELY, use `ACERandomCoincidences` (see 6.7.8).

D.3 Mode Options

1 ▶ `mode`

Prints the possible enumeration `modes`. (Shortest abbreviation: `mo`.)

Prints the possible enumeration `modes` (i.e. which of `continu`, `redo` or `start` are possible (see D.3.4, D.3.3 and D.3.2).

INTERACTIVELY, use `ACEModes` (see 6.2.1).

2 ▶ `begin`
▶ `start`

Start an enumeration. (Shortest abbreviation of `begin` is `beg`.)

Any existing information in the table is cleared, and the enumeration starts from coset 1 (i.e., the subgroup).

Normally when a non-interactive ACE interface function is called, the option `start` (see D.3.2), is quietly inserted after all the options provided by the user, to initiate a coset enumeration; however, this is not done, if the user herself supplies either the `begin` or `start` option.

INTERACTIVELY, use `ACEStart` (see 6.1.1).

3 ▶ `check`
▶ `redo`

`Redo` an extant enumeration, using the current parameters.

As opposed to `start` (see D.3.2), which clears an existing coset table, any existing information in the table is retained, and the enumeration is restarted from coset 1 (i.e., the subgroup).

Notes: This option is really intended for the case where additional relators (option `rl`; see D.2.5) and/or subgroup generators (option `sg`; see D.2.4) have been introduced. The current table, which may be incomplete or exhibit a finite index, is still **valid**. However, the additional data may allow the enumeration to complete, or cause a collapse to a smaller index.

INTERACTIVELY, use `ACERedo` (see 6.2.3).

4 ▶ `continu`

`Continu`es the current enumeration, building upon the existing table. (Shortest abbreviation: `cont`.)

If a previous run stopped without producing a finite index you can, in principle, change any of the parameters and `continue` on. Of course, if you make any changes which invalidate the current table, you won't be allowed to `continue`, although you may be allowed to `redo` (see D.3.3). If `redo` is not allowed, you must re-`start` (see D.3.2).

Note: The ACE standalone allows the option `continue`, but this is a GAP keyword, and so GAP users must use (mixed-case abbreviations of) `continu`.

INTERACTIVELY, use `ACEContinue` (see 6.2.2).

D.4 Options that Interact with the Operating System

1 ► `ai`
► `ai:=`*filename*

Alter `input` to standard input or *filename*; *filename* must be a string.

By default, commands to ACE are read from standard input (i.e., the keyboard). With no value `ai` causes ACE to revert to reading from standard input; otherwise, the `ai` command closes the current input file, and opens *filename* as the source of commands. If *filename* can't be opened, input reverts to standard input.

Notes: If you switch to taking input from (another) file, remember to switch back before the end of that file; otherwise the `EOF` there will cause ACE to terminate.

2 ► `bye`
► `exit`
► `qui`

Quit ACE. (Shortest abbreviation of `qui` is `q`.)

This quits ACE nicely, printing the date and the time. An `EOF` (end-of-file; i.e., `^d`) has the same effect, so proper termination occurs if ACE is taking its input from a script file.

Note that `qui` actually abbreviates the corresponding ACE directive `quit`, but since `quit` is a GAP keyword it is not available via the GAP interface to ACE.

INTERACTIVELY, use `ACEQuit` (see 6.1.2).

3 ► `system:=`*string*

Does a shell escape, to execute *string*; *string* must be a string. (Shortest abbreviation: `sys`.)

Since GAP already provides `Exec()` for this purpose, this option is unlikely to have a use.

D.5 Query Options

1 ► `cycles`

Prints out the table in `cycles`. (Shortest abbreviation: `cy`.)

This option prints out the permutation representation.

INTERACTIVELY, use `ACECycles` (see 6.5.16).

2 ► `dump`
► `dump:=`*level*
► `dump:=[`*level*`]`
► `dump:=[`*level*`,` *detail*`]`

`Dump` the internal variables of ACE; *level* must be an integer in the range 0 to 2, and *detail* must be 0 or 1. (Shortest abbreviation: `d`.)

The value of *level* determines which of the three levels of ACE to dump. (You will need to read the standalone manual `acce3001.dvi` in the `standalone-doc` directory to understand what Levels 0, 1 and 2 are all about.) The value of *detail* determines the amount of detail (*detail* `= 0` means less detail). The first form (with no arguments) selects *level* `= 0,` *detail* `= 0`. The second form of this command makes *detail* `= 0`. This option is intended for gurus; the source code should be consulted to see what the output means.

INTERACTIVELY, use `ACEDumpVariables` (see 6.5.23).

3 ► `help`

Prints the ACE help screen. (Shortest abbreviation: `h`.)

This option prints the list of options of the ACE binary. Note that this list is longer than a standard screenful.

4 ▶ `nc`
▶ `nc:=`*val*
▶ `normal`
▶ `normal:=`*val*

Check or attempt to enforce normal closure; *val* must be 0 or 1.

This option tests the subgroup for normal closure within the group. If a conjugate of a subgroup generator by a generator, is determined to belong to a coset other than coset 1, it is printed out, and if *val* `= 1`, then any such conjugate is also added to the subgroup generators. With no argument or if *val* `= 0`, ACE does not add any new subgroup generators.

Notes: The method of determination of whether a conjugate of a subgroup generator is in the subgroup, is by testing whether it can be traced from coset 1 to coset 1 (see `trace`: D.5.12).

The resultant subgroup need not be normally closed after executing option `nc` with the value 1. It is still possible that some conjugates of the newly added subgroup generators will not be elements of the subgroup.

INTERACTIVELY, use `ACEConjugatesForSubgroupNormalClosure` (see 6.7.10).

5 ▶ `options`

Dumps version information of the ACE binary. (Shortest abbreviation: `opt`.)

A rather unfortunate name for an option; this command dumps details of the "options" included in the version of ACE when the ACE binary was compiled.

A typical output, is as follows:

```
Executable built:
  Sat Feb 27 15:57:59 EST 1999
Level 0 options:
  statistics package = on
  coinc processing messages = on
  dedn processing messages = on
Level 1 options:
  workspace multipliers = decimal
Level 2 options:
  host info = on
```

INTERACTIVELY and non-interactively, use the command `ACEBinaryVersion();` (see 6.5.25) for this information, instead, unless you want it in an ACE standalone input file.

6 ▶ `oo:=`*val*
▶ `order:=`*val*

Print a coset representative of a coset number with order a multiple of *val* modulo the subgroup; *val* must be an integer.

This option finds a coset with order a multiple of `|`*val*`|` modulo the subgroup, and prints out its coset representative. If *val* `< 0`, then all coset numbers meeting the requirement are printed. If *val* `> 0`, then just the first coset number meeting the requirement is printed. Also, *val* `= 0` is permitted; this special value effects the printing of the orders (modulo the subgroup) of all coset numbers.

INTERACTIVELY, use `ACEOrders` (see 6.5.18), for the case *val* `= 0`, or `ACEOrder` (see 6.5.19), otherwise.

7 ▶ `sr`
▶ `sr:=`*val*

Print out parameters of the current presentation; *val* must be an integer in the range 0 to 5.

No argument, or *val* `= 0`, prints out the `Group Name`, the group's `relators`, `Subgroup Name` and the subgroup's `generators`. If *val* `= 1`, then the `Group Generators` and the current setting of the "run parameters" is also printed. The printout is the same as that produced at the start of a run when option `messages` (see 4.18.1) is non-zero. Also, *val* equal to 2, 3, 4, or 5 print out just the `Group Name`, just the group's `relators`, just the `Subgroup Name`, or just the subgroup's `generators`, respectively.

Notes: The `sr` option should only be used **after** an enumeration run; otherwise, the value 0 for some options will be unreliable. To ensure this occurs non-interactively, ensure one of the options that invokes an enumeration: `start` (see D.3.2) or `aep` (see D.1.1) or `rep` (see D.1.2), precedes the `sr` option. (When an enumeration-invoking option is included non-interactively the quiet inclusion step of the `start` option is omitted.)

INTERACTIVELY, use `ACEGroupGenerators` (see 6.5.1), `ACERelators` (see 6.5.2), `ACESubgroupGenerators` (see 6.5.3), and `ACEParameters` (see 6.5.10).

8 ► `print`
► `print:=`*val*
► `print:=[`*val*`]`
► `print:=[`*val*`,` *last*`]`
► `print:=[`*val*`,` *last*`,` *by*`]`

Compact and print the coset table; *val* must be an integer, and *last* and *by* must be positive integers. (Shortest abbreviation: `pr`.)

In the first (no value) form, `print` prints the entire coset table, without orders or coset representatives. In the second and third forms, the absolute value of *val* is taken to be the last line of the table to be printed (and 1 is taken to be the first); in the fourth and fifth forms, `|`*val*`|` is taken to be the first line of the table to be printed, and *last* is taken to be the number of the last line to be printed. In the last form, the table is printed from line `|`*val*`|` to line *last* in steps of *by*. If *val* is negative, then the orders modulo the subgroup (if available) and coset representatives are printed also.

INTERACTIVELY, use `ACEDisplayCosetTable` (see 6.5.12).

9 ► `sc:=`*val*
► `stabilising:=`*val*

Print out the coset numbers whose elements normalise the subgroup; *val* must be an integer. (Shortest abbreviation of `stabilising` is `stabil`.)

If *val* `> 0`, the first *val* non-trivial (i.e. other than coset 1) coset numbers whose elements normalise the subgroup are printed. If *val* `= 0`, all non-trivial coset numbers whose elements normalise the subgroup, plus their representatives, are printed. If *val* `< 0`, the first `|`*val*`|` non-trivial coset numbers whose elements normalise the subgroup, plus their representatives, are printed.

Note: The name of this option is an historical hangover. It is named for the property that elements that "normalise" a subgroup, may be said to "stabilise" that subgroup when they act "by conjugation". Also, the option `normal` (see D.5.4) already performs a different function.

INTERACTIVELY, use `ACECosetsThatNormaliseSubgroup` (see 6.5.21).

10 ► `statistics`
► `stats`

Dump enumeration statistics. (Shortest abbreviation of `statistics` is `stat`.)

If the statistics package is compiled into the ACE code, which it is by default (see the `options` D.5.5 option), then this option dumps the statistics accumulated during the most recent enumeration. See the `enum.c` source file for the meaning of the variables.

INTERACTIVELY, use `ACEDumpStatistics` (see 6.5.24).

11 ► `style`

Prints the current enumeration style.

This option prints the current enumeration style, as deduced from the current `ct` and `rt` parameters (see 3.1).

INTERACTIVELY, use `ACEStyle` (see 6.5.22).

12 ► `tw:=[`*val*`,` *word*`]`
► `trace:=[`*val*`,` *word*`]`

Trace *word* through the coset table, starting at coset *val*; *val* must be a positive integer, and *word* must be a word in the group generators.

This option prints the final coset number of the trace, if the trace completes.

INTERACTIVELY, use `ACETraceWord` (see 6.5.17).

D.6 Options that Modify the Coset Table

1 ► `recover`
► `contiguous`

`Recover` space used by dead coset numbers. (Shortest abbreviation of `recover` is `reco`, and shortest abbreviation of `contiguous` is `contig`.)

This option invokes the compaction routine on the table to recover the space used by any dead coset numbers. A `CO` message line is printed if any cosets were recovered, and a `co` line if none were. This routine is called automatically if the `cycles`, `nc`, `print` or `standard` options (see D.5.1, D.5.4, D.5.8 and D.6.2) are invoked.

INTERACTIVELY, use `ACERecover` (see 6.7.1).

2 ► `standard`

Compacts ACE's coset table and standardises the numbering of cosets, according to the `lenlex` scheme (see Section 3.4). (Shortest abbreviation: `st`.)

For a given ordering of the generators in the columns of the table, it produces a canonical numbering of the cosets. This function does not display the new table; use the `print` (see D.5.8) for that. Such a table has the property that a scan of the successive rows of the **body** of the table row by row, from left to right, encounters previously unseen cosets in numeric order.

Notes: In a `lenlex` standard table, the coset representatives are ordered first according to length and then the lexicographic order defined by the order the generators and their inverses head the columns. Note that, since ACE avoids having an involutory generator in the first column when it can, this lexicographic order does not necessarily correspond with the order in which the generators were first put to ACE. Two tables are equivalent only if their canonic forms are the same. Invoking this option directly does **not** affect the GAP coset table obtained via `ACECosetTable`; use the `lenlex` (see 4.11.4) option, if you want your table `lenlex` standardised. (The `lenlex` option restarts the enumeration, if it is necessary to ensure the generators have not been rearranged.)

Guru Notes: In five of the ten standard enumeration strategies of Sims [Sim94] (i.e. the five Sims strategies not provided by ACE), the table is standardised repeatedly. This is expensive computationally, but can result in fewer cosets being necessary. The effect of doing this can be investigated in ACE by (repeatedly) halting the enumeration (via restrictive options), standardising the coset numbering, and continuing (see Section C.2 for an interactive example).

INTERACTIVELY, use `ACEStandardCosetNumbering` (see 6.7.2).

D.7 Options for Comments

1 ► `text:=`*string*

Prints *string* in the output; *string* must be a string.

This allows the user to add comments to the output from ACE.

Note: Please avoid using this option to insert comments starting with three asterisks: `***`, since this string is used as a sentinel internally in flushing output from ACE.

2 ► `aceincomment:=`*string*

Prints comment *string* in the ACE input; *string* must be a string. (Shortest abbreviation: `aceinc`.)

This option prints the comment *string* behind a sharp sign (`#`) in the input to ACE. Only useful for adding comments (that ACE ignores) to standalone input files.

Bibliography

[CDHW73] John J. Cannon, Lucien A. Dimino, George Havas, and Jane M. Watson. Implementation and analysis of the Todd-Coxeter algorithm. *Mathematics of Computation*, 27(123):463–490, July 1973.

[CHHR02] Colin M. Campbell, George Havas, Alexander Hulpke, and Edmund F. Robertson. Efficient simple groups. *Communications in Algebra*, 30(9):4613–4619, 2002.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[CM72] H.S.M. Coxeter and W.O.J. Moser. *Generators and Relations for Discrete Groups*. Springer-Verlag, 3rd edition, 1972.

[Hav91] George Havas. Coset enumeration strategies. In Stephen M. Watt, editor, *ISSAC'91 (Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation)*, pages 191–199. ACM Press, 1991.

[HR99] George Havas and Colin Ramsay. Coset enumeration: ACE version 3, 1999. ACE version 3.001 is available from

`http://staff.itee.uq.edu.au/havas`.

[HR01] George Havas and Colin Ramsay. Groups and computation III. In *Ohio State University Mathematical Research Institute Publications*, volume 8, pages 183–192. de Gruyter, 2001.

[Lee77] John Leech. Computer proof of relations in groups. In Michael P.J. Curran, editor, *Topics in Group Theory and Computation*, pages 38–61. Academic Press, 1977.

[Lee84] John Leech. Coset enumeration. In Michael D. Atkinson, editor, *Computational Group Theory*, pages 3–18. Academic Press, 1984.

[Neu82] J. Neubüser. An elementary introduction to coset table methods in computational group theory. In *Groups – St. Andrews 1981*, London Mathematical Society Lecture Note Series 71, pages 1–45. Cambridge University Press, 1982.

[Ram99] Colin Ramsay. ACE for amateurs (version 3.001). Technical Report 14, Centre for Discrete Mathematics and Computing, The University of Queensland, St. Lucia 4072, Australia, 1999.

[Sim94] Charles C. Sims. *Computation with finitely presented groups*. Cambridge University Press, 1994.

Index

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