# Browse

( Version 1.8.21 )

March 2023

**Thomas Breuer**
**Frank Lübeck**

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

**Frank Lübeck** Email: Frank.Luebeck@Math.RWTH-Aachen.De
Homepage: https://www.math.rwth-aachen.de/~Frank.Luebeck

# Copyright

# Contents

# Chapter 1

# Introduction and Overview

## 1.1 Introduction

The motivation of the package Browse was to provide better functionality for displaying two-dimensional arrays of data (e.g., character tables): moving through the data without loosing row and column labels, searching for text, displaying extra information, hiding information, allowing interactive user input, ...

We wanted to achieve this by using the capabilities of the terminal emulations in which GAP is running, and not by some external graphical user interface. For this we have chosen to use the widely available C-library `ncurses`, see [NCu]. It contains functions to find out terminal capabilities, to change properties of terminals, to place text, to handle several windows with overlapping, ... To use these functions the terminal is switched to a *visual mode* so that the display of the non-visual mode of your terminal in which GAP is running is not clobbered.

Browse has now three levels of functionality:

**A low level interface to `ncurses`**
: This may be interesting for all kinds of applications which want to display text with some markup including colors, maybe in several windows, using the available capabilities of a terminal.

**A medium level interface to a generic function `NCurses.BrowseGeneric` (4.3.1)**
: This is for displaying two-dimensional arrays of data, handles labels for rows and columns, searching, sorting, binding keys to actions, ... If you want to implement such applications for further kinds of data, first look at the examples in Section `BrowseData.IsBrowseTable` (4.2.3), then check what can be copied from the examples in Chapter 6, and consult the descriptions in Chapters 4 and 5.

**Applications of these interfaces**
: We provide some applications of the `ncurses` interface and of the generic `NCurses.BrowseGeneric` (4.3.1) function. These may be interesting for end users, and also as examples for programmers of further applications. This includes (of course) a method for browsing through character tables, functions for browsing through data collections, several games, and an interface for demos.

Users interested only in these applications should perhaps just try `NCurses.Demo()`.

## 1.2 Overview

### 1.2.1 The ncurses interface

Chapter 2 describes GAP's interface to the ncurses C-library. The imported C-functions are shortly explained, but for further details we refer to the documentation of that library. There are also a few utility functions on GAP level, such as NCurses.SetTerm (2.2.2), which simplify the use of the library.

The concept of an *attribute line*, see NCurses.IsAttributeLine (2.2.3), helps to deal with text with markup for its display in a terminal window.

This chapter is for users who want to write their own applications of ncurses.

### 1.2.2 Applications of ncurses

In Chapter 3 we describe some interactive applications of the ncurses interface. For example, there is NCurses.Select (3.1.2) for asking a user to choose one or several of a given list of items. There is also a demo function NCurses.Demo (3.2.1) which we use to demonstrate features of the Browse package, but it may be interesting for other kinds of demos as well.

### 1.2.3 The interface to browse two-dimensional arrays

Chapters 4 and 5 describe the interface to a generic function NCurses.BrowseGeneric (4.3.1) which can be used for an interactive display of two-dimensional arrays of data. The first of these covers the basic functionality which may be sufficient for many applications and the second gives more technical details. With interactive display we mean that it is not only possible to scroll through the data, but one can search for strings, sort by rows or columns, select entries, bind arbitrary actions to keys and mouse events, ask for help, and more.

### 1.2.4 Applications of the generic function NCurses.BrowseGeneric

In Chapter 6 we describe several applications which are using the generic NCurses.BrowseGeneric (4.3.1) interface introduced before. They are provided as prototype applications and so we include some implementation remarks in their documentation.

Users who just want to use these applications hopefully do not need to read this Browse manual, all applications are coming with built-in help windows.

There are different kinds of applications. First, there are methods for browsing through character tables and tables of marks (our original motivation for this package). Then there are applications for browsing through data collections, e.g., the data available through the AtlasRep package, the GAP bibliography or the sections of the GAP manuals. Finally, there are several games like Sam Loyd's fifteen puzzle (generalized), peg solitaire, and Sudoku (including functions to create new puzzles and to solve puzzles).

## 1.3 User preferences provided by the Browse package

See SetUserPreference (**Reference: SetUserPreference**) for GAP's user preferences mechanism, and BrowseUserPreferences (6.10.1) for viewing and modifying user preferences.

### 1.3.1 The user preference `EnableMouseEvents`

This user preference defines whether mouse events are enabled by default in visual mode (value `true`) or not (value `false`, this is the default). During the GAP session, the value can be changed using `NCurses.UseMouse` (2.2.10). Inside browse applications based on `NCurses.BrowseGeneric` (4.3.1) or `NCurses.Select` (3.1.2), the value can be toggled usually by hitting the M key.

### 1.3.2 The user preference `SelectHelpMatches`

In the case that the GAP help system finds multiple matches, `true` (the default) means that the user can choose one entry from a list that is shown via `NCurses.Select` (3.1.2), and `false` means that the matches are just shown in a pager.

### 1.3.3 The user preference `SelectPackageName`

In the case that `LoadPackage` (**Reference: LoadPackage**) is called with a prefix of some package names, `true` (the default) means that the user can choose one matching entry, and `false` means that the matches are just printed.

# Chapter 2

# Interface to the ncurses Library

In this chapter we describe the GAP interface to the GNU curses/ncurses C-library. This library contains routines to manipulate the contents of terminal windows. It allows one to write programs which should work on a wide variety of terminal emulations with different sets of capabilities.

This technical chapter is intended for readers who want to program new applications using the ncurses functionality. If you are only interested in the function NCurses.BrowseGeneric (4.3.1) from this package or some of its applications you can skip this chapter.

Detailed documentation of the ncurses library is probably available in your operating system (try man ncurses) and from the web (see for example [NCu]). Here, we only give short reminders about the functions provided in the GAP interface and explain how to use the GAP functions.

## 2.1 The ncurses Library

In this section we list the functions from the GNU ncurses library and its panel extension which are made available in GAP via the Browse package. See the following section 2.2 for explanations how to use these functions from within GAP.

The basic objects to manipulate are called *windows*, they correspond to rectangular regions of the terminal screen. Windows can overlap but ncurses cannot handle this for the display. Therefore windows can be wrapped in *panels*, they provide a display depth for windows and it is possible to move panels to the top and bottom of the display or to hide a panel.

We will not import all the functions of the ncurses library to GAP. For example, there are many pairs of functions with the same name except for a leading w (like move and wmove for moving the cursor in a window). Here, we only import the versions with w, which get a window as first argument. The functions without w are for the ncurses standard screen window stdscr which is available as window 0 in GAP. Similarly, there are functions with the same name except for an extra n (like waddstr and waddnstr for placing a string into a window). Here, we only import the safer functions with n which get the number of characters to write as argument. (More convenient functions are then implemented on the GAP level.)

### 2.1.1 Setting the terminal

We first list flags for setting the basic behavior of a terminal. With savetty/resetty a setting can be stored and recovered.

`savetty()`
: This stores the current setting of the terminal in a buffer.

`resetty()`
: This resets the terminal to what was stored in the last call to `savetty`.

`cbreak()/nocbreak()`
: In `cbreak` mode each input character from a terminal is directly forwarded to the application (but see `keypad`). With `nocbreak` this only happens after a newline or return is typed.

`keypad(win, bool)`
: If set to `true` some special input like arrow or function keys can be read as single characters from the input (such keys actually generate certain sequences of characters), see also 2.1.4. (The *`win`* argument is irrelevant.)

`echo()/noecho()`
: This determines if input characters are automatically echoed by the terminal at the current cursor position.

`curs_set(vis)`
: This determines the visibility of the cursor. The argument *`vis`*=0 makes the cursor invisible. With *`vis`*=1 it becomes visible; some terminals allow also higher levels of visibility.

`wtimeout(win, delay)`
: Here *`delay`* determines a timeout in milliseconds for reading characters from the input of a window. Negative values mean infinity, that is a blocking read.

`nl()/nonl()`
: With `nl` a return on input is translated to a newline character and a newline on output is interpreted as return and linefeed.

`intrflush(win, bool)`
: This flag determines if after an interrupt pending output to the terminal is flushed. (The *`win`* argument is irrelevant.)

`idlok(win, bool)`
: With `true` the library tries to use a hardware line insertion functionality (in particular for scrolling).

`scrollok(win, bool)`
: If set to `true` moving the cursor down from the last line of a window causes scrolling of the whole window, otherwise nothing happens.

`leaveok(win, bool)`
: If set to `true` a refresh of the window leaves the cursor at its current location, otherwise this is not guaranteed.

`clearok(win, bool)`
: If set to `true` the next refresh of the window will clear the screen completely and redraw everything.

`immedok(win, bool)`
: If set to `true` all changes of the window will automatically also call a `wrefresh`.

`raw()/noraw()`
: Similar to `cbreak`, usually not needed (see the `ncurses` documentation for details).

### 2.1.2 Manipulating windows

In `ncurses` an arbitrary number of windows which correspond to rectangular regions (maybe overlapping) of the screen can be handled. You should always delete windows which are no longer needed. To get a proper display of overlapping windows (which may occur by recursively called functions using this library) we suggest that you always wrap windows in panels, see 2.1.3.

For functions which involve coordinates recall that the upper left corner of the screen or internally of any window has the coordinates (0,0).

`newwin(nlines, ncols, y, x)`
: This creates a new window whose upper left corner has the coordinates ($y$,$x$) on the screen and has *`nlines`* lines and *`ncols`* columns, if this is possible. The arguments *`nlines`* and *`ncols`* can be zero, then their maximal possible values are assumed.

`delwin(win)`
: Deletes a window.

`mvwin(win, y, x)`
: Moves the upper left corner of the window to the given coordinates, if the window still fits on the screen. With panels don't use this function, but use `move_panel` mentioned below.

`wrefresh(win)`
: Writing to a window only changes some internal buffers, this function copies the window content to the actual display screen. You don't need this function if you wrap your windows in panels, use `update_panels` and `doupdate` instead.

`doupdate()`
: Use this function to update the content of your display screen to the current content of all windows. If your terminal is not yet in visual mode this function changes to visual mode.

`endwin()`
: Use this function to leave the visual mode of your terminal. (Remark: If you use this function while not in visual mode the cursor will be moved to the line where the visual mode was started last time. To avoid this use `isendwin` first.)

`isendwin()`
: Returns `true` if called while not in visual mode and `false` otherwise

`getbegyx(win)`
: Get the coordinates of the upper left corner of a window on the screen.

`getmaxyx(win)`
: Get the number of lines and columns of a window.

### 2.1.3 Manipulating panels

Wrap windows in panels to get a proper handling of overlapping windows on the display. Don't forget to delete a panel before deleting the corresponding window.

`new_panel(win)`
: Create a panel for a window.

`del_panel(pan)`
: Delete a panel.

`update_panels()`
: Use this function to copy changes of windows and panels to a screen buffer. Then call `doupdate()` to update the display screen.

`move_panel(pan, y, x)`
: Move top left corner of a panel wrapped window to coordinates ($y$,$x$) if possible.

`hide_panel(pan)`/`show_panel(pan)`
: Hide or show, respectively, the content of a panel on the display screen.

`top_panel(pan)`/`bottom_panel(pan)`
: Move a panel to the top or bottom of all panels, respectively.

`panel_below(pan)`/`panel_above(pan)`
: Return the panel directly below or above the given one, respectively. With argument `0` the top or bottom panel are returned, respectively. If argument is the bottom or top panel, respectively, then `false` is returned.

### 2.1.4 Getting keyboard input

If you want to read input from the user first adjust the terminal settings of `cbreak`, `keypad`, `echo`, `wtimeout` and `curs_set` to your needs, see 2.1.1. The basic functions are as follows.

`wgetch(win)`
: Reads one character from user input (returned as integer). If `wtimeout` was set with a positive *`delay`* then the function returns `false` if there was no input for *`delay`* milliseconds. Note that in `nocbreak` mode typed characters reach the application only after typing a return. If the `keypad` flag is set to `true` some special keys can be read like single characters; the keys are explained below. (Note that there is only one input queue for all windows.)

`ungetch(char)`
: Puts back the character *`char`* on the input queue.

Some terminals allow one to read special keys like one character, we import some of the symbolic names of such keys into GAP. You can check for such characters by comparing with the components of the record `NCurses.keys`, these are

`UP`/`DOWN`/`LEFT`/`RIGHT`
: the arrow keys

`PPAGE/NPAGE`
: the page up and page down keys

`HOME/END`
: the home and end keys

`BACKSPACE/DC`
: the backspace and delete keys

`IC` the insert key

`ENTER`
: the enter key

`F1/F2/../F24`
: the function keys

`MOUSE`
: a pseudo key to detect mouse events

`A1/A3/B2/C1/C3`
: the keys around the arrow keys on a num pad

It can happen that on a specific keyboard there is no key for some of these. Also, not all terminals can interpret all of these keys. You can check this with the function

`has_key(key)`
: Checks if the special key *`key`* is recognized by the terminal.

### 2.1.5 Writing to windows

The display of text in `ncurses` windows has two aspects. The first is to get actual characters on the screen. The second is to specify attributes which influence the display, for example normal or bold fonts or colors. This subsection is for the first aspect. Possible attributes are explained below in 2.1.7.

`wmove(win, y, x)`
: Moves the cursor to position (*`y`*,*`x`*), recall that the coordinates are zero based, (0,0) being the top left corner.

`waddnstr(win, str, len)`
: Writes the string *`str`* to the window starting from the current cursor position. Writes at most *`len`* characters. At end of line the cursor moves to the beginning of next line. The behavior at the end of the window depends on the setting of `scrollok`, see 2.1.1.

`waddch(win, char)`
: Writes a character to the window at the current cursor position and moves the cursor on. The character *`char`* is given as integer and can include attribute information.

`wborder(win, charlist)`
: Draws a border around the window. If *`charlist`* is a plain list of eight GAP characters these are taken for left/right/top/bottom sides and top-left/top-right/bottom-left/bottom-right corners. Otherwise default characters are used. (See `NCurses.WBorder` (2.2.9) for a more user friendly interface.)

`wvline(win, char, len)`
: Writes a vertical line of length *`len`* (or as long as fitting into the window) starting from the current cursor position to the bottom, using the character *`char`*. If *`char`*=0 the default character is used.

`whline(win, char, len)`
: Same as `wvline` but for horizontal lines starting from the cursor position to the right.

`werase(win)`
: Deletes all characters in the window.

`wclear(win)`
: Like `werase`, but also calls `clearok`.

`wclrtobot(win)`
: Deletes all characters from cursor position to the right and bottom.

`wclrtoeol(win)`
: Deletes all characters from cursor position to end of line.

`winch(win)`
: Returns the character at current cursor position, as integer and including color and attribute information.

`getyx(win)`
: Returns the current cursor position.

`waddstr(win, str)`
: Delegates to `waddnstr(win, str, Length(str))`.

### 2.1.6 Line drawing characters

For drawing lines and grids in a terminal window you should use some "virtual" characters which are available as components of the record `NCurses.lineDraw`. On some terminals these are nicely displayed as proper lines (on others they are simulated by ASCII characters). These are:

`BLOCK`
: solid block

`BOARD`
: board of squares

`BTEE/LTEE/RTEE/TTEE`
: bottom/left/right/top tee

`BULLET`
: bullet

`CKBOARD`
: checker board

`DARROW/LARROW/RARROW/UARROW`
: down/left/right/up arrow

`DEGREE`
: degree symbol

`DIAMOND`
: diamond

`GEQUAL`
: greater than or equal

`HLINE/VLINE`
: horizontal/vertical line

`LANTERN`
: lantern symbol

`LEQUAL`
: less than or equal

`LLCORNER/LRCORNER/ULCORNER/URCORNER`
: lower left/lower right/upper left/upper right corner

`NEQUAL`
: not equal

`PI` letter pi

`PLMINUS`
: plus-minus

`PLUS`
: crossing lines like a plus

`S1/S3/S7/S9`
: scan line 1/3/7/9

`STERLING`
: pound sterling

### 2.1.7 Text attributes and colors

In addition to the actual characters to be written to the screen the way they are displayed can be changed by additional *attributes*. (There should be no danger to mix up this notion of attributes with the one introduced in (**Reference: Attributes**).) The available attributes are stored in the record `NCurses.attrs`, they are

`NORMAL`
: normal display with no extra attributes.

`STANDOUT`
: displays text in the best highlighting mode of the terminal.

`UNDERLINE`
: underlines the text.

`REVERSE`
: display in reverse video by exchanging the foreground and background color.

`BLINK`
: displays the text blinking.

`DIM` displays the text half bright.

`BOLD`
: displays the text in a bold font.

Note that not all of these work with all types of terminals, or some may cause the same display. Furthermore, if `NCurses.attrs.has_colors` is `true` there is a list `NCurses.attrs.ColorPairs` of attributes to set the foreground and background color. These should be accessed indirectly with `NCurses.ColorAttr` (2.2.1). Attributes can be combined by adding their values (internally, they are represented by integers). They can also be added to the integer representing a character for use with `waddch`.

The library functions for setting attributes are:

`wattrset(win, attr)`
: This sets the default (combined) attributes for a window which is added to all characters written to it; using `NCurses.attrs.NORMAL` as attribute is a reset.

`wattron(win, attr)`/`wattroff(win, attr)`
: This sets or unsets one or some default attributes of the window without changing the others.

`wattr_get(win)`
: This returns the current default attribute and default color pair of a window.

`wbkgdset(win, attr)`
: This is similar to `wattrset` but you can also add a character to *`attr`* which is used as default instead of blanks.

`wbkgd(win, attr)`
: This function changes the attributes for all characters in the window to *`attr`*, also used for further characters written to that window.

### 2.1.8 Low level `ncurses` mouse support

Many `xterm` based terminals support mouse events. The recognition of mouse events by the `ncurses` input queue can be switched on and off. If switched on and a mouse event occurs, then `NCurses.wgetch` gets `NCurses.keys.MOUSE` if the `keypad` flag is `true` (see 2.1.4). If this is read one must call `NCurses.getmouse` which reads further characters from the input queue and interprets them as details on the mouse event. In most cases the function `NCurses.GetMouseEvent` (2.2.10) can be used in applications (it calls `NCurses.getmouse`). The following low level functions are available as components of the record `NCurses`.

The names of mouse events which may be possible are stored in the list `NCurses.mouseEvents`, which starts `[ "BUTTON1_PRESSED", "BUTTON1_RELEASED", "BUTTON1_CLICKED",`

`"BUTTON1_DOUBLE_CLICKED"`, `"BUTTON1_TRIPLE_CLICKED"`, `...` and contains the same for buttons number 2 to 5 and a few other events.

`mousemask(intlist)`
: The argument *intlist* is a list of integers specifying mouse events. An entry `i` refers to the event described in `NCurses.mouseEvents[i+1]`. It returns a record with components `.new` (for the current setting) and `.old` (for the previous setting) which are again lists of integers with the same meaning. Note that `.new` may be different from *intlist*, it is always the empty list if the terminal does not support mouse events. In applications use `NCurses.UseMouse` (2.2.10) instead of this low level function.

`getmouse()`
: This function must be called after a key `NCurses.keys.MOUSE` was read. It returns a list with three entries `[y, x, intlist]` where `y` and `x` are the coordinates of the character cell where the mouse event occured and `intlist` describes the event, it should have length one and refers to a position in `NCurses.mouseEvents`.

`wenclose(win, y, x)`
: This functions returns `true` if the screen position *y*, *x* is within window *win* and `false` otherwise.

`mouseinterval(t)`
: Sets the time to recognize a press and release of a mouse button as a click to *t* milliseconds. (Note that this may have no effect because a window manager may catch this.)

### 2.1.9 Miscellaneous function

We also provide the `ncurses` function `mnap(msec)` which is a sleep for *msec* milliseconds.

Furthermore, there a two utilities which can be useful for scripts and testing, namely a check if standard input or standard output are connected to terminals. These can be called as `NCurses.IsStdinATty()` or `NCurses.IsStdoutATty()`, respectively.

## 2.2 The `ncurses` GAP functions

The functions of the `ncurses` library are used within GAP very similarly to their `C` equivalents. The functions are available as components of a record `NCurses` with the name of the `C` function (e.g., `NCurses.newwin`).

In GAP the `ncurses` windows are accessed via integers (as returned by `NCurses.newwin`). The standard screen `stdscr` from the `ncurses` library is available as window number `0`. But this should not be used; to allow recursive applications of `ncurses` always create a new window, wrap it in a panel and delete both when they are no longer needed.

Each window can be wrapped in one panel which is accessed by the same integer. (Window `0` cannot be used with a panel.)

Coordinates in windows are the same zero based integers as in the corresponding `C` functions. The interface of functions which *return* coordinates is slightly different from the `C` version; they just return lists of integers and you just give the window as argument, e.g., `NCurses.getmaxyx(win)` returns a list `[nrows, ncols]` of two integers.

Characters to be written to a window can be given either as GAP characters like `'a'` or as integers like `INT_CHAR('a') = 97`. If you use the integer version you can also add attributes including color settings to it for use with `NCurses.waddch`.

When writing an application decide about an appropriate terminal setting for your visual mode windows, see 2.1.1 and the utility function `NCurses.SetTerm` (2.2.2) below. Use `NCurses.savetty()` and `NCurses.resetty()` to save and restore the previous setting.

We also provide some higher level functionality for displaying marked up text, see `NCurses.PutLine` (2.2.6) and `NCurses.IsAttributeLine` (2.2.3).

We now describe some utility functions for putting text on a terminal window.

### 2.2.1 NCurses.ColorAttr

▷ `NCurses.ColorAttr(`*`fgcolor, bgcolor`*`)` (function)

**Returns:** an attribute for setting the foreground and background color to be used on a terminal window (it is a GAP integer).

▷ `NCurses.attrs.has_colors` (global variable)

The return value can be used like any other attribute as described in 2.1.7. The arguments *`fgcolor`* and *`bgcolor`* can be given as strings, allowed are those in `[ "black", "red", "green", "yellow", "blue", "magenta", "cyan", "white" ]`. These are the default foreground colors 0 to 7 on ANSI terminals. Alternatively, the numbers 0 to 7 can be used directly as arguments.

Note that terminals can be configured in a way such that these named colors are not the colors which are actually displayed.

The variable `NCurses.attrs.has_colors` (2.2.1) is set to `true` or `false` if the terminal supports colors or not, respectively. If a terminal does not support colors then `NCurses.ColorAttr` always returns `NCurses.attrs.NORMAL`.

For an attribute setting the foreground color with the default background color of the terminal use `-1` as *`bgcolor`* or the same as *`fgcolor`*.

Example

```
gap> win := NCurses.newwin(0,0,0,0);; pan := NCurses.new_panel(win);;
gap> defc := NCurses.defaultColors;;
gap> NCurses.wmove(win, 0, 0);;
gap> for a in defc do for b in defc do
>      NCurses.wattrset(win, NCurses.ColorAttr(a, b));
>      NCurses.waddstr(win, Concatenation(a,"/",b,"\t"));
>    od; od;
gap> if NCurses.IsStdoutATty() then
>      NCurses.update_panels();; NCurses.doupdate();;
>      NCurses.napms(5000);;     # show for 5 seconds
>      NCurses.endwin();; NCurses.del_panel(pan);; NCurses.delwin(win);;
>    fi;
```

### 2.2.2 NCurses.SetTerm

▷ `NCurses.SetTerm(`*`[record]`*`)` (function)

This function provides a unified interface to the various terminal setting functions of `ncurses` listed in 2.1.1. The optional argument is a record with components which are assigned to `true` or

`false`. Recognised components are: `cbreak`, `echo`, `nl`, `intrflush`, `leaveok`, `scrollok`, `keypad`, `raw` (with the obvious meaning if set to `true` or `false`, respectively).

The default, if no argument is given, is `rec(cbreak := true, echo := false, nl := false, intrflush := false, leaveok := true, scrollok := false, keypad := true)`. (This is a useful setting for many applications.) If there is an argument *record*, then the given components overwrite the corresponding defaults.

### 2.2.3 NCurses.IsAttributeLine

▷ `NCurses.IsAttributeLine(`*obj*`)` (function)

**Returns:** `true` if the argument describes a string with attributes.

An *attribute line* describes a string with attributes. It is represented by either a string or a dense list of strings, integers, and Booleans immediately following integers, where at least one list entry must *not* be a string. (The reason is that we want to be able to distinguish between an attribute line and a list of such lines, and that the case of plain strings is perhaps the most usual one, so we do not want to force wrapping each string in a list.) The integers denote attribute values such as color or font information, the Booleans denote that the attribute given by the preceding integer is set or reset.

If an integer is not followed by a Boolean then it is used as the attribute for the following characters, that is it overwrites all previously set attributes. Note that in some applications the variant with explicit Boolean values is preferable, because such a line can nicely be highlighted just by prepending a `NCurses.attrs.STANDOUT` attribute.

For an overview of attributes, see 2.1.7.

Example

```
gap> NCurses.IsAttributeLine( "abc" );
true
gap> NCurses.IsAttributeLine( [ "abc", "def" ] );
false
gap> NCurses.IsAttributeLine( [ NCurses.attrs.UNDERLINE, true, "abc" ] );
true
gap> NCurses.IsAttributeLine( "" );  NCurses.IsAttributeLine( [] );
true
false
```

The *empty string* is an attribute line whereas the *empty list* (which is not in `IsStringRep` (**Reference: IsStringRep**)) is *not* an attribute line.

### 2.2.4 NCurses.ConcatenationAttributeLines

▷ `NCurses.ConcatenationAttributeLines(`*lines*`[, `*keep*`])` (function)

**Returns:** an attribute line.

For a list *lines* of attribute lines (see `NCurses.IsAttributeLine` (2.2.3)), `NCurses.ConcatenationAttributeLines` returns the attribute line obtained by concatenating the attribute lines in *lines*.

If the optional argument *keep* is `true` then attributes set in an entry of *lines* are valid also for the following entries of *lines*. Otherwise (in particular if there is no second argument) the attributes are reset to `NCurses.attrs.NORMAL` between the entries of *lines*.

Example

```
gap> plain_str:= "hello";;
gap> with_attr:= [ NCurses.attrs.BOLD, "bold" ];;
```

```
gap> NCurses.ConcatenationAttributeLines( [ plain_str, plain_str ] );
"hellohello"
gap> NCurses.ConcatenationAttributeLines( [ plain_str, with_attr ] );
[ "hello", 2097152, "bold" ]
gap> NCurses.ConcatenationAttributeLines( [ with_attr, plain_str ] );
[ 2097152, "bold", 0, "hello" ]
gap> NCurses.ConcatenationAttributeLines( [ with_attr, with_attr ] );
[ 2097152, "bold", 0, 2097152, "bold" ]
gap> NCurses.ConcatenationAttributeLines( [ with_attr, with_attr ], true );
[ 2097152, "bold", 2097152, "bold" ]
```

### 2.2.5 NCurses.RepeatedAttributeLine

▷ `NCurses.RepeatedAttributeLine(`*`line, width`*`)` (function)

**Returns:** an attribute line.

For an attribute line *`line`* (see `NCurses.IsAttributeLine` (2.2.3)) and a positive integer *`width`*, `NCurses.RepeatedAttributeLine` returns an attribute line with *`width`* displayed characters (see `NCurses.WidthAttributeLine` (2.2.7)) that is obtained by concatenating sufficiently many copies of *`line`* and cutting off a tail if applicable.

Example

```
gap> NCurses.RepeatedAttributeLine( "12345", 23 );
"12345123451234512345123"
gap> NCurses.RepeatedAttributeLine( [ NCurses.attrs.BOLD, "12345" ], 13 );
[ 2097152, "12345", 0, 2097152, "12345", 0, 2097152, "123" ]
```

### 2.2.6 NCurses.PutLine

▷ `NCurses.PutLine(`*`win, y, x, lines[, skip]`*`)` (function)

**Returns:** `true` if *`lines`* were written, otherwise `false`.

The argument *`lines`* can be a list of attribute lines (see `NCurses.IsAttributeLine` (2.2.3)) or a single attribute line. This function writes the attribute lines to window *`win`* at and below of position *`y`*, *`x`*.

If the argument *`skip`* is given, it must be a nonnegative integer. In that case the first *`skip`* characters of each given line are not written to the window (but the attributes are).

### 2.2.7 NCurses.WidthAttributeLine

▷ `NCurses.WidthAttributeLine(`*`line`*`)` (function)

**Returns:** number of displayed characters in an attribute line.

For an attribute line *`line`* (see `NCurses.IsAttributeLine` (2.2.3)), the function returns the number of displayed characters of *`line`*.

Example

```
gap> NCurses.WidthAttributeLine( "abcde" );
5
gap> NCurses.WidthAttributeLine( [ NCurses.attrs.BOLD, "abc",
>        NCurses.attrs.NORMAL, "de" ] );
5
```

### 2.2.8 NCurses.Grid

▷ `NCurses.Grid(`*win, trow, brow, lcol, rcol, rowinds, colinds*`)` (function)

This function draws a grid of horizontal and vertical lines on the window *win*, using the line drawing characters explained in 2.1.6. The given arguments specify the top and bottom row of the grid, its left and right column, and lists of row and column numbers where lines should be drawn.

Example

```
gap> fun := function() local win, pan;
>       win := NCurses.newwin(0,0,0,0);
>       pan := NCurses.new_panel(win);
>       NCurses.Grid(win, 2, 11, 5, 22, [5, 6], [13, 14]);
>       NCurses.PutLine(win, 12, 0, "Press <Enter> to quit");
>       NCurses.update_panels(); NCurses.doupdate();
>       NCurses.wgetch(win);
>       NCurses.endwin();
>       NCurses.del_panel(pan); NCurses.delwin(win);
> end;;
gap> fun();
```

### 2.2.9 NCurses.WBorder

▷ `NCurses.WBorder(`*win[, chars]*`)` (function)

This is a convenient interface to the `ncurses` function `wborder`. It draws a border around the window *win*. If no second argument is given the default line drawing characters are used, see 2.1.6. Otherwise, *chars* must be a list of GAP characters or integers specifying characters, possibly with attributes. If *chars* has length 8 the characters are used for the left/right/top/bottom sides and top-left/top-right/bottom-left/bottom-right corners. If *chars* contains 2 characters the first is used for the sides and the second for all corners. If *chars* contains just one character it is used for all sides including the corners.

### 2.2.10 Mouse support in `ncurses` applications

▷ `NCurses.UseMouse(`*on*`)` (function)

**Returns:** a record

▷ `NCurses.GetMouseEvent()` (function)

**Returns:** a list of records

`ncurses` allows on some terminals (`xterm` and related) to catch mouse events. In principle a subset of events can be caught, see `mousemask` in 2.1.8. But this does not seem to work well with proper subsets of possible events (probably due to intermediate processes X, window manager, terminal application, ...). Therefore we suggest to catch either all or no mouse events in applications.

This can be done with `NCurses.UseMouse` with argument `true` to switch on the recognition of mouse events and `false` to switch it off. The function returns a record with components `.new` and `.old` which are both set to the status `true` or `false` from after and before the call, respectively. (There does not seem to be a possibility to get the current status without calling `NCurses.UseMouse`.) If you call the function with argument `true` and the `.new` component of the result is `false`, then the terminal does not support mouse events.

When the recognition of mouse events is switched on and a mouse event occurs then the key `NCurses.keys.MOUSE` is found in the input queue, see `wgetch` in 2.1.4. If this key is read the low level function `NCurses.getmouse` must be called to fetch further details about the event from the input queue, see 2.1.8. In many cases this can be done by calling the function `NCurses.GetMouseEvent` which also generates additional information. The return value is a list of records, one for each panel over which the event occured, these panels sorted from top to bottom (so, often you will just need the first entry if there is any). Each of these records has components `.win`, the corresponding window of the panel, `.y` and `.x`, the relative coordinates in window `.win` where the event occured, and `.event`, which is bound to one of the strings in `NCurses.mouseEvents` which describes the event.

*Suggestion:* Always make the use of the mouse optional in your application. Allow the user to switch mouse usage on and off while your application is running. Some users may not like to give mouse control to your application, for example the standard cut and paste functionality cannot be used while mouse events are caught.

### 2.2.11 NCurses.SaveWin

▷ `NCurses.SaveWin(`*`win`*`)` (function)
▷ `NCurses.StringsSaveWin(`*`cont`*`)` (function)
▷ `NCurses.RestoreWin(`*`win, cont`*`)` (function)
▷ `NCurses.ShowSaveWin(`*`cont`*`)` (function)

**Returns:** a GAP object describing the contents of a window.

These functions can be used to save and restore the contents of `ncurses` windows. `NCurses.SaveWin` returns a list `[nrows, ncols, chars]` giving the number of rows, number of columns, and a list of integers describing the content of window *`win`*. The integers in the latter contain the displayed characters plus the attributes for the display.

The function `NCurses.StringsSaveWin` translates data *`cont`* in form of the output of `NCurses.SaveWin` to a list of `nrows` strings giving the text of the rows of the saved window, and ignoring the attributes. You can view the result with `NCurses.Pager` (3.1.4).

The argument *`cont`* for `NCurses.RestoreWin` must be of the same format as the output of `NCurses.SaveWin`. The content of the saved window is copied to the window *`win`*, starting from the top-left corner as much as it fits.

The utility `NCurses.ShowSaveWin` can be used to display the output of `NCurses.SaveWin` (as much of the top-left corner as fits on the screen).

# Chapter 3

# Utilities using ncurses

In this chapter we describe the usage of some example applications of the ncurses interface provided by the Browse package. They may be of interest by themselves, or they may be used as utility functions within larger applications.

## 3.1 ncurses utilities

If you call the functions NCurses.Alert (3.1.1), NCurses.Select (3.1.2), NCurses.GetLineFromUser (3.1.3), or NCurses.Pager (3.1.4) from another ncurses application in visual mode, you should refresh the windows that are still open, by calling NCurses.update_panels and NCurses.doupdate afterwards, see Section 2.1.3 and 2.1.2. Also, if the cursor shall be hidden after that, you should call curs_set with argument 0, see Section 2.1.1, since the cursor is automatically made visible in NCurses.endwin.

### 3.1.1 NCurses.Alert

▷ NCurses.Alert(*messages, timeout[, attrs]*) (function)

**Returns:** the integer corresponding to the character entered, or fail.

In visual mode, Print (**Reference: Print**) cannot be used for messages. An alternative is given by NCurses.Alert.

Let *messages* be either an attribute line or a nonempty list of attribute lines, and *timeout* be a nonnegative integer. NCurses.Alert shows *messages* in a bordered box in the middle of the screen.

If *timeout* is zero then the box is closed after any user input, and the integer corresponding to the entered key is returned. If *timeout* is a positive number $n$, say, then the box is closed after $n$ milliseconds, and fail is returned.

If timeout is zero and mouse events are enabled (see NCurses.UseMouse (2.2.10)) then the box can be moved inside the window via mouse events.

If the optional argument *attrs* is given, it must be an integer representing attributes such as the components of NCurses.attrs (see Section 2.1.7) or the return value of NCurses.ColorAttr (2.2.1); these attributes are used for the border of the box. The default is NCurses.attrs.NORMAL.

Example

```
gap> NCurses.Alert( "Hello world!", 1000 );
fail
gap> NCurses.Alert( [ "Hello world!",
>      [ "Hello ", NCurses.attrs.BOLD, "bold!" ] ], 1500,
```

```
> NCurses.ColorAttr( "red", -1 ) + NCurses.attrs.BOLD );
fail
```

### 3.1.2 NCurses.Select

▷ `NCurses.Select(`*`poss[, single[, none]]`*`)` (function)

**Returns:** Position or list of positions, or `false`.

This function allows the user to select one or several items from a given list. In the simplest case *`poss`* is a list of attribute lines (see `NCurses.IsAttributeLine` (2.2.3)), each of which should fit on one line. Then `NCurses.Select` displays these lines and lets the user browse through them. After pressing the RETURN key the index of the highlighted item is returned. Note that attributes in your lines should be switched on and off separately by `true`/`false` entries such that the lines can be nicely highlighted.

The optional argument *`single`* must be `true` (default) or `false`. In the second case, an arbitrary number of items can be marked and the function returns the list of their indices.

If *`single`* is `true` a third argument *`none`* can be given. If it is `true` then it is possible to leave the selection without choosing an item, in this case `false` is returned.

More details can be given to the function by giving a record as argument *`poss`*. It can have the following components:

`items`
: The list of attribute lines as described above.

`single`
: Boolean with the same meaning as the optional argument *`single`*.

`none`
: Boolean with the same meaning as the optional argument *`none`*.

`size`
: The size of the window like the first two arguments of `NCurses.newwin` (default is `[0, 0]`, as big as possible), or the string `"fit"` which means the smallest possible window.

`align`
: A substring of `"bclt"`, which describes the alignment of the window in the terminal. The meaning and the default are the same as for `BrowseData.IsBrowseTableCellData` (4.2.1).

`begin`
: Top-left corner of the window like the last two arguments of `NCurses.newwin` (default is `[0, 0]`, top-left of the screen). This value has priority over the `align` component.

`attribute`
: An attribute used for the display of the window (default is `NCurses.attrs.NORMAL`).

`border`
: If the window should be displayed with a border then set to `true` (default is `false`) or to an integer representing attributes such as the components of `NCurses.attrs` (see Section 2.1.7) or the return value of `NCurses.ColorAttr` (2.2.1); these attributes are used for the border of the box. The default is `NCurses.attrs.NORMAL`.

`header`
: An attribute line used as header line (the default depends on the settings of `single` and `none`).

`hint`
: An attribute line used as hint in the last line of the window (the default depends on the settings of `single` and `none`).

`onSubmitFunction`
: A function that is called when the user submits the selection; the argument for this call is the current value of the record *poss*. If the function returns `true` then the selected entries are returned as usual, otherwise the selection window is kept open, waiting for new inputs; if the function returns a nonempty list of attribute lines then these messages are shown using `NCurses.Alert` (3.1.1).

If mouse events are enabled (see `NCurses.UseMouse` (2.2.10)) then the window can be moved on the screen via mouse events, the focus can be moved to an entry, and (if `single` is `false`) the selection of an entry can be toggled.

Example

```
gap> index := NCurses.Select(["Apples", "Pears", "Oranges"]);
gap> index := NCurses.Select(rec(
>                     items := ["Apples", "Pears", "Oranges"],
>                     single := false,
>                     border := true,
>                     begin := [5, 5],
>                     size := [8, 60],
>                     header := "Choose at least two fruits",
>                     attribute := NCurses.ColorAttr("yellow","red"),
>                     onSubmitFunction:= function( r )
>                       if Length( r.RESULT ) < 2 then
>                         return [ "Choose at least two fruits" ];
>                       else
>                         return true;
>                       fi;
>                     end ) );
```

### 3.1.3 NCurses.GetLineFromUser

▷ `NCurses.GetLineFromUser(`*pre*`)` (function)

**Returns:** User input as string.

This function can be used to get an input string from the user. It opens a one line window and writes the given string *pre* into it. Then it waits for user input. After hitting the RETURN key the typed line is returned as a string to GAP. If the user exits via hitting the ESC key instead of hitting the RETURN key, the function returns `false`. (The ESC key may be recognized as input only after a delay of about a second.)

Some simple editing is possible during user input: The LEFT, RIGHT, HOME and END keys, the INSERT/REPLACE keys, and the BACKSPACE/DELETE keys are supported.

Instead of a string, *pre* can also be a record with the component `prefix`, whose value is the string described above. The following optional components of this record are supported.

`window`
: The window with the input field is created relative to this window, the default is 0.

`begin`
: This is a list with the coordinates of the upper left corner of the window with the input field, relative to the window described by the `window` component; the default is `[ y-4, 2 ]`, where `y` is the height of this window.

`default`
: This string appears as result when the window is opened, the default is an empty string.

Example

```
gap> str := NCurses.GetLineFromUser("Your Name: ");;
gap> Print("Hello ", str, "!\n");
```

### 3.1.4 NCurses.Pager

▷ `NCurses.Pager(`*`lines[, border[, ly, lx, y, x]]`*`)` (function)

This is a simple pager utility for displaying and scrolling text. The argument *`lines`* can be a list of attribute lines (see `NCurses.IsAttributeLine` (2.2.3)) or a string (the lines are separated by newline characters) or a record. In case of a record the following components are recognized:

`lines`
: The list of attribute lines or a string as described above.

`start`
: Line number to start the display.

`size`
: The size `[ly, lx]` of the window like the first two arguments of `NCurses.newwin` (default is `[0, 0]`, as big as possible).

`begin`
: Top-left corner `[y, x]` of the window like the last two arguments of `NCurses.newwin` (default is `[0, 0]`, top-left of the screen).

`attribute`
: An attribute used for the display of the window (default is `NCurses.attrs.NORMAL`).

`border`
: Either one of `true`/`false` to show the pager window with or without a standard border. Or it can be string with eight, two or one characters, giving characters to be used for a border, see `NCurses.WBorder` (2.2.9).

`hint`
: A text for usage info in the last line of the window.

As an abbreviation the information from `border`, `size` and `begin` can also be specified in optional arguments.

Example

```
gap> lines := List([1..100],i-> ["line ",NCurses.attrs.BOLD,String(i)]);;
gap> NCurses.Pager(lines);
```

### 3.1.5 Selection of help matches

After loading the Browse package GAP's help system behaves slightly different when a request yields several matches. The matches are shown via `NCurses.Select` (3.1.2), the list can be searched and filtered, and one can choose one match for immediate display. It is possible to not choose a match and the `?<nr>` syntax still works.

If you want the original behavior call `SetUserPreference( "Browse", "SelectHelpMatches", false );` in your GAP session or `gap.ini` file, see (**Reference: Configuring User preferences**).

### 3.1.6 Selection of package names

The function `LoadPackage` (**Reference: LoadPackage**) shows a list of matches if only a prefix of a package name is given. After loading the Browse package, `NCurses.Select` (3.1.2) is used for that, and one can choose a match.

If you want the original behavior call `SetUserPreference( "Browse", "SelectPackageName", false );` in your GAP session or `gap.ini` file, see (**Reference: Configuring User preferences**).

## 3.2 A Demo Function

### 3.2.1 NCurses.Demo

▷ `NCurses.Demo(`*`[inputs]`*`)` (function)

Let *`inputs`* be a list of records, each with the components `title` (a string), `inputblocks` (a list of strings, each describing some GAP statements), and optionally `footer` (a string) and `cleanup` (a string describing GAP statements). The default is `NCurses.DemoDefaults`.

`NCurses.Demo` lets the user choose an entry from *`inputs`*, via `NCurses.Select` (3.1.2), and then executes the GAP statements in the first entry of the `inputblocks` list of this entry; these strings, together with the values of `title` and `footer`, are shown in a window, at the bottom of the screen. The effects of calls to functions using `ncurses` are shown in the rest of the screen. After the execution of the statements (which may require user input), the user can continue with the next entry of `inputblocks`, or return to the `inputs` selection (and thus cancel the current `inputs` entry), or return to the execution of the beginning of the current `inputs` entry. At the end of the current entry of `inputs`, the user returns to the `inputs` selection.

The GAP statements in the `cleanup` component, if available, are executed whenever the user does not continue; this is needed for deleting panels and windows that are defined in the statements of the current entry.

Note that the GAP statements are executed in the *global* scope, that is, they have the same effect as if they would be entered at the GAP prompt. Initially, `NCurses.Demo` sets the value of `BrowseData.defaults.work.windowParameters` to the parameters that describe the part of the screen above the window that shows the inputs; so applications of `NCurses.BrowseGeneric` (4.3.1) use automatically the maximal part of the screen as their window. It is recommended to use a screen with at least 80 columns and at least 37 rows.

# Chapter 4

# Browsing Tables in GAP using ncurses –The User Interface

As stated in Section 1.1, one aim of the Browse package is to provide tools for the quite usual task to show a two-dimensional array or certain rows and columns of it on a character screen in a formatted way, to navigate in this array via key strokes (and mouse events), and to search for strings, to sort the array by row or column values etc.

The idea is that one starts with an array of data, the *main table*. Optionally, labels for each row of the main table are given, which are also arranged in an array (with perhaps several columns), the *row labels table*; analogously, a *column labels table* of labels for the columns of the main table may be given. The row labels are shown to the left of the main table, the column labels are shown above the main table. The space above the row labels and to the left of the column labels can be used for a fourth table, the *corner table*, with information about the labels or about the main table. Optionally, a *header* and a *footer* may be shown above and below these four tables, respectively. Header and footer are not separated into columns. So the shown window has the following structure.

<table>
<tr><td colspan="2">header</td></tr>
<tr><td>corner</td><td>column labels</td></tr>
<tr><td>row<br>labels</td><td>main<br>table</td></tr>
<tr><td colspan="2">footer</td></tr>
</table>

If not the whole tables fit into the window then only subranges of rows and columns of the main table are shown, together with the corresponding row and column labels. Also in this case, the row heights and column widths are computed w.r.t. the whole table not w.r.t. the shown rows and columns. This means that the shown row labels are unchanged if the range of shown columns is changed, the shown column labels are unchanged if the range of shown rows is changed, and the whole corner table is always shown.

The current chapter describes the user interface for *standard applications* of this kind, i. e., those applications for which one just has to collect the data to be shown in a record –which we call a *browse table*– without need for additional GAP programming.

Section 4.1 gives an overview of the features available in standard browse table applications, and Section 4.2 describes the data structures used in browse tables. Finally, Section 4.3 introduces the

function `NCurses.BrowseGeneric` (4.3.1), which is the generic function for showing browse table in visual mode.

For technical details needed to extend these applications and to build other applications, see Chapter 5.

Examples of browse table applications are shown in Chapter 6.

## 4.1 Features Supported by the Function `NCurses.BrowseGeneric`

Standard applications of the function `NCurses.BrowseGeneric` (4.3.1) have the following functionality. Other applications may provide only a subset, or add further functionality, see Chapters 5 and 6.

**Scrolling:**
The subranges of shown rows and columns of the main table can be modified, such that the focus area is moved to the left, to the right, up, or down; depending on the context, the focus is moved by one character, by one table cell or a part of it, by the window height/width (minus one character or minus one table cell). If mouse events are enabled then cells can be selected also via mouse clicks.

**Selecting:**
A cell, row, or column of the main table can be selected; then it is shown highlighted on the screen (by default using the attribute `NCurses.attrs.STANDOUT`, see Section 2.1.7). The selection can be moved inside the main table to a neighboring cell, row, or column; this causes also scrolling of the main table when the window borders are reached.

**Searching:**
A search string is entered by the user, and the first matching cell becomes selected; one can search further for the next matching cell. Global search parameters define what matching means (case sensitive or not, search for substrings or complete words) and what the first and the next matching cells are (search in the whole table or just in the selected row or column, search for whole words or prefixes or suffixes, search forwards or backwards).

**Sorting:**
If a row or column is selected then the main table can be sorted w.r.t. the entries in this row or column. Global sort parameters describe for example whether one wants to sort ascending or descending, or case sensitive or not.

If a categorized table is sorted by a column then the category rows are removed and the current sorting and filtering by rows is reset before the table is sorted by the given column. If a table is sorted by a column/row that is already sorted by a column/row then this ordering is reset first.

Sorting can be undone globally, i. e., one can return to the unsorted table.

**Sorting and Categorizing:**
If a column is selected then the main table can be sorted w.r.t. the entries in this column, and additionally these entries are turned into *category rows*, i. e., additional rows are added to the main table, appearing immediately above the table rows with a fixed value in the selected column, and showing this column value. (There should be no danger to mix up this notion of categories with the one introduced in (**Reference: Categories**).) The category rows can be *collapsed* (that is, the table rows that belong to this category row are not shown) or *expanded*

(that is, the corresponding table rows are shown). Some of the global search parameters affect the category rows, for example, whether the category rows shall involve a counter showing the number of corresponding data rows, or whether a row of the browse table appears under different category rows.

Sorting and categorizing can be undone globally, i. e., one can return to the unsorted table without category rows.

**Filtering:**

The browse table can be restricted to those rows or columns in which a given search string occurs. (Also entries in collapsed rows can match; they remain collapsed then.) As a consequence, the category rows are restricted to those under which a matching row occurs. (It is irrelevant whether the search string occurs in category rows.)

If the search string does not occur at all then a message is printed, and the table remains as it was before. If a browse table is restricted then this fact is indicated by the message "restricted table" in the lower right corner of the window.

When a column or row is selected then the search is restricted to the entries in this column or row, respectively. Besides using a search, one can also explicitly hide the selected row or column. Filtering in an already restricted table restricts the shown rows or columns further.

Filtering can be undone globally, i. e., one can return to the unrestricted table.

**Help:**

Depending on the application and on the situation, different sets of user inputs may be available and different meanings of these inputs are possible. An overview of the currently available inputs and their meanings can be opened in each situation, by hitting the ? key.

**Re-entering:**

When one has called `NCurses.BrowseGeneric` (4.3.1) with a browse table, and returns from visual mode to the GAP prompt after some navigation steps, calling `NCurses.BrowseGeneric` again with this table will enter visual mode in the same situation where it was left. For example, the cell in the top-left position will be the same as before, and if a cell was selected before then this cell will be selected now. (One can avoid this behavior using the optional second argument of `NCurses.BrowseGeneric`.)

**Logging:**

The integers corresponding to the user inputs in visual mode are collected in a list that is stored in the component `dynamic.log` of the browse table. It can be used for repeating the inputs with the replay feature. (For browse table applications that give the user no access to the browse table itself, one can force the log to be assigned to the component `log` of the global variable `BrowseData`, see Section 5.4.1.)

**Replay:**

Instead of interactively hitting keys in visual mode, one can prescribe the user inputs to a browse table via a "replay record"; the inputs are then processed with given time intervals. The easiest way to create a meaningful replay record is via the function `BrowseData.SetReplay` (5.4.2), with argument the `dynamic.log` component of the browse table in question that was stored in an interactive session.

The following features are not available in standard applications. They require additional programming.

**Clicking:**

One possible action is to "click" a selected cell, row, or column, by hitting the ENTER key. It depends on the application what the effect is. A typical situation is that a corresponding GAP object is added to the list of return values of `NCurses.BrowseGeneric` (4.3.1). Again it depends on the application what this GAP object is. In order to use this feature, one has to provide a record whose components are GAP functions, see Section 5.4.1 for details. If mouse events are enabled (see `NCurses.UseMouse` (2.2.10)) then also mouse clicks can be used as an alternative to hitting the ENTER key.

**Return Value:**

The function `NCurses.BrowseGeneric` (4.3.1) may have an application dependent return value. A typical situation is that a list of objects corresponding to those cells is returned that were "clicked" in visual mode. In order to use this feature, one has to assign the desired value to the component `dynamic.Return` of the browse table.

## 4.2 Data Structures used by `NCurses.BrowseGeneric`

### 4.2.1 BrowseData.IsBrowseTableCellData

▷ `BrowseData.IsBrowseTableCellData(`*obj*`)` (function)

**Returns:** `true` if the argument is a list or a record in a supported format.

A *table cell data object* describes the input data for the contents of a cell in a browse table. It is represented by either an attribute line (see `NCurses.IsAttributeLine` (2.2.3)), for cells of height one, or a list of attribute lines or a record with the components `rows`, a list of attribute lines, and optionally `align`, a substring of `"bclt"`, which describes the alignment of the attribute lines in the table cell -- bottom, horizontally centered, left, and top alignment; the default is right and vertically centered alignment. (Note that the height of a table row and the width of a table column can be larger than the height and width of an individual cell.)

Example

```
gap> BrowseData.IsBrowseTableCellData( "abc" );
true
gap> BrowseData.IsBrowseTableCellData( [ "abc", "def" ] );
true
gap> BrowseData.IsBrowseTableCellData( rec( rows:= [ "ab", "cd" ],
>                                           align:= "tl" ) );
true
gap> BrowseData.IsBrowseTableCellData( "" );
true
gap> BrowseData.IsBrowseTableCellData( [] );
true
```

The *empty string* is a table cell data object of height one and width zero whereas the *empty list* (which is not in `IsStringRep` (**Reference: IsStringRep**)) is a table cell data object of height zero and width zero.

### 4.2.2 BrowseData.BlockEntry

▷ `BrowseData.BlockEntry(`*`tablecelldata, height, width`*`)` (function)

**Returns:** a list of attribute lines.

For a table cell data object *tablecelldata* (see `BrowseData.IsBrowseTableCellData` (4.2.1)) and two positive integers *height* and *width*, `BrowseData.BlockEntry` returns a list of *height* attribute lines of displayed length *width* each (see `NCurses.WidthAttributeLine` (2.2.7)), that represents the formatted version of *tablecelldata*.

If the rows of *tablecelldata* have different numbers of displayed characters then they are filled up to the desired numbers of rows and columns, according to the alignment prescribed by *tablecelldata*; the default alignment is right and vertically centered.

Example

```
gap> BrowseData.BlockEntry( "abc", 3, 5 );
[ "     ", "  abc", "     " ]
gap> BrowseData.BlockEntry( rec( rows:= [ "ab", "cd" ],
>                                align:= "tl" ), 3, 5 );
[ "ab   ", "cd   ", "     " ]
```

### 4.2.3 BrowseData.IsBrowseTable

▷ `BrowseData.IsBrowseTable(`*`obj`*`)` (function)

**Returns:** `true` if the argument record has the required components and is consistent.

A *browse table* is a GAP record that can be used as the first argument of the function `NCurses.BrowseGeneric` (4.3.1).

The supported components of a browse table are `work` and `dynamic`, their values must be records: The components in `work` describe that part of the data that are not likely to depend on user interactions, such as the table entries and their heights and widths. The components in `dynamic` describe that part of the data that is intended to change with user interactions, such as the currently shown top-left entry of the table, or the current status w.r.t. sorting. For example, suppose you call `NCurses.BrowseGeneric` (4.3.1) twice with the same browse table; the second call enters the table in the same status where it was left *after* the first call if the component `dynamic` is kept, whereas one has to reset (which usually simply means to unbind) the component `dynamic` if one wants to start in the same status as *before* the first call.

The following components are the most important ones for standard browse applications. All these components belong to the `work` record. For other supported components (of `work` as well as of `dynamic`) and for the meaning of the term "mode", see Section 5.2.

`main`
: is the list of lists of table cell data objects that form the matrix to be shown. There is no default for this component. (It is possible to compute the entries of the main table on demand, see the description of the component `Main` in Section 5.4.1. In this situation, the value of the component `main` can be an empty list.)

`header`
: describes a header that shall be shown above the column labels. The value is either a list of attribute lines ("static header") or a function or a record whose component names are names of available modes of the browse table ("dynamic header"). In the function case, the function must take the browse table as its only argument, and return a list of attribute lines. In the record case,

the values of the components must be such functions. It is assumed that the number of these lines depends at most on the mode. The default is an empty list, i. e., there is no header.

`footer`
: describes a footer that shall be shown below the table. The value is analogous to that of `footer`. The default is an empty list, i. e., there is no footer.

`labelsRow`
: is a list of row label rows, each being a list of table cell data objects. These rows are shown to the left of the main table. The default is an empty list, i. e., there are no row labels.

`labelsCol`
: is a list of column information rows, each being a list of table cell data objects. These rows are shown between the header and the main table. The default is an empty list, i. e., there are no column labels.

`corner`
: is a list of lists of table cell data objects that are printed in the upper left corner, i. e., to the left of the column label rows and above the row label columns. The default is an empty list.

`sepRow`
: describes the separators above and below rows of the main table and of the row labels table. The value is either an attribute line or a (not necessarily dense) list of attribute lines. In the former case, repetitions of the attribute line are used as separators below each row and above the first row of the table; in the latter case, repetitions of the entry at the first position (if bound) are used above the first row, and repetitions of the last bound entry before the $(i+2)$-th position (if there is such an entry at all) are used below the $i$-th table row. The default is an empty string, which means that there are no row separators.

`sepCol`
: describes the separators in front of and behind columns of the main table and of the column labels table. The format of the value is analogous to that of the component `sepRow`; the default is the string `" "` (whitespace of width one).

`sepLabelsCol`
: describes the separators above and below rows of the column labels table and of the corner table, analogously to `sepRow`. The default is an empty string, which means that there are no column label separators.

`sepLabelsRow`
: describes the separators in front of and behind columns of the row labels table and of the corner table, analogously to `sepCol`. The default is an empty string.

We give a few examples of standard applications.

The first example defines a small browse table by prescribing only the component `work.main`, so the defaults for row and column labels (no such labels), and for separators are used. The table cells are given by plain strings, so they have height one. Usually this table will fit on the screen.

Example

```
gap> m:= 10;;  n:= 5;;
gap> xpl1:= rec( work:= rec(
```

```
>      main:= List( [ 1 .. m ], i -> List( [ 1 .. n ],
>        j -> String( [ i, j ] ) ) ) );;
gap> BrowseData.IsBrowseTable( xpl1 );
true
```

In the second example, also row and column labels appear, and different separators are used. The table cells have height three. Also this table will usually fit on the screen.

Example

```
gap> m:= 6;;  n:= 5;;
gap> xpl2:= rec( work:= rec(
>      main:= List( [ 1 .. m ], i -> List( [ 1 .. n ],
>        j -> rec( rows:= List( [ -i*j, i*j*1000+j, i-j ], String ),
>                  align:= "c" ) ) ),
>      labelsRow:= List( [ 1 .. m ], i -> [ String( i ) ] ),
>      labelsCol:= [ List( [ 1 .. n ], String ) ],
>      sepRow:= "-",
>      sepCol:= "|",
>  ) );;
gap> BrowseData.IsBrowseTable( xpl2 );
true
```

The third example additionally has a static header and a dynamic footer, and the table cells involve attributes. This table will usually not fit on the screen. Note that `NCurses.attrs.ColorPairs` is available only if the terminal supports colors, which can be checked using `NCurses.attrs.has_colors` (2.2.1).

Example

```
gap> m:= 30;;  n:= 25;;
gap> xpl3:= rec( work:= rec(
>      header:= [ "                    Example 3" ],
>      labelsRow:= List( [ 1 .. 30 ], i -> [ String( i ) ] ),
>      sepLabelsRow:= " % ",
>      sepLabelsCol:= "=",
>      sepRow:= "*",
>      sepCol:= " |",
>      footer:= t -> [ Concatenation( "top-left entry is: ",
>                        String( t.dynamic.topleft{ [ 1, 2] } ) ) ],
>  ) );;
gap> if NCurses.attrs.has_colors then
>   xpl3.work.main:= List( [ 1 .. m ], i -> List( [ 1 .. n ],
>     j -> rec( rows:= [ String( -i*j ),
>                        [ NCurses.attrs.BOLD, true,
>                          NCurses.attrs.ColorPairs[56+1], true,
>                          String( i*j*1000+j ),
>                          NCurses.attrs.NORMAL, true ],
>                          String( i-j ) ],
>               align:= "c" ) ) );
>   xpl3.work.labelsCol:= [ List( [ 1 .. 30 ], i -> [
>     NCurses.attrs.ColorPairs[ 56+4 ], true,
>     String( i ),
>     NCurses.attrs.NORMAL, true ] ) ];
> else
```

```
>   xpl3.work.main:= List( [ 1 .. m ], i -> List( [ 1 .. n ],
>     j -> rec( rows:= [ String( -i*j ),
>                        [ NCurses.attrs.BOLD, true,
>                          String( i*j*1000+j ),
>                          NCurses.attrs.NORMAL, true ],
>                          String( i-j ) ],
>               align:= "c" ) ) );
>   xpl3.work.labelsCol:= [ List( [ 1 .. 30 ], i -> [
>     NCurses.attrs.BOLD, true,
>     String( i ),
>     NCurses.attrs.NORMAL, true ] ) ];
> fi;
gap> BrowseData.IsBrowseTable( xpl3 );
true
```

The fourth example illustrates that highlighting may not work properly for browse tables containing entries whose attributes are not set with explicit Boolean values, see `NCurses.IsAttributeLine` (2.2.3). Call `NCurses.BrowseGeneric` (4.3.1) with the browse table `xpl4`, and select an entry (or a column or a row): Only the middle row of each selected cell will be highlighted, because only in this row, the color attribute is switched on with an explicit `true`.

Example

```
gap> xpl4:= rec(
>     defc:= NCurses.defaultColors,
>     wd:= Maximum( List( ~.defc, Length ) ),
>     ca:= NCurses.ColorAttr,
>     work:= rec(
>       header:= [ "Examples of NCurses.ColorAttr" ],
>       main:= List( ~.defc, i -> List( ~.defc,
>         j -> [ [ ~.ca( i, j ), String( i, ~.wd ) ],        # no true!
>                [ ~.ca( i, j ), true, String( "on", ~.wd ) ],
>                [ ~.ca( i, j ), String( j, ~.wd ) ] ] ) ),  # no true!
>       labelsRow:= List( ~.defc, i -> [ String( i ) ] ),
>       labelsCol:= [ List( ~.defc, String ) ],
>       sepRow:= "-",
>       sepCol:= [ " |", "|" ],
>  ) );;
gap> BrowseData.IsBrowseTable( xpl4 );
true
```

## 4.3 The Function `NCurses.BrowseGeneric`

### 4.3.1 NCurses.BrowseGeneric

▷ `NCurses.BrowseGeneric(`*t*`[, `*arec*`])` (function)

**Returns:** an application dependent value, or nothing.

`NCurses.BrowseGeneric` is used to show the browse table *t* (see `BrowseData.IsBrowseTable` (4.2.3)) in a formatted way on a text screen, and allows the user to navigate in this table.

The optional argument *arec*, if given, must be a record whose components `work` and `dynamic`, if bound, are used to provide defaults for missing values in the corresponding components of *t*.

The default for *arec* and for the components not provided in *arec* is `BrowseData.defaults`, see `BrowseData` (5.4.1), the function `BrowseData.SetDefaults` sets these default values.

At least the component `work.main` must be bound in *t*, with value a list of list of table cell data objects, see `BrowseData.IsBrowseTableCellData` (4.2.1).

When the window or the screen is too small for the browse table, according to its component `work.minyx`, the table will not be shown in visual mode, and `fail` is returned. (This holds also if there would be no return value of the call in a large enough screen.) Thus one should check for `fail` results of programmatic calls of `NCurses.BrowseGeneric`, and one should better not admit `fail` as a regular return value.

Most likely, `NCurses.BrowseGeneric` will not be called on the command line, but the browse table *t* will be composed by a suitable function which then calls `NCurses.BrowseGeneric`, see the examples in Chapter 6.

# Chapter 5

# Browsing Tables in GAP using `ncurses` –The Programming Interface

This chapter describes some aspects of the internals of the browse table handling. The relevant objects are *action functions* that implement the individual navigation steps (see Section 5.1), *modes* that describe the sets of available navigation steps in given situations (see Section 5.2), and *browse applications* that are given by the combination of several modes (see Section 5.3). Most of the related data is stored in the global variable `BrowseData` (5.4.1). For more details, one should look directly at the code in the file `lib/browse.gi` of the Browse package.

## 5.1 Navigation Steps in Browse Tables

Navigating in a browse table means that after entering visual mode by calling `NCurses.BrowseGeneric` (4.3.1), the user hits one or several keys, or uses a mouse button, and if this input is in a given set of admissible inputs then a corresponding function is executed with argument the browse table (plus additional information in the case of mouse events). The function call then may change components in this table (recommended: components in its `dynamic` component), such that the appearance in the window may be different afterwards, and also the admissible inputs and their effects may have changed.

The relation between the admissible inputs and the corresponding functions is application dependent. However, it is recommended to associate the same input to the same function in different situations; for example, the ? key and the F1 key should belong to a function that shows a help window (see Section 5.4.4), the Q key and the ESC key should belong to a function that exits the current mode (Note that the ESC key may be recognized as input only after a delay of about a second.), the Q key should belong to a function that exits the browse application (see Section 5.4.6), the F2 key should belong to a function that saves the current window contents in a global variable (see Section 5.4.5), and the E key should belong to a function that enters a break loop (see Section 5.4.7). The ENTER and RETURN keys should belong to a "click" on a selected table entry, and if a category row is selected then they should expand/collapse this category. The M key should toggle enabling and disabling mouse events. Mouse events on a cell or on a category row of a browse table should move the selected entry to this position; it is recommended that no functionality is lost if no mouse events are used, although the number of steps might be reduced when the mouse is used.

Each such function is wrapped into a record with the components `action` (the function itself) and `helplines` (a list of attribute lines that describes what the function does). The help lines are used by

the help feature of `NCurses.BrowseGeneric`, see Section 5.4.4.

The action functions need not return anything. Whenever the shown screen shall be recomputed after the function call, the component `dynamic.changed` of the browse table must be set to `true` by the action functions.

After entering the first characters of an admissible input that consists of more characters, the last line of the window with the browse table shows these characters behind the prefix "partial input:". One can delete the last entered character of a partial input via the DELETE and BACKSPACE keys. It is not possible to make these keys part of an admissible input. When a partial input is given, only those user inputs have an effect that extend the partial input to (a prefix of) an admissible input. For example, asking for help by hitting the ? key will in general not work if a partial input had been entered before.

## 5.2 Modes in Browse Tables

In different situations, different inputs may be admissible for the same browse table, and different functions may belong to the same input. For example, the meaning of "moving down" can be different depending on whether a cell is selected or not.

The set of admissible user inputs and corresponding functions for a particular situation is collected in a *mode* of the browse table. (There should be no danger to mix up this notion of mode with the "visual mode" introduced in Section 1.1.) A mode is represented by a record with the components `name` (a string used to associate the mode with the components of `header`, `headerLength`, `footer`, `footerLength`, `Click`, and for the help screen), `flag` (a string that describes properties of the mode but that can be equal for different modes), `actions` (a list of records describing the navigation steps that are admissible in the mode, see Section 5.1), and `ShowTables` (the function used to eventually print the current window contents, the default is `BrowseData.ShowTables`). Due to the requirement that each admissible user input uniquely determines a corresponding function, no admissible user input can be a prefix of another admissible input, for the same mode.

Navigation steps (see Section 5.1) can change the current mode or keep the mode. It is recommended that each mode has an action to leave this mode; also an action to leave the browse table application is advisable.

In a browse table, all available modes are stored in the component `work.availableModes`, whose value is a list of mode records. The value of the component `dynamic.activeModes` is a list of mode records that is used as a stack: The *current mode* is the last entry in this list, changing the current mode is achieved by unbinding the last entry (so one returns to the mode from which the current mode had been entered by adding it to the list), by adding a new mode record (so one can later return to the current mode), or by replacing the last entry by another mode record. As soon as the `dynamic.activeModes` list becomes empty, the browse table application is left. (In this situation, if the browse table had been entered from the GAP prompt then visual mode is left, and one returns to the GAP prompt.)

The following modes are predefined by the Browse package. Each of these modes admits the user inputs ?, F1, Q, ESC, Q, F2, E, and M that have been mentioned in Section 5.1.

**browse**
: This mode admits scrolling of the browse table by a cell or by a screen, searching for a string, selecting a row, a column, or an entry, and expanding or collapsing all category rows.

**help** This mode is entered by calling `BrowseData.ShowHelpTable`; it shows a help window concerning the actions available in the mode from which the `help` mode was entered. The `help`

mode admits scrolling in the help table by a cell or by a screen. See Section 5.4.4 for details.

**select_entry**
In this mode, one table cell is regarded as selected; this cell is highlighted using the attribute in the component `work.startSelect` as a prefix of each attribute line, see the remark in Section 2.2.3. The mode admits moving the selection by one cell in the four directions, searching for a string and for further occurrences of this string, expanding or collapsing the current category row or all category rows, and executing the "click" function of this mode, provided that the component `work.Click.( "select_entry" )` of the browse table is bound.

**select_row**
This is like the `select_entry` mode, except that a whole row of the browse table is highlighted. Searching is restricted to the selected row, and "click" refers to the function `work.Click.( "select_row" )`.

**select_row_and_entry**
This is a combination of the `select_entry` mode and the `select_row` mode.

**select_column**
This is like the `select_row` mode, just a column is selected not a row.

**select_column_and_entry**
This is like the `select_row_and_entry` mode, just a column is selected not a row.

## 5.3 Browse Applications

The data in a browse table together with the set of its available modes and the stack of active modes forms a browse application. So the part of or all functionality of the Browse package can be available ("standard application"), or additional functionality can be provided by extending available modes or adding new modes.

When `NCurses.BrowseGeneric` (4.3.1) has been called with the browse table `t`, say, the following loop is executed.

1. If the list `t.dynamic.activeModes` is empty then exit the browse table, and if the component `t.dynamic.Return` is bound then return its value. Otherwise proceed with step 2.

2. If `t.dynamic.changed` is `true` then call the `ShowTables` function of the current mode; this causes a redraw of the window that shows the browse table. Then go to step 3.

3. Get one character of user input. If then the current user input string is the name of an action of the current mode then call the corresponding action function and go to step 1; if the current user input string is just a prefix of the name of some actions of the current mode then go to step 3; if the current user input string is not a prefix of any name of an action of the current mode then discard the last read character and go to step 3.

When one designs a new application, it may be not obvious whether some functionality shall be implemented via one mode or via several modes. As a rule of thumb, introducing a new mode is recommended when one needs a new set of admissible actions in a given situation, and also if one wants to allow the user to perform some actions and then to return to the previous status.

## 5.4 Predefined Browse Functionalities

### 5.4.1 BrowseData

▷ `BrowseData` (global variable)

This is the record that contains the global data used by the function `NCurses.BrowseGeneric` (4.3.1). The components are `actions`, `defaults`, and several capitalized names for which the values are functions.

`BrowseData.actions` is a record containing the action records that are provided by the package, see Section 5.1. These actions are used in standard applications of `NCurses.BrowseGeneric` (4.3.1). Of course there is no problem with using actions that are not stored in `BrowseData.actions`.

`BrowseData.defaults` is a record that contains the defaults for the browse table used as the first argument of `NCurses.BrowseGeneric` (4.3.1). Important components have been described above, see `BrowseData.IsBrowseTable` (4.2.3), in the sense that these components provide default values of `work` components in browse tables. Here is a list of further interesting components.

The following components are provided in `BrowseData.defaults.work`.

`windowParameters`
: is a list of four nonnegative integers, denoting the arguments of `NCurses.newwin` for the window in which the browse table shall be shown. The default is `[ 0, 0, 0, 0 ]`, i. e., the window for the browse table is the full screen.

`minyx`
: is a list of length two, the entries must be either nonnegative integers, denoting the minimal number of rows and columns that are required by the browse table, or unary functions that return these values when they are applied to the browse table; this is interesting for applications that do not support scrolling, or for applications that may have large row or column labels tables. The default is a list with two functions, the return value of the first function is the sum of the heights of the table header, the column labels table, the first table row, and the table footer, and the return value of the second function is the sum of widths of the row labels table and the width of the first column. (If the header/footer is given by a function then this part of the table is ignored in the `minyx` default.) Note that the conditions are checked only when `NCurses.BrowseGeneric` (4.3.1) is called, not after later changes of the screen size in a running browse table application.

`align`
: is a substring of `"bclt"`, which describes the alignment of the browse table in the window. The meaning and the default are the same as for `BrowseData.IsBrowseTableCellData` (4.2.1). (Of course this is relevant only if the table is smaller than the window.)

`headerLength`
: describes the lengths of the headers in the modes for which `header` functions are provided. The value is a record whose component names are names of modes and the corresponding components are nonnegative integers. This component is ignored if the `header` component is unbound or bound to a list, missing values are computed by calls to the corresponding `header` function as soon as they are needed.

`footerLength`
: corresponds to `footer` in the same way as `headerLength` to `header`.

`Main`
: if bound to a function then this function can be used to compute missing values for the component `main`; this way one can avoid computing/storing all `main` values at the same time. The access to the entries of the main matrix is defined as follows: If `mainFormatted[i][j]` is bound then take it, if `main[i][j]` is bound then take it and compute the formatted version, if `Main` is a function then call it with arguments the browse table, `i`, and `j`, and compute the formatted version, otherwise compute the formatted version of `work.emptyCell`. (For the condition whether entries in `mainFormatted` can be bound, see below in the description of the component `cacheEntries`.)

`cacheEntries`
: describes whether formatted values of the entries in the matrices given by the components `corner`, `labelsCol`, `labelsRow`, `main`, and of the corresponding row and column separators shall be stored in the components `cornerFormatted`, `labelsColFormatted`, `labelsRowFormatted`, and `mainFormatted`. The value must be a Boolean, the default is `false`; it should be set to `true` only if the tables are reasonably small.

`cornerFormatted`
: is a list of lists of formatted entries corresponding to the `corner` component. Each entry is either an attribute line or a list of attribute lines (with the same number of displayed characters), the values can be computed from the input format with `BrowseData.FormattedEntry`. The entries are stored in this component only if the component `cacheEntries` has the value `true`. The default is an empty list.

`labelsColFormatted`
: corresponds to `labelsCol` in the same way as `cornerFormatted` to `corner`.

`labelsRowFormatted`
: corresponds to `labelsRow` in the same way as `cornerFormatted` to `corner`.

`mainFormatted`
: corresponds to `main` in the same way as `cornerFormatted` to `corner`.

`m0`
: is the maximal number of rows in the column labels table. If this value is not bound then it is computed from the components `corner` and `labelsCol`.

`n0`
: is the maximal number of columns in `corner` and `labelsRow`.

`m`
: is the maximal number of rows in `labelsRow` and `main`. This value *must* be set in advance if the values of `main` are computed using a `Main` function, and if the number of rows in `main` is larger than that in `labelsRow`.

`n`
: is the maximal number of columns in `labelsCol` and `main`. This value *must* be set in advance if the values of `main` are computed using a `Main` function, and if the number of columns in `main` is larger than that in `labelsCol`.

`heightLabelsCol`
: is a list of 2 `m0`+1 nonnegative integers, the entry at position $i$ is the maximal height of the entries in the $i$-th row of `cornerFormatted` and `labelsColFormatted`. Values that are not bound are computed on demand from the table entries, with the function

`BrowseData.HeightLabelsCol`. (So if one knows the needed heights in advance, it is advisable to set the values, in order to avoid that formatted table entries are computed just for computing their size.) The default is an empty list.

`widthLabelsRow`
: is the corresponding list of 2 `n0`+1 maximal widths of entries in `cornerFormatted` and `labelsRowFormatted`.

`heightRow`
: is the corresponding list of 2 `m`+1 maximal heights of entries in `labelsRowFormatted` and `mainFormatted`.

`widthCol`
: is the corresponding list of 2 `n`+1 maximal widths of entries in `labelsColFormatted` and `mainFormatted`.

`emptyCell`
: is a table cell data object to be used as the default for unbound positions in the four matrices. The default is the empty list.

`sepCategories`
: is an attribute line to be used repeatedly as a separator below expanded category rows. The default is the string `"-"`.

`startCollapsedCategory`
: is a list of attribute lines to be used as prefixes of unhidden but collapsed category rows. For category rows of level $i$, the last bound entry before the $(i+1)$-th position is used. The default is a list of length one, the entry is the boldface variant of the string `"> "`, so collapsed category rows on different levels are treated equally.

`startExpandedCategory`
: is a list of attribute lines to be used as prefixes of expanded category rows, analogously to `startCollapsedCategory`. The default is a list of length one, the entry is the boldface variant of the string `"* "`, so expanded category rows on different levels are treated equally.

`startSelect`
: is an attribute line to be used as a prefix of each attribute line that belongs to a selected cell. The default is to switch the attribute `NCurses.attrs.STANDOUT` on, see Section 2.1.7.

`Click`
: is a record whose component names are names of available modes of the browse table. The values are unary functions that take the browse table as their argument. If the action `Click` is available in the current mode and the corresponding input is entered then the function in the relevant component of the `Click` record is called.

`availableModes`
: is a list whose entries are the mode records that can be used when one navigates through the browse table, see Section 5.2.

`SpecialGrid`
: is a function that takes a browse table and a record as its arguments. It is called by `BrowseData.ShowTables` after the current contents of the window has been computed, and it is intended to draw an individual grid into the table that fits better than anything that can be specified in terms of row and column separators. (If other functions than `BrowseData.ShowTables` are used in some modes of the browse table, these functions must deal with this aspect themselves.) The default is to do nothing.

The following components are provided in `BrowseData.defaults.dynamic`.

`changed`
: is a Boolean that must be set to `true` by action functions whenever a refresh of the window is necessary; it is automatically reset to `false` after the refresh.

`indexRow`
: is a list of positive integers. The entry $k$ at position $i$ means that the $k$-th row in the `mainFormatted` table is shown as the $i$-th row. Note that depending on the current status of the browse table, the rows of `mainFormatted` (and of `main`) may be permuted, or it may even happen that a row in `mainFormatted` is shown several times, for example under different category rows. It is assumed (as a "sort convention") that the *even* positions in `indexRow` point to *even* numbers, and that the subsequent *odd* positions (corresponding to the following separators) point to the subsequent *odd* numbers. The default value is the list $[1,2,\ldots,m]$, where $m$ is the number of rows in `mainFormatted` (including the separator rows, so $m$ is always odd).

`indexCol`
: is the analogous list of positive integers that refers to columns.

`topleft`
: is a list of four positive integers denoting the current topleft position of the main table. The value $[i,j,k,l]$ means that the topleft entry is indexed by the $i$-th entry in `indexRow`, the $j$-th entry in `indexCol`, and the $k$-th row and $l$-th column inside the corresponding cell. The default is $[1,1,1,1]$.

`isCollapsedRow`
: is a list of Booleans, of the same length as the `indexRow` value. If the entry at position $i$ is `true` then the $i$-th row is currently not shown because it belongs to a collapsed category row. It is assumed (as a "hide convention") that the value at any even position equals the value at the subsequent odd position. The default is that all entries are `false`.

`isCollapsedCol`
: is the corresponding list for `indexCol`.

`isRejectedRow`
: is a list of Booleans. If the entry at position $i$ is `true` then the $i$-th row is currently not shown because it does not match the current filtering of the table. Defaults, length, and hide convention are as for `isCollapsedRow`.

`isRejectedCol`
: is the corresponding list for `indexCol`.

`isRejectedLabelsRow`
: is a list of Booleans. If the entry at position $i$ is `true` then the $i$-th column of row labels is currently not shown.

`isRejectedLabelsCol`
: is the corresponding list for the column labels.

`activeModes`
: is a list of mode records that are contained in the `availableModes` list of the `work` component of the browse table. The current mode is the last entry in this list. The default depends on the application, `BrowseData.defaults` prescribes the list containing only the mode with `name` component `"browse"`.

`selectedEntry`
: is a list $[i, j]$. If $i = j = 0$ then no table cell is selected, otherwise $i$ and $j$ are the row and column index of the selected cell. (Note that $i$ and $j$ are always even.) The default is $[0,0]$.

`selectedCategory`
: is a list $[i, l]$. If $i = l = 0$ then no category row is selected, otherwise $i$ and $l$ are the row index and the level of the selected category row. (Note that $i$ is always even.) The default is $[0,0]$.

`searchString`
: is the last string for which the user has searched in the table. The default is the empty string.

`searchParameters`
: is a list of parameters that are modified by the function `BrowseData.SearchStringWithStartParameters`. If one sets these parameters in a search then these values hold also for subsequent searches. So it may make sense to set the parameters to personally preferred ones.

`sortFunctionForColumnsDefault`
: is a function with two arguments used to compare two entries in the same column of the main table (or two category row values). The default is the operation `\<`. (Note that this default may be not meaningful if some of the rows or columns contain strings representing numbers.)

`sortFunctionForRowsDefault`
: is the analogous function for comparing two entries in the same row of the main table.

`sortFunctionsForRows`
: is a list of comparison functions, if the $i$-th entry is bound then it replaces the `sortFunctionForRowsDefault` value when the table is sorted w.r.t. the $i$-th row.

`sortFunctionsForColumns`
: is the analogous list of functions for the case that the table is sorted w.r.t. columns.

`sortParametersForRowsDefault`
: is a list of parameters for sorting the main table w.r.t. entries in given rows, e. g., whether one wants to sort ascending or descending.

`sortParametersForColumnsDefault`
: is the analogous list of parameters for sorting w.r.t. given columns. In addition to the parameters for rows, also parameters concerning category rows are available, e. g., whether the data columns that are transformed into category rows shall be hidden afterwards or not.

`sortParametersForRows`
: is a list that contains ar position $i$, if bound, a list of parameters that shall replace those in `sortParametersForRowsDefault` when the table is sorted w.r.t. the $i$-th row.

`sortParametersForColumns`
: is the analogous list of parameters lists for sorting w.r.t. columns.

`categories`
: describes the current category rows. The value is a list $[l_1, l_2, l_3]$ where $l_1$ is a *sorted* list $[i_1, i_2, ..., i_k]$ of positive integers, $l_2$ is a list of length $k$ where the $j$-th entry is a record with the components `pos` (with value $i_j$), `level` (the level of the category row), `value` (an attribute line to be shown), `separator` (the separator below this category row is a repetition of this string), `isUnderCollapsedCategory` (`true` if the category row is hidden because a category row of an outer level is collapsed; note that in the `false` case, the category row itself can be collapsed), `isRejectedCategory` (`true` if the category row is hidden because none of th edata rows below this category match the current filtering of the table); the list $l_3$ contains the levels for which the category rows shall include the numbers of data rows under these category rows. The default value is `[ [], [], [] ]`. (Note that this "hide convention" makes sense mainly if together with a hidden category row, also the category rows on higher levels and the corresponding data rows are hidden –but this property is *not* checked.) Category rows are computed with the `CategoryValues` function in the `work` component of the browse table.

`log` describes the session log which is currently written. The value is a list of positive integers, representing the user inputs in the current session. When GAP returns from a call to `NCurses.BrowseGeneric` (4.3.1), one can access the log list of the user interactions in the browse table as the value of its component `dynamic.log`.

If `BrowseData.logStore` had been set to `true` before `NCurses.BrowseGeneric` (4.3.1) had been called then the list can also be accessed as the value of `BrowseData.log`. If `BrowseData.logStore` is unbound or has a value different from `true` then `BrowseData.log` is not written. (This can be interesting in the case of browse table applications where the user does not get access to the browse table itself.)

`replay`
: describes the non-interactive input for the current browse table. The value is a record with the components `logs` (a dense list of records, the default is an empty list) and `pointer` (a positive integer, the default is 1). If `pointer` is a position in `logs` then currently the `pointer`-th record is processed, otherwise the browse table has exhausted its non-interactive part, and requires interactive input. The records in `log` have the components `steps` (a list of user inputs, the default is an empty list), `position` (a positive integer denoting the current position in the `steps` list if the log is currently processed, the default is 1), `replayInterval` (the timeout between two steps in milliseconds if the log is processed, the default is 0), and `quiet` (a Boolean, `true` if the steps shall not be shown on the screen until the end of the log is reached, the default is `false`).

### 5.4.2 BrowseData.SetReplay

▷ `BrowseData.SetReplay(`*`data`*`)` (function)

This function sets and resets the value of `BrowseData.defaults.dynamic.replay`.

When `BrowseData.SetReplay` is called with a list *`data`* as its argument then the entries are assumed to describe user inputs for a browse table for which `NCurses.BrowseGeneric` (4.3.1) will be called afterwards, such that replay of the inputs runs. (Valid input lists can be obtained from the component `dynamic.log` of the browse table in question.)

When `BrowseData.SetReplay` is called with the only argument `false`, the component is unbound (so replay is disabled, and thus calls to `NCurses.BrowseGeneric` (4.3.1) will require interactive user input).

The replay feature should be used by initially setting the input list, then running the replay (perhaps several times), and finally unbinding the inputs, such that subsequent uses of other browse tables do not erroneously expect their input in `BrowseData.defaults.dynamic.replay`.

Note that the value of `BrowseData.defaults.dynamic.replay` is used in a call to `NCurses.BrowseGeneric` (4.3.1) only if the browse table in question does not have a component `dynamic.replay` before the call.

### 5.4.3 BrowseData.AlertWithReplay

▷ `BrowseData.AlertWithReplay(`*`t, messages[, attrs]`*`)` (function)

**Returns:** an integer representing a (simulated) user input.

The function `BrowseData.AlertWithReplay` is a variant of `NCurses.Alert` (3.1.1) that is adapted for the replay feature of the browse table *`t`*, see Section 4.1. The arguments *`messages`* and *`attrs`* are the same as the corresponding arguments of `NCurses.Alert` (3.1.1), the argument `timeout` of `NCurses.Alert` (3.1.1) is taken from the browse table *`t`*, as follows. If `BrowseData.IsDoneReplay` returns `true` for `t` then `timeout` is zero, so a user input is requested for closing the alert box; otherwise the requested input character is fetched from `t.dynamic.replay`.

If `timeout` is zero and mouse events are enabled (see `NCurses.UseMouse` (2.2.10)) then the box can be moved inside the window via mouse events.

No alert box is shown if `BrowseData.IsQuietSession` returns `true` when called with *`t`*`.dynamic.replay`, otherwise the alert box is closed after the time (in milliseconds) that is given by the `replayInterval` value of the current entry in *`t`*`.dynamic.replay.logs`.

The function returns either the return value of the call to `NCurses.Alert` (3.1.1) (in the interactive case) or the value that was fetched from the current replay record (in the replay case).

### 5.4.4 BrowseData.actions.ShowHelp

▷ `BrowseData.actions.ShowHelp` (global variable)

There are two predefined ways for showing an overview of the admissible inputs and their meaning in the current mode of a browse table. The function `BrowseData.ShowHelpTable` displays this overview in a browse table (using the `help` mode), and `BrowseData.ShowHelpPager` uses `NCurses.Pager`.

Technically, the only difference between these two functions is that `BrowseData.ShowHelpTable` supports the replay feature of `NCurses.BrowseGeneric` (4.3.1), whereas

BrowseData.ShowHelpPager simply does not call the pager in replay situations.

The action record BrowseData.actions.ShowHelp is associated with the user inputs ? or F1 in standard NCurses.BrowseGeneric (4.3.1) applications, and it is recommended to do the same in other NCurses.BrowseGeneric (4.3.1) applications. This action calls the function stored in the component work.ShowHelp of the browse table, the default (i. e., the value of BrowseData.defaults.work.ShowHelp) is BrowseData.ShowHelpTable.

Example

```
gap> xpl1.work.ShowHelp:= BrowseData.ShowHelpPager;;
gap> BrowseData.SetReplay( "?Q" );
gap> Unbind( xpl1.dynamic );
gap> NCurses.BrowseGeneric( xpl1 );
gap> xpl1.work.ShowHelp:= BrowseData.ShowHelpTable;;
gap> BrowseData.SetReplay( "?dQQ" );
gap> Unbind( xpl1.dynamic );
gap> NCurses.BrowseGeneric( xpl1 );
gap> BrowseData.SetReplay( false );
gap> Unbind( xpl1.dynamic );
```

### 5.4.5 BrowseData.actions.SaveWindow

▷ BrowseData.actions.SaveWindow (global variable)

The function BrowseData.actions.SaveWindow.action asks the user to enter the name of a global GAP variable, using NCurses.GetLineFromUser (3.1.3). If this variable name is valid and if no value is bound to this variable yet then the current contents of the window of the browse table that is given as the argument is saved in this variable, using NCurses.SaveWin (2.2.11).

### 5.4.6 BrowseData.actions.QuitMode

▷ BrowseData.actions.QuitMode (global variable)

▷ BrowseData.actions.QuitTable (global variable)

The function BrowseData.actions.QuitMode.action unbinds the current mode in the browse table that is given as its argument (see Section 5.2), so the browse table returns to the mode from which this mode had been called. If the current mode is the only one, first the user is asked for confirmation whether she really wants to quit the table; only if the Y key is hit, the last mode is unbound.

The function BrowseData.actions.QuitTable.action unbinds all modes in the browse table that is given as its argument, without asking for confirmation; the effect is to exit the browse application (see Section 5.3).

### 5.4.7 BrowseData.actions.Error

▷ BrowseData.actions.Error (global variable)

After NCurses.BrowseGeneric (4.3.1) has been called, interrupting by hitting the CTRL-C keys is not possible. It is recommended to provide the action BrowseData.actions.Error for each mode of a NCurses.BrowseGeneric (4.3.1) application, which enters a break loop and admits returning to the application. The recommended user input for this action is the E key.

# Chapter 6

# Examples of Applications based on NCurses.BrowseGeneric

This chapter introduces the operation Browse (6.1.1) and lists several examples how the function NCurses.BrowseGeneric (4.3.1) can be utilized for rendering GAP related data or for playing games. Each section describes the relevant GAP functions and briefly sketches the technical aspects of the implementation; more details can be found in the GAP files, in the app directory of the package.

Only Section 6.4 describes a standard application in the sense of the introduction to Chapter 4, perhaps except for a special function that is needed to compare table entries. The other examples in this chapter require some of the programming described in Chapter 5.

The GAP examples in this chapter use the "replay" feature of NCurses.BrowseGeneric (4.3.1), see Section 4.1. This means that the NCurses.BrowseGeneric (4.3.1) based function is called between two calls of BrowseData.SetReplay (5.4.2). If you want to paste these examples into the GAP session with the mouse then do not paste the final BrowseData.SetReplay (5.4.2) call, since NCurses.BrowseGeneric (4.3.1) would regard the additional input as a user interrupt.

## 6.1 The Operation Browse

### 6.1.1 Browse

▷ Browse(*obj[, arec]*) (operation)

This operation displays the GAP object *obj* in a nice, formatted way, similar to the operation Display (**Reference: Display**). The difference is that Browse is intended to use ncurses facilities.

Currently there are methods for matrices (see Browse (6.2.2)), for character tables (see Browse (6.3.1)) and for tables of marks (see Browse (6.4.1)).

## 6.2 Matrix Display

The GAP library provides several Display (**Reference: Display**) methods for matrices. In order to cover the functionality of these methods, Browse provides the function NCurses.BrowseDenseList (6.2.1) that uses the standard facilities of the function NCurses.BrowseGeneric (4.3.1), i. e., one can scroll in the matrix, searching and sorting are provided etc.

The idea is to customize this function for different special cases, and to install corresponding `Browse` (6.1.1) methods. Examples are methods for matrices over finite fields and residue class rings of the rational integers, see `Browse` (6.2.2).

The code can be found in the file `app/matdisp.g` of the package.

### 6.2.1 NCurses.BrowseDenseList

▷ `NCurses.BrowseDenseList(`*`list, arec`*`)` (function)

**Returns:** nothing.

Let *`list`* be a dense list whose entries are lists, for example a matrix, and let *`arec`* be a record. This function displays *`list`* in a window, as a two-dimensional array with row and column positions as row and column labels, respectively.

The following components of *`arec`* are supported.

`header`
: If bound, the value must be a valid value of the `work.header` component of a browse table, see `BrowseData.IsBrowseTable` (4.2.3); for example, the value can be a list of strings. If this component is not bound then the browse table has no header.

`footer`
: If bound, the value must be a valid value of the `work.footer` component of a browse table, see `BrowseData.IsBrowseTable` (4.2.3); for example, the value can be a list of strings. If this component is not bound then the browse table has no footer.

`convertEntry`
: If bound, the value must be a unary function that returns a string describing its argument. The default is the operation `String` (**Reference: String**). Another possible value is `NCurses.ReplaceZeroByDot`, which returns the string `"."` if the argument is a zero element in the sense of `IsZero` (**Reference: IsZero**), and returns the `String` (**Reference: String**) value otherwise. For each entry in a row of *`list`*, the `convertEntry` value is shown in the browse table.

`labelsRow`
: If bound, the value must be a list of row label rows for *`list`*, as described in Section `BrowseData.IsBrowseTable` (4.2.3). The default is `[ [ "1" ], [ "2" ], ... ]`.

`labelsCol`
: If bound, the value must be a list of column label rows for *`list`*, as described in Section `BrowseData.IsBrowseTable` (4.2.3). The default is `[ [ "1", "2", ... ] ]`.

The full functionality of the function `NCurses.BrowseGeneric` (4.3.1) is available.

### 6.2.2 Browse (for a list of lists)

▷ `Browse(`*`list`*`)` (method)

**Returns:** nothing.

Several methods for the operation `Browse` (6.1.1) are installed for the case that the argument is a list of lists. These methods cover a default method for lists of lists and the `Display` (**Reference: Display**) methods for matrices over finite fields and residue class rings of the rational integers. Note

that matrices over finite prime fields, small extension fields, and large extension fields are displayed differently, and the same holds for the corresponding Browse (6.1.1) methods.

Example

```
gap> n:= [ 14, 14, 14, 14 ];;
gap> input:= Concatenation( n, n, n, "Q" );;  # ‘‘do nothing and quit’’
gap> BrowseData.SetReplay( input );
gap> Browse( RandomMat( 10, 10, Integers ) );
gap> BrowseData.SetReplay( input );
gap> Browse( RandomMat( 10, 10, GF(3) ) );
gap> BrowseData.SetReplay( input );
gap> Browse( RandomMat( 10, 10, GF(4) ) );
gap> BrowseData.SetReplay( input );
gap> Browse( RandomMat( 10, 10, Integers mod 6 ) );
gap> BrowseData.SetReplay( input );
gap> Browse( RandomMat( 10, 10, GF( NextPrimeInt( 2^16 ) ) ) );
gap> BrowseData.SetReplay( input );
gap> Browse( RandomMat( 10, 10, GF( 2^20 ) ) );
gap> BrowseData.SetReplay( false );
```

## 6.3 Character Table Display

The GAP library provides a Display (**Reference: Display**) method for character tables that breaks the table into columns fitting on the screen. Browse provides an alternative, using the standard facilities of the function NCurses.BrowseGeneric (4.3.1), i. e., one can scroll in the matrix of character values, searching and sorting are provided etc.

The Browse (6.1.1) method for character tables can be called instead of Display (**Reference: Display**). For convenience, one can additionally make this function the default Display (**Reference: Display**) method for character tables, by assigning it to the Display component in the global record CharacterTableDisplayDefaults.User, see (**Reference: Printing Character Tables**); for example, one can do this in one's gaprc file, see (**Reference: The gap.ini and gaprc files**). (This can be undone by unbinding the component CharacterTableDisplayDefaults.User.Display.)

The function BrowseDecompositionMatrix (6.3.2) can be used to display decomposition matrices for Brauer character tables.

### 6.3.1 Browse (for character tables)

▷ Browse(*tbl[, options]*) (method)

This method displays the character table *tbl* in a window. The optional record *options* describes what shall be displayed, the supported components and the default values are described in (**Reference: Printing Character Tables**).

The full functionality of the function NCurses.BrowseGeneric (4.3.1) is available.

Example

```
gap> if TestPackageAvailability( "CTblLib" ) = true then
>      BrowseData.SetReplay( Concatenation(
>         # scroll in the table
>         "DRULdddddrrrrrlluu",
>         # select an entry and move it around
```

```
>         "seddrruuuddlll",
>         # search for the pattern 135 (six times)
>         "/135", [ NCurses.keys.ENTER ], "nnnnn",
>         # deselect the entry, select the first column
>         "qLsc",
>         # sort and categorize by this column
>         "sc",
>         # select the first row, move down the selection
>         "srdddd",
>         # expand the selected category, scroll the selection down
>         "xd",
>         # and quit the application
>         "Q" ) );
>     Browse( CharacterTable( "HN" ) );
>     BrowseData.SetReplay( false );
> fi;
```

*Implementation remarks*: The first part of the code in the `Browse` (6.1.1) method for character tables is almost identical with the code for extracting the data to be displayed from the input data in the GAP library function `CharacterTableDisplayDefault`. The second part of the code transforms these data into a browse table. Character names and (if applicable) indicator values are used as row labels, and centralizer orders, power maps, and class names are used as column labels. The identifier of the table is used as the static header. When an irrational entry is selected, a description of this entry is shown in the dynamic footer.

The standard modes in `BrowseData` (5.4.1) (except the `help` mode) have been extended by three new actions. The first two of them open pagers giving an overview of all irrationalities in the table, or of all those irrationalities that have been shown on the screen in the current call, respectively. The corresponding user inputs are the I and the `I` key. (The names assigned to the irrationalities are generated column-wise. If one just scrolls through the table, without jumping, then these names coincide with the names generated by the default `Display` (**Reference: Display**) method for character tables; this is in general *not* the case, for example when a row-wise search in the table is performed.) The third new action, which is associated with the `P` key, toggles the visibility status of the column label rows for centralizer orders and power maps.

An individual `minyx` function does not only check whether the desired table fits into the window but also whether a table with too high column labels (centralizer orders and power maps) would fit if these labels get collapsed via the `P` key. In this case, the labels are automatically collapsed, and the `P` key is disabled.

In order to keep the required space small also for large character tables, caching of formatted matrix entries is disabled, and the strings to be displayed are computed on demand with a `Main` function in the `work` component of the browse table. For the same reason, the constant height one for all table rows is set in advance, so one need not inspect a whole character if only a few values of it shall be shown.

Special functions are provided for sorting (concerning the comparison of character values, which can be integers or irrationalities) and categorizing the table by a column (the value in the category row involves the class name of the column in question).

The code can be found in the file `app/ctbldisp.g` of the package.

### 6.3.2 BrowseDecompositionMatrix

▷ BrowseDecompositionMatrix(*modtbl[, b][, options]*) (function)

This method displays the decomposition matrix of (the *b*-th block of) the Brauer character table *modtbl* in a window. The arguments are the same as for LaTeXStringDecompositionMatrix (**Reference: LaTeXStringDecompositionMatrix**).

The positions of the ordinary and modular irreducible characters are shown in the labels of the rows and columns, respectively, that are indexed by these characters. When an entry in the decomposition matrix is selected then information about the degrees of these characters is shown in the table footer.

The full functionality of the function NCurses.BrowseGeneric (4.3.1) is available.

Example

```
gap> BrowseData.SetReplay( Concatenation(
>         # select the first entry
>         "se",
>         # scroll in the table
>         "drrrr",
>         # keep the table open for a while
>         [ 14, 14, 14, 14, 14 ],
>         # and quit the application
>         "Q" ) );
gap> BrowseDecompositionMatrix( CharacterTable( "J1" ) mod 2 );
gap> BrowseData.SetReplay( false );
```

The code can be found in the file app/ctbldisp.g of the package.

## 6.4 Table of Marks Display

The GAP library provides a Display (**Reference: Display**) method for tables of marks that breaks the table into columns fitting on the screen. Similar to the situation with character tables, see Section 6.3, but with a much simpler implementation, Browse provides an alternative based on the function NCurses.BrowseGeneric (4.3.1).

Browse (6.1.1) can be called instead of Display (**Reference: Display**) for tables of marks, cf. (**Reference: Printing Tables of Marks**).

### 6.4.1 Browse (for tables of marks)

▷ Browse(*tom[, options]*) (method)

This method displays the table of marks *tom* in a window. The optional record *options* describes what shall be displayed, the supported components and the default values are described in (**Reference: Printing Tables of Marks**).

The full functionality of the function NCurses.BrowseGeneric (4.3.1) is available.

Example

```
gap> if TestPackageAvailability( "TomLib" ) = true then
>      BrowseData.SetReplay( Concatenation(
>         # scroll in the table
>         "DDRRR",
```

```
>         # search for the (exact) value 100 (three times)
>         "/100",
>         [ NCurses.keys.DOWN, NCurses.keys.DOWN, NCurses.keys.RIGHT ],
>         [ NCurses.keys.DOWN, NCurses.keys.DOWN, NCurses.keys.DOWN ],
>         [ NCurses.keys.RIGHT, NCurses.keys.ENTER ], "nn",
>         # no more occurrences of 100, confirm
>         [ NCurses.keys.ENTER ],
>         # and quit the application
>         "Q" ) );
>       Browse( TableOfMarks( "A10" ) );
>       BrowseData.SetReplay( false );
>     fi;
```

*Implementation remarks*: Rows and columns are indexed by their positions. The identifier of the table is used as the static header, there is no footer.

In order to keep the required space small also for large tables of marks, caching of formatted matrix entries is disabled, and the strings to be displayed are computed on demand with a `Main` function in the `work` component of the browse table. For the same reason, the constant height one for the table rows is set in advance. (For example, the table of marks of the group with identifier `"O8+(2)"`, with 11171 rows and columns, can be shown with `Browse` (6.1.1) in a GAP session requiring about 100 MB.)

The code can be found in the file `app/tomdisp.g` of the package.

## 6.5 Table of Contents of AtlasRep

The GAP package AtlasRep (see [WPN+19]) is an interface to a database of representations and related data. The table of contents of this database can be displayed via the function `DisplayAtlasInfo` (**AtlasRep: DisplayAtlasInfo**) of this package. The Browse package provides an alternative based on the function `NCurses.BrowseGeneric` (4.3.1); one can scroll, search, and fetch data for later use.

### 6.5.1 BrowseAtlasInfo

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

**Returns:** the list of "clicked" info records.

This function shows the information available via the GAP package AtlasRep in a browse table, cf. Section (**AtlasRep: Accessing Data via AtlasRep**) in the AtlasRep manual.

The optional arguments can be used to restrict the table to core data or data extensions, or to show an overview for one particular group. The arguments are the same as for `DisplayAtlasInfo` (**AtlasRep: DisplayAtlasInfo**), see the documentation of this function for details. (Note that additional conditions such as `IsPermGroup` (**Reference: IsPermGroup**) can be entered also in the case that no *gapname* is given. In this situation, the additional conditions are evaluated for the "second level tables" that are opened by "clicking" on a table row or entry.)

When one "clicks" on one of the table rows or entries then a browse table with an overview of the information available for this group is shown, and "clicking" on one of the rows in these tables adds the corresponding info record (see `OneAtlasGeneratingSetInfo` (**AtlasRep: OneAtlasGeneratingSetInfo**)) to the list of return values of `BrowseAtlasInfo`.

The full functionality of the function `NCurses.BrowseGeneric` (4.3.1) is available.

The following example shows how `BrowseAtlasInfo` can be used to fetch info records about permutation representations of the alternating groups $A_5$ and $A_6$: We search for the group name `"A5"` in the overview table, and the first cell in the table row for $A_5$ becomes selected; hitting the ENTER key causes a new window to be opened, with an overview of the data available for $A_5$; moving down two rows and hitting the ENTER key again causes the second representation to be added to the result list; hitting Q closes the second window, and we are back in the overview table; we move the selection down twice (to the row for the group $A_6$), and choose the first representation for this group; finally we leave the table, and the return value is the list with the data for the two representations.

Example

```
gap> d:= [ NCurses.keys.DOWN ];;  r:= [ NCurses.keys.RIGHT ];;
gap> c:= [ NCurses.keys.ENTER ];;
gap> BrowseData.SetReplay( Concatenation(
>        "/A5",          # Find the string A5 ...
>        d, d, r,        # ... such that just the word matches,
>        c,              # start the search,
>        c,              # click the table entry A5,
>        d, d,           # move down two rows,
>        c,              # click the row for this representation,
>        "Q",            # quit the second level table,
>        d, d,           # move down two rows,
>        c,              # click the table entry A6,
>        d,              # move down one row,
>        c,              # click the first row,
>        "Q",            # quit the second level table,
>        "Q" ) );        # and quit the application.
gap> if IsBound( BrowseAtlasInfo ) and IsBound( AtlasProgramInfo ) then
>      SetUserPreference( "AtlasRep", "AtlasRepMarkNonCoreData", "" );
>      tworeps:= BrowseAtlasInfo();
>    else
>      tworeps:= [ fail ];
>    fi;
gap> BrowseData.SetReplay( false );
gap> if fail in tworeps then
>      Print( "no access to the Web ATLAS\n" );
>    else
>      Print( List( tworeps, x -> x.identifier[1] ), "\n" );
>    fi;
[ "A5", "A6" ]
```

*Implementation remarks*: The first browse table shown has a static header, no footer and row labels, one row of column labels describing the type of data summarized in the columns.

Row and column separators are drawn as grids (cf. `NCurses.Grid` (2.2.8)) composed from the special characters described in Section 2.1.6, using the component `work.SpecialGrid` of the browse table, see `BrowseData` (5.4.1).

When a row is selected, the "click" functionality opens a new window (via a second level call to `NCurses.BrowseGeneric` (4.3.1)), in which a browse table with the list of available data for the given group is shown; in this table, "click" results in adding the info for the selected row to the result list, and a message about this addition is shown in the footer row. One can choose further data, return

to the first browse table, and perhaps iterate the process for other groups. When the first level table is left, the list of info records for the chosen data is returned.

For the two kinds of browse tables, the standard modes in `BrowseData` (5.4.1) (except the `help` mode) have been extended by a new action that opens a pager giving an overview of all data that have been chosen in the current call. The corresponding user input is the Y key.

This function is available only if the GAP package AtlasRep is available.

The code can be found in the file `app/atlasbrowse.g` of the package.

## 6.6 Access to GAP Manuals–a Variant

A Browse adapted way to access several manuals is to show the hierarchy of books, chapters, sections, and subsections as collapsible category rows, and to regard the contents of each subsection as a data row of a matrix with only one column.

This application is mainly intended as an example with table cells that exceed the screen, and as an example with several category levels.

### 6.6.1 BrowseGapManuals

▷ `BrowseGapManuals(`*`[start]`*`)` (function)

This function displays the contents of the GAP manuals (the main GAP manuals as well as the loaded package manuals) in a window. The optional argument *`start`* describes the initial status, admissible values are the strings `"inline/collapsed"`, `"inline/expanded"`, `"pager/collapsed"`, and `"pager/expanded"`.

In the `inline` cases, the parts of the manuals are shown in the browse table, and in the `pager` case, the parts of the manuals are shown in a different window when they are "clicked", using the user's favourite help viewer, see (**Reference: Changing the Help Viewer**).

In the `collapsed` cases, all category rows are collapsed, and the first row is selected; typical next steps are moving down the selection and expanding single category rows. In the `expanded` cases, all category rows are expanded, and nothing is selected; a typical next step in the `inline/expanded` case is a search for a string in the manuals. (Note that searching in quite slow: For viewing a part of a manual, the file with the corresponding section is read into GAP, the text is formatted, the relevant part is cut out from the section, perhaps markup is stripped off, and finally the search is performed in the resulting strings.)

If no argument is given then the user is asked for selecting an initial status, using `NCurses.Select` (3.1.2).

The full functionality of the function `NCurses.BrowseGeneric` (4.3.1) is available.

Example

```
gap> n:= [ 14, 14, 14 ];;  # ‘‘do nothing’’
gap> BrowseData.SetReplay( Concatenation(
>        "xdxd",                          # expand a Tutorial section
>        n, "Q" ) );                      # and quit
gap> BrowseGapManuals( "inline/collapsed" );
gap> BrowseData.SetReplay( Concatenation(
>        "/Browse", [ NCurses.keys.ENTER ],  # search for "Browse"
>        "xdxddxd",                       # expand a section
>        n, "Q" ) );                      # and quit
```

```
gap> BrowseGapManuals( "inline/collapsed" );
gap> BrowseData.SetReplay( false );
```

*Implementation remarks*: The browse table has a dynamic header showing the name of the currently selected manual, no footer, no row or column labels, and exactly one column of fixed width equal to the screen width. The category rows are precomputed, i. e., they do not arise from a table column; this way, the contents of each data cell can be computed on demand, as soon as it is shown on the screen, in particular the category hierarchy is computed without reading the manuals into GAP. Also, the data rows are not cached. There is no return value. The heights of many cells are bigger than the screen height, so scrolling is a mixture of scrolling to the next cell and scrolling inside a cell. The different initial states are realized via executing different initial steps before the table is shown to the user.

For the variants that show the manuals in a pager, the code temporarily replaces the `show` function of the default viewer `"screen"` (see (**Reference: Changing the Help Viewer**)) by a function that uses `NCurses.Pager` (3.1.4). Note that in the case that the manual bit in question fits into one screen, the default `show` function writes this text directly to the screen, but this is used already by the browse table.

The implementation should be regarded as a sketch.

For example, the markup available in the text file format of GAPDoc manuals (using Esc sequences) is stripped off instead of being transferred to the attribute lines that arise, because of the highlighting problem mentioned in Section 2.2.3.

Some heuristics used in the code are due to deficiencies of the manual formats.

For the inline variant of the browse table, the titles of chapters, sections, and subsections are *not* regarded as parts of the actual text since they appear already as category rows; however, the functions of the GAP help system deliver the text *together with* these titles, so these lines must be stripped off afterwards.

The category hierarchy representing the tables of contents is created from the `manual.six` files of the manuals. These files do not contain enough information for determining whether several functions define the same subsection, in the sense that there is a common description text after a series of manual lines introducing different functions. In such cases, the browse table contains a category row for each of these functions (with its own number), but the corresponding text appears only under the *last* of these category rows, the data rows for the others are empty. (This problem does not occur in the GAPDoc manual format because this introduces explicit subsection titles, involving only the *first* of several function definitions.)

Also, index entries and sectioning entries in `manual.six` files of manuals in GAPDoc format are not explicitly distinguished.

The code can be found in the file `app/manual.g` of the package.

## 6.7 Overview of Bibliographies

The function `BrowseBibliography` (6.7.1) can be used to turn the contents of bibliography files in BibTeX or BibXMLext format (see (**GAPDoc: The BibXMLext Format**)) into a Browse table, such that one can scroll in the list, search for entries, sort by year, sort and categorize by authors etc.

The default bibliography used by `BrowseBibliography` (6.7.1) is the bibliography of GAP related publications, see [GAP]. The Browse package contains a (perhaps outdated) version of this bibliography. One can get an updated version as follows.

```
wget -N http://www.gap-system.org/Doc/Bib/gap-publishednicer.bib
```

The columns of the Browse table that is shown by `BrowseBibliography` (6.7.1) can be customized, two examples for that are given by the functions `BrowseBibliographySporadicSimple` (**AtlasRep: BrowseBibliographySporadicSimple**) and `BrowseBibliographyGapPackages` (6.7.2).

The function `BrowseMSC` (6.7.3) shows an overview of the AMS Mathematics Subject Classification codes.

### 6.7.1 BrowseBibliography

▷ `BrowseBibliography(`*`[bibfiles]`*`)` (function)

**Returns:** a record as returned by `ParseBibXMLExtFiles` (**GAPDoc: ParseBibXMLextFiles**).

This function shows the list of bibliography entries in the files given by *`bibfiles`*, which may be a string or a list of strings (denoting a filename or a list of filenames, respectively) or a record (see below for the supported components).

If no argument is given then the file `bibl/gap-publishednicer.bib` in the Browse package directory is taken, and `"GAP Bibliography"` is used as the header.

Unfortunately `bibl/gap-publishednicer.bib` does does not have a copyright statement saying it can be distributed under a free license. Therefore it can not be distributed by the debian free distribution. You can however retrieve the file using the script `/usr/share/gap/pkg/Browse/bibl/getnewestbibfile`. Copying the retrieved file to `/usr/share/gap/pkg/Browse/bibl/gap-publishednicer.bib` will enable the functionality.

Another perhaps interesting data file that should be available in the GAP distribution is `doc/manualbib.xml`. This file can be located as follows.

Example

```
gap> file:= Filename( DirectoriesLibrary( "doc" ), "manualbib.xml" );;
```

Both BibTeX format and the XML based extended format provided by the GAPDoc package are supported by `BrowseBibliography`, see Chapter (**GAPDoc: Utilities for Bibliographies**).

In the case of BibTeX format input, first a conversion to the extended format takes place, via `StringBibAsXMLext` (**GAPDoc: StringBibAsXMLext**) and `ParseBibXMLextString` (**GAPDoc: ParseBibXMLextString**). Note that syntactically incorrect entries are rejected in this conversion –this is signaled with `InfoBibTools` (**GAPDoc: InfoBibTools**) warnings– and that only a subset of the possible LaTeX markup is recognized –other markup appears in the browse table except that the leading backslash is removed.

In both cases of input, the problem arises that in visual mode, currently we can show only ASCII characters (and the symbols in `NCurses.lineDraw`, but these are handled differently, see Section 2.1.6). Therefore, we use the function `SimplifiedUnicodeString` (**GAPDoc: SimplifiedUnicodeString**) for replacing other unicode characters by ASCII text.

The return value is a record as returned by `ParseBibXMLExtFiles` (**GAPDoc: ParseBibXMLextFiles**), its `entries` component corresponds to the bibliography entries that have been "clicked" in visual mode. This record can be used as input for `WriteBibFile` (**GAPDoc: WriteBibFile**) or `WriteBibXMLextFile` (**GAPDoc: WriteBibXMLextFile**), in order to produce a bibliography file, or it can be used as input for `StringBibXMLEntry` (**GAPDoc: StringBibXMLEntry**), in order to produce strings from the entries, in various formats.

The full functionality of the function `NCurses.BrowseGeneric` (4.3.1) is available.

Example

```
gap> # sort and categorize by year, scroll down, expand a category row
gap> BrowseData.SetReplay( "scrrscsedddddxdddddQ" );
gap> BrowseBibliography();;
gap> # sort & categorize by authors, expand all category rows, scroll down
gap> BrowseData.SetReplay( "scscXseddddddQ" );
gap> BrowseBibliography();;
gap> # sort and categorize by journal, search for a journal name, expand
gap> BrowseData.SetReplay( Concatenation( "scrrrsc/J. Algebra",
>        [ NCurses.keys.ENTER ], "nxdddQ" ) );
gap> BrowseBibliography();;
gap> BrowseData.SetReplay( false );
```

*Implementation remarks*: The browse table has a dynamic header (showing the number of entries, which can vary when the table is restricted), no footer and row labels; one row of column labels is given by the descriptions of the table columns (authors, title, year, journal, MSC code).

Row and column separators are drawn as grids (cf. `NCurses.Grid` (2.2.8)) composed from the special characters described in Section 2.1.6, using the component `work.SpecialGrid` of the browse table, see `BrowseData` (5.4.1).

For categorizing by authors (or by MSC codes), the sort parameter `"split rows on categorizing"` is set to `"yes"`, so the authors (codes) are distributed to different category rows, hence each entry appears once for each of its authors (or its MSC codes) in the categorized table. When a data row or an entry in a data row is selected, "click" adds the corresponding bibliographhy entry to the result.

The width of the title column is preset; usually titles are too long for one line, and the contents of this column is formatted as a paragraph, using the function `FormatParagraph` (**GAPDoc: FormatParagraph**). For the authors and journal columns, maximal widths are prescribed, and `FormatParagraph` (**GAPDoc: FormatParagraph**) is used for longer entries.

For four columns, the sort parameters are defined as follows: The *authors* and *MSC code* columns do not become hidden when the table is categorized according to this column, sorting by the *year* yields a *de*scending order, and the category rows arising from these columns and the *journal* column show the numbers of the data rows that belong to them.

Those standard modes in `BrowseData` (5.4.1) where an entry or a row of the table is selected have been extended by three new actions, which open a pager showing the BibTeX, HTML, and Text format of the selected entry, respectively. The corresponding user inputs are the VB, VH, and VT. If the *MSC code* column is available then also the user input VM is admissible; it opens a pager showing the descriptions of the MSC codes attached to the selected entry.

This function requires some of the utilities provided by the GAP package GAPDoc (see [LN07]), such as `FormatParagraph` (**GAPDoc: FormatParagraph**), `NormalizeNameAndKey` (**GAPDoc: NormalizeNameAndKey**), `NormalizedNameAndKey` (**GAPDoc: NormalizedNameAndKey**), `ParseBibFiles` (**GAPDoc: ParseBibFiles**), `ParseBibXMLextFiles` (**GAPDoc: ParseBibXMLextFiles**), `ParseBibXMLextString` (**GAPDoc: ParseBibXMLextString**), `RecBibXMLEntry` (**GAPDoc: RecBibXMLEntry**), and `StringBibAsXMLext` (**GAPDoc: StringBibAsXMLext**).

The code can be found in the file `app/gapbibl.g` of the package.

The browse table can be customized by entering a record as the argument of `BrowseBibliography`, with the following supported components.

`files`
: a nonempty list of filenames containing the data to be shown; there is no default for this component.

`filesshort`
: a list of the same length as the `files` component, the entries are strings which are shown in the `"sourcefilename"` column of the table (if this column is present); the default is the list of filenames.

`filecontents`
: a list of the same length as the `files` component, the entries are strings which are shown as category values when the table is categorized by the `"sourcefilename"` column; the default is the list of filenames.

`header`
: is the constant part of the header shown above the browse table, the default is the first filename.

`columns`
: is a list of records that are valid as the second argument of `DatabaseAttributeAdd` (A.1.5), where the first argument is a database id enumerator created from the bibliography entries in the files in question. Each entry (and also the corresponding identifier) of this database id enumerator is a list of records obtained from `ParseBibXMLextFiles` (**GAPDoc: ParseBibXMLextFiles**) and `RecBibXMLEntry` (**GAPDoc: RecBibXMLEntry**), or from `ParseBibFiles` (**GAPDoc: ParseBibFiles**), such that the list elements are regarded as equal, in the sense that their fingerprints (see below) are equal. The records in the `columns` list are available for constructing the desired browse table, the actual appearance is controlled by the `choice` component described below. Columns showing authors, title, year, journal, MSC code, and filename are predefined and need not be listed here.

`choice`
: a list of strings denoting the `identifier` components of those columns that shall actually be shown in the table, the default is `[ "authors", "title", "year", "journal", "mrclass" ]`.

`fingerprint`
: a list of strings denoting component names of the entries of the database id enumerator that is constructed from the data (see above); two data records are regarded as equal if the values of these components are equal; the default is `[ "mrnumber", "title", "authorAsList", "editorAsList", "author" ]`.

`sortKeyFunction`
: either `fail` or a function that takes a record as returned by `RecBibXMLEntry` (**GAPDoc: RecBibXMLEntry**) and returns a list that is used for comparing and thus sorting the records; the default is `fail`, which means that the rows of the table appear in the same ordering as in the source files.

### 6.7.2 BrowseBibliographyGapPackages

▷ `BrowseBibliographyGapPackages()` (function)

**Returns:** a record as returned by `BrowseBibliography` (6.7.1).

This function collects the information from the `*.bib` and `*bib.xml` files in those subdirectories of installed GAP packages which contain the package documentation, and shows it in a Browse table, using the function `BrowseBibliography` (6.7.1).

*This function is experimental.* The result is not really satisfactory, for the following reasons.

- Duplicate entries may occur, due to subtle differences in various source files.
- The source files may contain more than what is actually cited in the package manuals.
- It may happen that some `*.bib` or `*bib.xml` file is accidentally distributed with the package but is not intended to serve as package bibliography.
- The heuristics for rewriting LaTeX code is of course not perfect; thus strange symbols may occur in the Browse table.

### 6.7.3 BrowseMSC

▷ `BrowseMSC(`*`[version]`*`)` (function)

**Returns:** nothing.

BrowseMSC requires data that can not be distributed as part of the debian free distribution. You can however get the files mscdata2010.txt, mscdiffs2000to2010.txt, mscdata2020.txt and mscdiffs2010to2020.txt from the upstream sources and copy them to /usr/share/gap/pkg/Browse/bibl/. Putting the files there will enable the functionality.

This function shows the valid MSC codes in a browse table that is categorized by the `..-XX` and the `...xx` codes. (Use X for expanding all categories or x for expanding the currently selected category.) Due to the categorization, only two columns of the table are visible, showing the codes and their descriptions.

If *`version`* is given then it must be one of the numbers 2010 or 2020, meaning that the MSC2010 or MSC2020 codes are shown; the default for *`version`* is 2020.

## 6.8 Profiling GAP functions–a Variant

A Browse adapted way to evaluate profiling results is to show the overview that is printed by the GAP function `DisplayProfile` (**Reference: DisplayProfile**) in a Browse table, which allows one to sort the profiled functions according to the numbers of calls, the time spent, etc., and to search for certain functions one is interested in.

### 6.8.1 BrowseProfile

▷ `BrowseProfile(`*`[functions][,] [mincount, mintime]`*`)` (function)

The arguments and their meaning are the same as for the function `DisplayProfile` (**Reference: DisplayProfile**), in the sense that the lines printed by that function correspond to the rows of the list that is shown by `BrowseProfile`. Initially, the table is sorted in the same way as the list shown by `BrowseProfile`; sorting the table by any of the first five columns will yield a non-increasing order of the rows.

The threshold values *`mincount`* and *`mintime`* can be changed in visual mode via the user input E. If mouse events are enabled (see `NCurses.UseMouse` (2.2.10)) then one can also use a mouse click

on the current parameter value shown in the table header in order to enter the mode for changing the parameters.

When a row or an entry in a row is selected, "click" shows the code of the corresponding function in a pager (see `NCurses.Pager` (3.1.4)) whenever this is possible, as follows. If the function was read from a file then this file is opened, if the function was entered interactively then the code of the function is shown in the format produced by `Print` (**Reference: Print**); other functions (for example GAP kernel functions) cannot be shown, one gets an alert message (see `NCurses.Alert` (3.1.1)) in such a case.

The full functionality of the function `NCurses.BrowseGeneric` (4.3.1) is available.

Example

```
gap> n:= [ 14, 14, 14, 14, 14 ];;  # ``do nothing''
gap> ProfileOperationsAndMethods( true );    # collect some data
gap> ConjugacyClasses( PrimitiveGroup( 24, 1 ) );;
gap> ProfileOperationsAndMethods( false );
gap> BrowseData.SetReplay( Concatenation(
>        "e",                                  # edit threshold paras
>        [ NCurses.keys.DC ], "2", "\t",       # replace 10000 by 20000
>        [ NCurses.keys.DC ], "2",             # replace 30 by 20
>        [ NCurses.keys.ENTER ],               # commit the changes
>        "scso",                               # sort by column 1,
>        n,
>        "rso",                                # sort by column 2,
>        n,
>        "rso",                                # sort by column 3,
>        n,
>        "q",                                  # deselect the column,
>        "/Normalizer", [ NCurses.keys.ENTER ],  # search for a function,
>        n, n, n, "Q" ) );                     # and quit
gap> BrowseProfile();
gap> BrowseData.SetReplay( false );
```

*Implementation remarks*: The browse table has a dynamic header, which shows the current values of *`mincount`* and *`mintime`*, and a dynamic footer, which shows the sums of counts and timings for the rows in the table (label `TOTAL`) and if applicable the sums for the profiled functions not shown in the table (label `OTHER`). There are no row labels, and the obvious column labels. There is no return value.

The standard modes in `BrowseData` (5.4.1) (except the `help` mode) have been modified by adding a new action for changing the threshold parameters *`mincount`* and *`mintime`* (user input E). The way how this in implemented made it necessary to change the standard "reset" action (user input !) of the table; note that resetting (a sorting or filtering of) the table must not make those rows visible that shall be hidden because of the threshold parameters.

The code can be found in the file `app/profile.g` of the package.

## 6.9 Variables defined in GAP packages–a Variant

A Browse adapted way to list the variables that are defined in a GAP package is to show the overview that is printed by the GAP function `ShowPackageVariables` (**Reference: ShowPackageVariables**) in a Browse table.

### 6.9.1 BrowsePackageVariables

▷ `BrowsePackageVariables(`*`pkgname[, version][, arec]`*`)` (function)

**Returns:** nothing.

The arguments can be the same as for `ShowPackageVariables` (**Reference: ShowPackageVariables**), that is, *`pkgname`* is the name of a GAP package, and the optional arguments *`version`* and *`arec`* are a version number of this package and a record used for customizing the output, respectively.

Alternatively, the second argument can be the output `info` of `PackageVariablesInfo` (**Reference: PackageVariablesInfo**) for the package in question, instead of the version number.

`BrowsePackageVariables` opens a browse table that shows the global variables that become bound and the methods that become installed when GAP loads the package *`pkgname`*.

The table is categorized by the kinds of variables (new or redeclared operations, methods, info classes, synonyms, other globals). The column "Doc.?" distinguishes undocumented and documented variables, so one can use this column as a filter or for categorizing. The column "Filename" shows the names of the package files. Clicking a selected row of the table opens the relevant package file at the code in question.

The idea behind the argument `info` is that using the same arguments as for `ShowPackageVariables` (**Reference: ShowPackageVariables**) does not allow one to apply `BrowsePackageVariables` to packages that have been loaded before the Browse package. Thus one can compute the underlying data `info` first, using `PackageVariablesInfo` (**Reference: PackageVariablesInfo**), then load the Browse package, and finally call `BrowsePackageVariables`.

For example, the overview of package variables for Browse can be shown by starting GAP without packages and then entering the following lines.

Example

```
gap> pkgname:= "Browse";;
gap> info:= PackageVariablesInfo( pkgname, "" );;
gap> LoadPackage( "Browse" );;
gap> BrowsePackageVariables( pkgname, info );
```

If the arguments are the same as for `ShowPackageVariables` (**Reference: ShowPackageVariables**) then this function is actually called, with the consequence that the package gets loaded when `BrowsePackageVariables` is called. This is not the case if the output of `PackageVariablesInfo` (**Reference: PackageVariablesInfo**) is entered as the second argument.

## 6.10 Configuring User preferences–a Variant

A Browse adapted way to show and edit GAP's user preferences is to show the overview that is printed by the GAP function `ShowUserPreferences` (**Reference: ShowUserPreferences**) in a Browse table.

### 6.10.1 BrowseUserPreferences

▷ `BrowseUserPreferences(`*`package1, package2, ...`*`)` (function)

**Returns:** nothing.

The arguments are the same as for `ShowUserPreferences` (**Reference: ShowUserPreferences**), that is, calling the function with no argument yields an overview of all known user preferences, and

if one or more strings *package1*, ... are given then only the user preferences for these packages are shown.

`BrowseUserPreferences` opens a browse table with the following columns:

**"Package"**
contains the names of the GAP packages to which the user preferences belong,

**"Pref. names"**
contains the names of the user preferences, and

**"Description"**
contains the `description` texts from the `DeclareUserPreference` (**Reference: DeclareUserPreference**) calls and the default values (if applicable), and the actual values.

When one "clicks" on one of the table rows or entries then the values of the user preference in question can be edited. If a list of admissible values is known then this means that one can choose from this list via `NCurses.Select` (3.1.2), otherwise one can enter the desired value as text.

The values of the user preferences are not changed before one closes the browse table. When the table is left and if one has changed at least one value, one is asked whether the changes shall be applied.

Example

```
gap> d:= [ NCurses.keys.DOWN ];;
gap> c:= [ NCurses.keys.ENTER ];;
gap> BrowseData.SetReplay( Concatenation(
>        "/PackagesToLoad",  # enter a search string,
>        c,                  # start the search,
>        c,                  # edit the entry (a list of choices),
>        " ", d,             # toggle the first four values,
>        " ", d,             #
>        " ", d,             #
>        " ", d,             #
>        c,                  # submit the values,
>        "Q",                # quit the table,
>        c ) );              # choose "cancel": do not apply the changes.
gap> BrowseUserPreferences();
gap> BrowseData.SetReplay( false );
```

The code can be found in the file `app/userpref.g` of the package.

## 6.11 Overview of GAP Data

The GAP system contains several data collections such as libraries of groups and character tables. Clearly the function `NCurses.BrowseGeneric` (4.3.1) can be used to visualize interesting information about such data collections, in the form of an "overview table" whose rows correspond to the objects in the collection; each column of the table shows a piece of information about the objects. (One possibility to create such overviews is given by `BrowseTableFromDatabaseIdEnumerator` (A.2.2).)

### 6.11.1 BrowseGapData

▷ `BrowseGapData()` (function)

**Returns:** the return value of the chosen application if there is one.

The function `BrowseGapData` shows the choices in the list `BrowseData.GapDataOverviews`, in a browse table with one column. When an entry is "clicked" then the associated function is called, and the table of choices is closed.

The idea is that each entry of `BrowseData.GapDataOverviews` describes an overview of a data collection.

The Browse package provides overviews of

- the current AMS Mathematics Subject Classification codes (see `BrowseMSC` (6.7.3)),
- the contents of the AtlasRep package [WPN+19] (only if this package is loaded, see Section 6.5),
- the Conway polynomials in GAP (calls `BrowseConwayPolynomials()`),
- profile information for GAP functions (see Section 6.8),
- the list of GAP related bibliography entries in the file `bibl/gap-publishednicer.bib` of the Browse package (see Section 6.7),
- the GAP manuals (see Section 6.6),
- GAP operations and methods (calls `BrowseGapMethods()`),
- the installed GAP packages (calls `BrowseGapPackages()`),
- GAP's user preferences (see Section 6.10),
- the contents of the TomLib package [NMP13] (only if this package is loaded, see Section A.4),

Other GAP packages may add more overviews, using the function `BrowseGapDataAdd` (6.11.2). For example, there are overviews of

- the bibliographies in the ATLAS of Finite Groups [CCN+85] and in the ATLAS of Brauer Characters [JLPW95] (see `BrowseBibliographySporadicSimple` (**AtlasRep: BrowseBibliographySporadicSimple**)),
- atomic irrationalities that occur in character tables in the ATLAS of Finite Groups [CCN+85] or the ATLAS of Brauer Characters [JLPW95] (see Section `BrowseCommonIrrationalities` (**CTblLib: BrowseCommonIrrationalities**)),
- the differences between the versions of the character table data in the CTblLib package (see Section `BrowseCTblLibDifferences` (**CTblLib: BrowseCTblLibDifferences**)),
- the information in the GAP Character Table Library (see Section `BrowseCTblLibInfo` (**CTblLib: BrowseCTblLibInfo**)),
- an overview of minimal degrees of representations of groups from the ATLAS of Group Representations (see Section `BrowseMinimalDegrees` (**AtlasRep: BrowseMinimalDegrees**)).

Except that always one table cell is selected, the full functionality of the function NCurses.BrowseGeneric (4.3.1) is available.

Example

```
gap> n:= [ 14, 14, 14 ];;  # ‘‘do nothing’’
gap> # open the overview of Conway polynomials
gap> BrowseData.SetReplay( Concatenation( "/Conway Polynomials",
>      [ NCurses.keys.ENTER, NCurses.keys.ENTER ], "srdddd", n, "Q" ) );
gap> BrowseGapData();;
gap> # open the overview of GAP packages
gap> BrowseData.SetReplay( Concatenation( "/GAP Packages",
>      [ NCurses.keys.ENTER, NCurses.keys.ENTER ], "/Browse",
>      [ NCurses.keys.ENTER ], "n", n, "Q" ) );
gap> BrowseGapData();;
gap> BrowseData.SetReplay( false );
```

*Implementation remarks*: The browse table has a static header, a dynamic footer showing the description of the currently selected entry, no row or column labels, and exactly one column of fixed width equal to the screen width. If the chosen application has a return value then this is returned by BrowseGapData, otherwise nothing is returned. The component work.SpecialGrid of the browse table is used to draw a border around the list of choices and another border around the footer. Only one mode is needed in which an entry is selected.

The code can be found in the file app/gapdata.g of the package.

### 6.11.2 BrowseGapDataAdd

▷ BrowseGapDataAdd(*title, call[, ret], documentation*) (function)

This function extends the list BrowseData.GapDataOverviews by a new entry. The list is used by BrowseGapData (6.11.1).

*title* must be a string of length at most 76; it will be shown in the browse table that is opened by BrowseGapData (6.11.1). *call* must be a function that takes no arguments; it will be called when *title* is "clicked". *ret*, if given, must be true if *call* has a return value and if BrowseGapData (6.11.1) shall return this value, and false otherwise. *documentation* must be a string that describes what happens when the function *call* is called; it will be shown in the footer of the table opened by BrowseGapData (6.11.1) when *title* is selected.

## 6.12 Navigating in a Directory Tree

A natural way to visualize the contents of a directory is via a tree whose leaves denote plain files, and the other vertices denote subdirectories. Browse provides a function based on NCurses.BrowseGeneric (4.3.1) for displaying such trees; the leaves correspond to the data rows, and the other vertices correspond to category rows.

### 6.12.1 BrowseDirectory

▷ BrowseDirectory(*[dir]*) (function)

**Returns:** a list of the "clicked" filenames.

If no argument is given then the contents of the current directory is shown, see `DirectoryCurrent` (**Reference: DirectoryCurrent**). If a directory object *dir* (see `Directory` (**Reference: Directory**)) is given as the only argument then the contents of this directory is shown; alternatively, *dir* may also be a string which is then understood as a directory path.

The full functionality of the function `NCurses.BrowseGeneric` (4.3.1) is available.

Example

```
gap> n:= [ 14, 14, 14 ];;  # ‘‘do nothing’’
gap> BrowseData.SetReplay( Concatenation(
>        "q",                                  # leave the selection
>        "X",                                  # expand all categories
>        "/filetree", [ NCurses.keys.ENTER ],  # search for "filetree"
>        n, "Q" ) );                           # and quit
gap> dir:= DirectoriesPackageLibrary( "Browse", "" )[1];;
gap> if IsBound( BrowseDirectory ) then
>      BrowseDirectory( dir );
>    fi;
gap> BrowseData.SetReplay( false );
```

*Implementation remarks*: The browse table has a static header, no footer, no row or column labels, and exactly one data column. The category rows are precomputed, i. e., they do not arise from a table column. The tree structure is visualized via a special grid that is shown in the separator column in front of the table column; the width of this column is computed from the largest nesting depth of files. For technical reasons, category rows representing *empty* directories are realized via “dummy” table rows; a special `ShowTables` function guarantees that these rows are always hidden.

When a data row or an entry in this row is selected, “click” adds the corresponding filename to the result list. Initially, the first row is selected. (So if you want to search in the whole tree then you should quit this selection by hitting the `Q` key.)

The category hierarchy is computed using `DirectoryContents` (**Reference: DirectoryContents**).

This function is available only if the GAP package IO (see [Neu07]) is available, because the check for cycles uses the function `IO_stat` (**IO: IO_stat**) from this package.

The code can be found in the file `app/filetree.g` of the package.

## 6.13 A Puzzle

We consider an $m$ by $n$ rectangle of squares numbered from 1 to $mn-1$, the bottom right square is left empty. The numbered squares are permuted by successively exchanging the empty square and a neighboring square such that in the end, the empty cell is again in the bottom right corner.

| | | | |
|---|---|---|---|
| 7 | 13 | 14 | 2 |
| 1 | 4 | 15 | 11 |
| 6 | 8 | 3 | 9 |
| 10 | 5 | 12 | |

The aim of the game is to order the numbered squares via these moves.

For the case $m = n = 4$, the puzzle is (erroneously?) known under the name “Sam Loyd’s Fifteen”, see [Bog] and [OR] for more information and references.

### 6.13.1 BrowsePuzzle

▷ `BrowsePuzzle(`*`[m, n[, pi]]`*`)` (function)

**Returns:** a record describing the initial and final status of the puzzle.

This function shows the rectangle in a window.

The arguments *m* and *n* are the dimensions of the rectangle, the default for both values is 4. The initial distribution of the numbers in the squares can be prescribed via a permutation *pi*, the default is a random element in the alternating group on the points $1, 2, \ldots, mn-1$. (Note that the game has not always a solution.)

In any case, the empty cell is selected, and the selection can be moved to neighboring cells via the arrow keys, or to any place in the same row or column via a mouse click.

The return value is a record with the components `dim` (the pair `[ m, n ]`), `init` (the initial permutation), `final` (the final permutation), and `steps` (the number of transpositions that were needed).

Example

```
gap> BrowseData.SetReplay( Concatenation(
>        BrowsePuzzleSolution.steps, "Q" ) );
gap> BrowsePuzzle( 4, 4, BrowsePuzzleSolution.init );;
gap> BrowseData.SetReplay( false );
```

An implementation using only mouse clicks but no key strokes is available in the GAP package XGAP (see [CN04]).

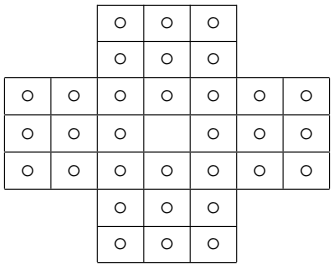
*Implementation remarks*: The game board is implemented via a browse table, without row and column labels, with static header, dynamic footer, and individual `minyx` function. Only one mode is needed in which one cell is selected, and besides the standard actions for quitting the table, asking for help, and saving the current window contents, only the four moves via the arrow keys and mouse clicks are admissible.

Some standard `NCurses.BrowseGeneric` (4.3.1) functionality, such as scrolling, selecting, and searching, are not available in this application.

The code can be found in the file `app/puzzle.g` of the package.

## 6.14 Peg Solitaire

Peg solitaire is a board game for one player. The game board consists of several holes some of which contain pegs. In each step of the game, one peg is moved horizontally or vertically to an empty hole at distance two, by jumping over a neighboring peg which is then removed from the board.



We consider the game that in the beginning, exactly one hole is empty, and in the end, exactly one peg is left.

### 6.14.1 PegSolitaire

▷ `PegSolitaire(`*`[format][,] [nrholes][,] [twoModes]`*`)` (function)

This function shows the game board in a window.

If the argument *`format`* is one of the strings `"small"` or `"large"` then small or large pegs are shown, the default is `"small"`.

Three shapes of the game board are supported, with 33, 37, and 45 holes, respectively; this number can be specified via the argument *`nrholes`*, the default is 33. In the cases of 33 and 45 holes, the position of both the initial hole and the destination of the final peg is the middle cell, whereas in the case of 37 holes, the initial hole is in the top left position and the final peg has to be placed in the bottom right position.

If a Boolean *`twoModes`* is entered as an argument then it determines whether a browse table with one or two modes is used; the default `false` yields a browse table with only one mode.

In any case, one cell of the board is selected, and the selection can be moved to neighboring cells via the arrow keys. A peg in the selected cell jumps over a neighboring peg to an adjacent hole via the `j` key followed by the appropriate arrow key.

Example

```
gap> for n in [ 33, 37, 45 ] do
>      BrowseData.SetReplay( Concatenation(
>          PegSolitaireSolutions.( String( n ) ), "Q" ) );
>      PegSolitaire( n );
>      PegSolitaire( "large", n );
>      PegSolitaire( n, true );
>      PegSolitaire( "large", n, true );
> od;
gap> BrowseData.SetReplay( false );
```

For more information such as variations of the game and references, see [Köla]. Also the solutions stored in the variable `PegSolitaireSolutions` have been taken from this web page.

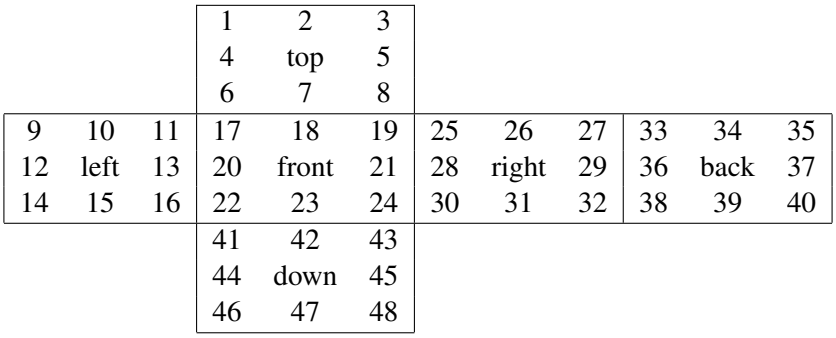
*Implementation remarks*: The game board is implemented via a browse table, without row and column labels, with static header, dynamic footer, and individual `minyx` function. In fact, two implementations are provided. The first one needs only one mode in which one cell is selected; moving the selection and jumping with the peg in the selected cell in one of the four directions are the supported user actions. The second implementation needs two modes, one for moving the selection and one for jumping.

Some standard `NCurses.BrowseGeneric` (4.3.1) functionality, such as scrolling, selecting, and searching, are not available in this application.

The code can be found in the file `app/solitair.g` of the package.

## 6.15 Rubik's Cube

We visualize the transformations of Rubik's magic cube in a model that is given by "unfolding" the faces and numbering them as follows.




Clockwise turns of the six layers (top, left, front, right, back, and down) are represented by the following permutations.

Example

```
gap> cubegens := [
>   ( 1, 3, 8, 6)( 2, 5, 7, 4)( 9,33,25,17)(10,34,26,18)(11,35,27,19),
>   ( 9,11,16,14)(10,13,15,12)( 1,17,41,40)( 4,20,44,37)( 6,22,46,35),
>   (17,19,24,22)(18,21,23,20)( 6,25,43,16)( 7,28,42,13)( 8,30,41,11),
>   (25,27,32,30)(26,29,31,28)( 3,38,43,19)( 5,36,45,21)( 8,33,48,24),
>   (33,35,40,38)(34,37,39,36)( 3, 9,46,32)( 2,12,47,29)( 1,14,48,27),
>   (41,43,48,46)(42,45,47,44)(14,22,30,38)(15,23,31,39)(16,24,32,40)
> ];;
```

GAP computations analyzing this permutation group have been part of the announcements of GAP 3 releases. For a GAP 4 equivalent, see [Sch]. For more information and references (not GAP related) about Rubik's cube, see [Kölb].

### 6.15.1 BrowseRubiksCube

▷ `BrowseRubiksCube(`*`[format][,] [pi]`*`)` (function)

This function shows the model of the cube in a window.

If the argument *format* is one of the strings `"small"` or `"large"` then small or large cells are shown, the default is `"small"`.

The argument *pi* is the initial permutation of the faces, the default is a random permutation in the cube group, see (**Reference: Random**).

Supported user inputs are the keys T, L, F, R, B, and D for clockwise turns of the six layers, and the corresponding capital letters for counter-clockwise turns. If the terminal supports colors, according to the global variable `NCurses.attrs.has_colors` (2.2.1), the input S switches between a screen that shows only the colors of the faces and a screen that shows the numbers; the color screen is the default.

The return value is a record with the components `inputs` (a string describing the user inputs), `init`, and `final` (the initial and final permutation of the faces, respectively). (The `inputs` component can be used for the replay feature, see the example below.)

In the following example, a word in terms of the generators is used to initialize the browse table, and then the letters in this word are used as a series of input steps, except that in between, the display is switched once from colors to numbers and back.

Example

```
gap> choice:= List( [ 1 .. 30 ], i -> Random( [ 1 .. 6 ] ) );;
gap> input:= List( "tlfrbd", IntChar ){ choice };;
```

```
gap> BrowseData.SetReplay( Concatenation(
>        input{ [ 1 .. 20 ] },
>        "s",                    # switch to number display
>        input{ [ 21 .. 25 ] },
>        "s",                    # switch to color display
>        input{ [ 26 .. 30 ] },
>        "Q" ) );;               # quit the browse table
gap> BrowseRubiksCube( Product( cubegens{ choice } ) );;
gap> BrowseRubiksCube( "large", Product( cubegens{ choice } ) );;
gap> BrowseData.SetReplay( false );
```

*Implementation remarks*: The cube is implemented via a browse table, without row and column labels, with static header, dynamic footer, and individual `minyx` function. Only one mode is needed, and besides the standard actions for quitting the table, asking for help, and saving the current window contents, only the twelve moves and the switch between color and number display are admissible.

Switching between the two display formats is implemented via a function `work.Main`, so this relies on *not* caching the formatted cells in `work.main`.

Row and column separators of the browse table are whitespace of height and width one. The separating lines are drawn using an individual `SpecialGrid` function in the browse table. Note that the relevant cells do not form a rectangular array.

Some standard `NCurses.BrowseGeneric` (4.3.1) functionality, such as scrolling, selecting, and searching, are not available in this application.

The code can be found in the file `app/rubik.g` of the package.

## 6.16 Changing Sides

We consider a 5 by 5 board of squares filled with two types of stones, as follows. The square in the middle is left empty.

| | | | | |
|---|---|---|---|---|
| × | × | × | × | × |
| ○ | × | × | × | × |
| ○ | ○ | | × | × |
| ○ | ○ | ○ | ○ | × |
| ○ | ○ | ○ | ○ | ○ |

The aim of the game is to exchange the two types of stones via a sequence of single steps that move one stone to the empty position on the board. Only those moves are allowed that increase or decrease one coordinate by 2 and increase or decrease the other by 1; these are the allowed moves of the knight in chess.

This game has been part of the MacTutor system [OR00].

### 6.16.1 BrowseChangeSides

▷ `BrowseChangeSides()` (function)

This function shows the game board in a window.

Each move is encoded as a sequence of three arrow keys; there are 24 admissible inputs.

Example

```
gap> for entry in BrowseChangeSidesSolutions do
>       BrowseData.SetReplay( Concatenation( entry, "Q" ) );
>       BrowseChangeSides();
> od;
gap> BrowseData.SetReplay( false );
```

*Implementation remarks*: The game board is implemented via a browse table, without row and column labels, with static header, dynamic footer, and individual `minyx` function. Only one mode is needed, and besides the standard actions for quitting the table, asking for help, and saving the current window contents, only moves via combinations of the four arrow keys are admissible.

The separating lines are drawn using an individual `SpecialGrid` function in the browse table.

Some standard `NCurses.BrowseGeneric` (4.3.1) functionality, such as scrolling, selecting, and searching, are not available in this application.

The code can be found in the file `app/knight.g` of the package.

## 6.17 Sudoku

We consider a 9 by 9 board of squares. Some squares are initially filled with numbers from 1 to 9. The aim of the game is to fill the empty squares in such a way that each row, each column, and each of the marked 3 by 3 subsquares contains all numbers from 1 to 9. A *proper Sudoku game* is defined as one with a unique solution. Here is an example.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | 5 | | |
| | 1 | 5 | 4 | | 6 | | 2 | |
| 9 | | | | 5 | | 3 | | |
| 6 | | 4 | | | | | | |
| | | | 8 | | | | | |
| 8 | | | 9 | | | | 5 | 3 |
| | | | | | 5 | | | |
| | 4 | | | | 7 | | | 2 |
| | | 9 | 1 | | | 8 | | |

The Browse package contains functions to create, play and solve these games. There are basic command line functions for this, which we describe first, and there is a user interface `PlaySudoku` (6.17.8) which is implemented using the generic browse functionality described in Chapter 4.

### 6.17.1 Sudoku.Init

▷ `Sudoku.Init(`*`[arg]`*`)` (function)

**Returns:** A record describing a Sudoku board or `fail`.

This function constructs a record describing a Sudoku game. This is used by the other functions described below. There a several possibilities for the argument *arg*.

***arg* is a string**

The entries of a Sudoku board are numbered row-wise from 1 to 81. A board is encoded as a string as follows. If one of the numbers 1 to 9 is in entry *i* then the corresponding digit character is written in position *i* of the string. If an entry is empty any character, except `'1'` to `'9'` or `'|'` is written in position *i* of the string. Trailing empty entries can be left out. Afterwards `'|'`-characters can be inserted in the string (for example to mark line ends). Such strings can be used for *arg*.

***arg* is a matrix**

A Sudoku board can also be encoded as a 9 by 9-matrix, that is a list of 9 lists of length 9, whose (i,j)-th entry is the (i,j)-th entry of the board as integer if it is not empty. Empty entries of the board correspond to unbound entries in the matrix.

***arg* is a list of integers**

Instead of the matrix just described the argument can also be given by the concatenation of the rows of the matrix (so, a list of integers and holes).

Example

```
gap> game := Sudoku.Init(" 3   68  | 85  1 69|  97   53|      79 |\
>  6  47   |45  2    |89   2 1 | 4   8 7 | ");;
```

### 6.17.2 Sudoku.Place

▷ `Sudoku.Place(`*game*`, `*i*`, `*n*`)` (function)

▷ `Sudoku.Remove(`*game*`, `*i*`)` (function)

**Returns:** The changed *game*.

Here *game* is a record describing a Sudoku board, as returned by `Sudoku.Init` (6.17.1). The argument *i* is the number of an entry, counted row-wise from 1 to 81, and *n* is an integer from 1 to 9 to be placed on the board. These functions change *game*.

`Sudoku.Place` tries to place number *n* on entry *i*. It is an error if entry *i* is not empty. The number is not placed if *n* is already used in the row, column or subsquare of entry *i*. In this case the component `game.impossible` is bound.

`Sudoku.Remove` tries to remove the number placed on position *i* of the board. It does not change the board if entry *i* is empty, or if entry *i* was given when the board *game* was created. In the latter case `game.impossible` is bound.

Example

```
gap> game := Sudoku.Init(" 3   68  | 85  1 69|  97   53|      79 |\
>  6  47   |45  2    |89   2 1 | 4   8 7 | ");;
gap> Sudoku.Place(game, 1, 3);; # 3 is already in first row
gap> IsBound(game.impossible);
true
gap> Sudoku.Place(game, 1, 2);; # 2 is not in row, col or subsquare
gap> IsBound(game.impossible);
false
```

### 6.17.3 Sudoku.RandomGame

▷ `Sudoku.RandomGame(`*[seed]*`)` (function)

**Returns:** A pair `[str, seed]` of string and seed.

The optional argument *seed*, if given, must be an integer. If not given some random integer from the current GAP session is used. This function returns a random proper Sudoku game, where the board is described by a string `str`, as explained in `Sudoku.Init` (6.17.1). With the same *seed* the same board is returned.

The games computed by this function have the property that after removing any given entry the puzzle does no longer have a unique solution.

Example

```
gap> Sudoku.RandomGame(5833750);
[ " 1         2     43  2   68   72    8     6 2   1 9 8  8 3   9     \
47 3   7  18  ", 5833750 ]
gap> last = Sudoku.RandomGame(last[2]);
true
```

### 6.17.4 Sudoku.SimpleDisplay

▷ `Sudoku.SimpleDisplay(`*game*`)` (function)

Displays a Sudoku board on the terminal. (But see `PlaySudoku` (6.17.8) for a fancier interface.)

Example

```
gap> game := Sudoku.Init(" 3   68  | 85  1 69|  97   53|      79 |\
>  6  47   |45  2    |89   2 1 | 4   8 7 | ");;
gap> Sudoku.SimpleDisplay(game);
 3 |  6|8
 85|  1| 69
  9|7  | 53
-----------
   |   |79
 6 | 47|
45 | 2 |
-----------
89 |  2| 1
 4 |  8| 7
   |   |
```

### 6.17.5 Sudoku.DisplayString

▷ `Sudoku.DisplayString(`*game*`)` (function)

The string returned by this function can be used to display the Sudoku board *game* on the terminal, using `PrintFormattedString` (**GAPDoc: PrintFormattedString**). The result depends on the value of `GAPInfo.TermEncoding`.

Example

```
gap> game := Sudoku.Init(" 3   68  | 85  1 69|  97   53|      79 |\
>  6  47   |45  2    |89   2 1 | 4   8 7 | ");;
gap> str:= Sudoku.DisplayString( game );;
gap> PrintFormattedString( str );
                    +===+===+===+===+===+===+===+===+===+
                    |   | 3 |   |   |   | 6 | 8 |   |   |
                    +---+---+---+---+---+---+---+---+---+
```

```
|   | 8 | 5 |   |   | 1 |   | 6 | 9 |
+---+---+---+---+---+---+---+---+---+
|   |   | 9 | 7 |   |   |   | 5 | 3 |
+===+===+===+===+===+===+===+===+===+
|   |   |   |   |   |   | 7 | 9 |   |
+---+---+---+---+---+---+---+---+---+
|   | 6 |   |   | 4 | 7 |   |   |   |
+---+---+---+---+---+---+---+---+---+
| 4 | 5 |   |   | 2 |   |   |   |   |
+===+===+===+===+===+===+===+===+===+
| 8 | 9 |   |   |   | 2 |   | 1 |   |
+---+---+---+---+---+---+---+---+---+
|   | 4 |   |   |   | 8 |   | 7 |   |
+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |
+===+===+===+===+===+===+===+===+===+
```

### 6.17.6 Sudoku.OneSolution

▷ `Sudoku.OneSolution(`*game*`)` (function)

**Returns:** A completed Sudoku board that solves *game*, or `fail`.

Here *game* must be a Sudoku board as returned by `Sudoku.Init` (6.17.1). It is not necessary that *game* describes a proper Sudoku game (has a unique solution). It may have several solutions, then one random solution is returned. Or it may have no solution, then `fail` is returned.

Example

```
gap> Sudoku.SimpleDisplay(Sudoku.OneSolution(Sudoku.Init("  3")));
493|876|251
861|542|739
527|193|648
-----------
942|618|573
156|739|482
738|425|916
-----------
289|354|167
375|961|824
614|287|395
```

### 6.17.7 Sudoku.UniqueSolution

▷ `Sudoku.UniqueSolution(`*game*`)` (function)

**Returns:** A completed Sudoku board that solves *game*, or `false`, or `fail`.

Here *game* must be a Sudoku board as returned by `Sudoku.Init` (6.17.1). It is not necessary that *game* describes a proper Sudoku game. If it has several solutions, then `false` is returned. If it has no solution, then `fail` is returned. Otherwise a board with the unique solution is returned.

Example

```
gap> s := "      5  | 154 6 2 |9   5 3  |6 4      |   8     |8  9   53\
> |     5   | 4   7  2|  91  8  ";;
gap> sol := Sudoku.UniqueSolution(Sudoku.Init(s));;
gap> Sudoku.SimpleDisplay(sol);
```

```
438|219|576
715|436|928
962|758|314
-----------
694|573|281
153|862|749
827|941|653
-----------
281|695|437
546|387|192
379|124|865
```

### 6.17.8 PlaySudoku

▷ `PlaySudoku(`*`[arg]`*`)` (function)

**Returns:** A record describing the latest status of a Sudoku board.

This function allows one to solve Sudoku puzzles interactively. There are several possibilities for the optional argument *arg*. It can either be a string, matrix or list of holes and integers as described in `Sudoku.Init` (6.17.1), or a board as returned by `Sudoku.Init` (6.17.1). Furthermore *arg* can be an integer or not be given, in that case `Sudoku.RandomGame` (6.17.3) is called to produce a random game.

The usage of this function is self-explanatory, pressing the ? key displays a help screen. Here, we mention two keys with a particular action: Pressing the H key you get a hint, either an empty entry is filled or the program tells you that there is no solution (so you must delete some entries and try others). Pressing the S key the puzzle is solved by the program or it tells you that there is no or no unique solution.

*Implementation remarks*: The game board is implemented via a browse table, without row and column labels, with static header, dynamic footer, and individual `minyx` function. Two modes are supported, with the standard actions for quitting the table and asking for help; one cell is selected in each mode. The first mode provides actions for moving the selected cell via arrow keys, for changing the value in the selected cell, for getting a hint or the (unique) solution. (Initial entries of the matrix cannot be changed via user input. They are shown in boldface.) The second mode serves for error handling: When the user enters an invalid number, i. e., a number that occurs already in the current row or column or subsquare, then the application switches to this mode, which causes that a message is shown in the footer, and the invalid entry is shown in red and blinking; similarly, error mode is entered if a hint or solution does not exist.

The separating lines are drawn using an individual `SpecialGrid` function in the browse table, since they cannot be specified within the generic browse table functions.

Some standard `NCurses.BrowseGeneric` (4.3.1) functionality, such as scrolling, selecting, and searching, are not available in this application.

The code can be found in the file `app/sudoku.g` of the package.

### 6.17.9 Sudoku.HTMLGame

▷ `Sudoku.HTMLGame(`*`game`*`)` (function)

▷ `Sudoku.LaTeXGame(`*`game`*`)` (function)

**Returns:** A string with HTML or LaTeX code, respectively.

The argument of these functions is a record describing a Sudoku game. These functions return code for including the current status of the board into a webpage or a LaTeX document.

## 6.18 Managing simple Workflows

The idea behind the function `BrowseWizard` (6.18.1) is that one wants to collect interactively information from a user, by asking a series of questions. Default answers for these questions can be provided, perhaps depending on the answers to earlier questions. The questions and answers are shown in a browse table, the current question is highlighted, and this selection is automatically moved to the next question after a valid answer has been entered. One may move up in the table, in order to change previous answers, but one can move down only to the first unanswered question. When the browse table gets closed (by submitting or canceling), a record with the collected information is returned.

### 6.18.1 BrowseWizard

▷ `BrowseWizard(`*`data`*`)` (function)

**Returns:** a record.

The argument *`data`* must be a record with the components `steps` (a list of records, each representing one step in the questionnaire) and `defaults` (a record). The component `header`, if present, must be a string that is used as a header line; the default for it is `"BrowseWizard"`.

`BrowseWizard` opens a browse table whose rows correspond to the entries of *`data`*`.steps`. The components of *`data`*`.defaults` are used as default values if they are present.

Beginning with the first entry, the user is asked to enter information, one record component per entry; this may be done by entering some text, by choosing keys from a given list of choices, or by editing a list of tabular data. Then one can go to the next step by hitting the ArrowDown key (or by entering D), and edit this step by hitting the Enter key. One can also go back to previous steps and edit them again.

Some steps may be hidden from the user, depending on the information that has been entered for the previous steps. The hide conditions are evaluated after each step.

An implementation of a questionnaire is given by `BrowseData.ChooseSimpleGroupQuestions`, which is used in the following example. The idea is to choose the description of a finite simple group by entering first the type (cyclic, alternating, classical, exceptional, or sporadic) and then the relevant parameter values. The information is then evaluated by `BrowseData.InterpretSimpleGroupDescription`, which returns a description that fits to the return values of `IsomorphismTypeInfoFiniteSimpleGroup` (**Reference: IsomorphismTypeInfoFiniteSimpleGroup**). For example, this function identifies the group $\mathrm{PSL}(4,2)$ as $A_8$.)

Example

```
gap> d:= [ NCurses.keys.DOWN ];;  r:= [ NCurses.keys.RIGHT ];;
gap> c:= [ NCurses.keys.ENTER ];;
gap> BrowseData.SetReplay( Concatenation(
>        c,             # confirm the initial message
>        d,             # enter the first step
>        d, d,          # go to the choice of classical groups
>        c,             # confirm this choice
>        c,             # enter the next step
>        d, d,          # go to the choice of unitary groups
>        c,             # confirm this choice
>        c,             # enter the next step
```

```
>        "5", c,        # enter the dimension and confirm
>        c,             # enter the next step
>        "3", c,        # enter the field size and confirm
>        c ) );         # confirm all choices (closes the table)
gap> res:= BrowseWizard( rec(
>      steps:= BrowseData.ChooseSimpleGroupQuestions,
>      defaults:= rec(),
>      header:= "Choose a finite simple group" ) );;
gap> BrowseData.SetReplay( false );
gap> BrowseData.InterpretSimpleGroupDescription( res );
rec( parameter := [ 4, 3 ], requestedname := "U5(3)", series := "2A",
  shortname := "U5(3)" )
```

The supported components of each entry in *data*`.steps` are as follows.

`key` a string, the name of the component of the result record that gets bound for this entry.

`description`
a string describing what information shall be entered.

`type`
one of `"editstring"`, `"edittable"`, `"key"`, `"keys"`, `"ok"`, `"okcancel"`, `"submitcancelcontinue"`.

`keys` **(only if** `type` **is** `"key"` **or** `"keys"`**)**
either the list of pairs `[ key, alias ]` such that the user shall choose from the list of `key` values (strings), and the `alias` values (any GAP object) corresponding to the chosen values are entered into the result record, or a function that takes *steps* and the current result record as its arguments and returns the desired list of pairs.

`validation` **(optional)**
a function that takes *steps*, the current result record, and a result candidate for the current step as its arguments; it returns `true` if the result candidate is valid, and a string describing the reason for the failure otherwise.

`default` **(optional)**
depending on the `type` value, the alias part(s) of the chosen key(s) or the string or the list of data records, or alternatively a function that takes *steps* and the current result record as its arguments and returns the desired value. If the `key` component of *data*`.defaults` is bound and valid (according to the `validation` function) then this value is taken as the default; otherwise, the `default` component of the entry is taken as the default.

`isVisible` **(optional)**
a function that takes *steps* and the current result record as its arguments and returns `true` if the step shall be visible, and `false` otherwise,

If the `type` value of a step is `"edittable"` then also the following components are mandatory.

`list`
the current list of records to be edited; only strings are supported as the values of the record components.

`mapping`
: a list of pairs `[ component, label ]` such that `component` is the name of a component in the entries in `list`, and `label` is the label shown in the dialog box for editing the record.

`choices` **(optional)**
: a list of records which can be added to `list`.

`rectodisp`
: a function that takes a record from `list` and returns a string that is shown in the browse table.

`title`
: a string, the header line of the dialog box for editing an entry.

The code of `BrowseWizard`, `BrowseData.ChooseSimpleGroupQuestions`, and `BrowseData.InterpretSimpleGroupDescription` can be found in the file `app/wizard.g` of the package.

## 6.19 Utility for GAP Demos

This application can be used with GAP if the user interface has `readline` support. The purpose is to simplify the typing during a demonstration of GAP commands.

The file format to specify GAP code for a demonstration is very simple: it contains blocks of lines with GAP input, separated by lines starting with the sequence `#%`. Comments in such a file can be added to one or several lines starting with `#%`. Here is the content of an example file `demo.demo`:

```
#% Add comments after #% characters at the beginning of a line.
#% A comment can have several lines.
#% Here is a multi-line input block:
g := MathieuGroup(11);;
cl := ConjugacyClasses(g);
#% Calling a help page
?MathieuGroup
#% The next line contains a comment in the GAP session:
a := 12;; b := 13;; # assign two numbers
#%
a*b;
#%
```

(Single `%` in the beginning of a line will also work as separators.)

A demonstration can be loaded into a GAP session with the command

### 6.19.1 LoadDemoFile

▷ `LoadDemoFile(`*`demoname, demofile[, singleline]`*`)` (function)

**Returns:** Nothing.

This function loads a demo file in the format described above. The argument *`demoname`* is a string containing a name for the demo, and *`demofile`* is the file name containing the demo.

If the optional argument *singleline* is given and its value is true, the demo behaves differently with respect to input blocks that span several lines. By default full blocks are treated as a single input line for readline (maybe spanning several physical lines in the terminal). If *singleline* is true then all input lines of a block except the last one are sent to GAP and are evaluated automatically before the last line of the block is displayed.

Example

```
gap> dirs := DirectoriesPackageLibrary("Browse");;
gap> demofile := Filename(dirs, "../app/demo.demo");;
gap> if IsBound(GAPInfo.UseReadline) and GAPInfo.UseReadline = true then
>   LoadDemoFile("My first demo", demofile);
>   LoadDemoFile("My first demo (single lines)", demofile, true);
> fi;
```

Many demos can be loaded at the same time. They are used with the PAGEDOWN and PAGEUP keys.

The PAGEUP key leads to a (Browse) menu which allows one to choose a demo to start (if several are loaded), to stop a demo or to move to another position in the current demo (e.g., to go back to a previous point or to skip part of a demo).

The next input block of the current demo is copied into the current input line of the GAP session by pressing the PAGEDOWN key. This line is not yet sent to GAP, use the RETURN key if you want to evaluate the input. (You can also still edit the input line before evaluation.)

So, in the simplest case a demo can be done by just pressing PAGEDOWN and RETURN in turns. But it is always possible to type extra input during a demo by hand or to change the input lines from the demo file before evaluation. It is no problem if commands are interrupted by CTRL-C. During a demo you are in a normal GAP session, this application only saves you some typing. The input lines from the demo are put into the history of the session as if they were typed by hand.

Try it yourself with the two demos loaded in the example. This also shows the different behaviour between default and single line mode.

# Appendix A

# Some Tools for Database Handling

Two aims of the tools described in this appendix are

- speeding up selection functions such as `AllCharacterTableNames` (**CTblLib: AllCharacterTableNames**) for certain data libraries of GAP (with not too many entries), in the sense that users can extend the list of attributes that are treated in a special way
- and a programmatic extension for rendering overviews of information about the contents of databases, using `BrowseTableFromDatabaseIdEnumerator` (A.2.2).

The GAP objects introduced for that are *database id enumerators* (see A.1.1) and *database attributes* (see A.1.2).

Contrary to the individual interfaces to the GAP manuals (see Section 6.6), the GAP bibliography (see Section 6.7), and the overviews of GAP packages, GAP methods, and Conway polynomials available in GAP (see Section 6.11), the approach that will be described here assumes a special way to access database entries. Thus it depends on the structure of a given database whether the tools described here are useful, or whether an individual interface fits better. Perhaps the example shown in Section A.3 gives an impression what is possible.

## A.1 GAP Objects for Database Handling

### A.1.1 Database Id Enumerators

A *database id enumerator* is a record `r` with at least the following components.

`identifiers`
: a list of "identifiers" of the database entries, which provides a bijection with these entries,

`entry`
: a function that takes `r` and an entry in the `identifiers` list, and returns the corresponding database entry,

`attributes`
: the record whose components are the database attribute records (see Section A.1.2) for `r`; this components is automatically initialized when `r` is created with `DatabaseIdEnumerator` (A.1.4); database attributes can be entered with `DatabaseAttributeAdd` (A.1.5).

If the `identifiers` list may change over the time (because the database is extended or corrected) then the following components are supported. They are used by `DatabaseIdEnumeratorUpdate` (A.1.7).

`version`
: a GAP object that describes the version of the `identifiers` component, this can be for example a string describing the time of the last change (this time need not coincide with the time of the last update); the default value (useful only for the case that the `identifiers` component is never changed) is an empty string,

`update`
: a function that takes `r` as its argument, replaces its `identifiers` and `version` values by up-to-date versions if necessary (for example by downloading the data), and returns `true` or `false`, depending on whether the update process was successful or not; the default value is `ReturnTrue` (**Reference: ReturnTrue**),

The following component is optional.

`isSorted`
: `true` means that the `identifiers` list is sorted w.r.t. GAP's ordering `\<`; the default is `false`.

The idea behind database id enumerator objects is that such an object defines the set of data covered by database attributes (see Section A.1.2), it provides the mapping between identifiers and the actual entries of the database, and it defines when precomputed data of database attributes are outdated.

### A.1.2 Database Attributes

A *database attribute* is a record `a` whose components belong to the aspects of *defining* the attribute, *accessing* the attribute's data, *computing* (and recomputing) data, *storing* data on files, and *checking* data. (Additional parameters used for creating browse table columns from database attributes are described in Section A.2.1.)

The following components are *defining*, except `description` they are mandatory.

`idenumerator`
: the database id enumerator to which the attribute `a` is related,

`identifier`
: a string that identifies `a` among all database attributes for the underlying database id enumerator (this is used by `BrowseTableFromDatabaseIdEnumerator` (A.2.2) and when the data of `a` are entered with `DatabaseAttributeSetData` (A.1.11), for example when precomputed values are read from a file),

`description`
: a string that describes the attribute in human readable form (currently just for convenience, the default is an empty string).

The following components are used for *accessing* data. Except `type`, they are optional, but enough information must be provided in order to make the database attribute meaningful. If an individual `attributeValue` function is available then this function decides what is needed; for the default

function DatabaseAttributeValueDefault (A.1.6), at least one of the components name, data, datafile must be bound (see DatabaseAttributeValueDefault (A.1.6) for the behaviour in this case).

type
: one of the strings "values" or "pairs"; the format of the component data is different for these cases,

name
: if bound, a string that is the name of a GAP function such that the database attribute encodes the values of this function for the database entries; besides the computation of attribute values on demand (see DatabaseAttributeValueDefault (A.1.6)), this component can be used by selection functions such as OneCharacterTableName (**CTblLib: OneCharacterTableName**) or AllCharacterTableNames (**CTblLib: AllCharacterTableNames**), which take GAP functions and prescribed return values as their arguments –of course these functions must then be prepared to deal with database attributes.

data
: if bound, the data for this attribute; if the component type has the value "values" then the value is a list, where the entry at position *i*, if bound, belongs to the *i*-th entry of the identifiers list of idenumerator; if type is "pairs" then the value is a record with the components automatic and nonautomatic, and the values of these components are lists such that each entry is a list of length two whose first entry occurs in the identifiers list of a.idenumerator and whose second entry encodes the corresponding attribute value,

datafile
: if bound, the absolute name of a file that contains the data for this attribute,

attributeValue
: a function that takes a and an identifiers entry of its idenumerator value, and returns the attribute value for this identifier; typically this is *not* a table cell data object that can be shown in a browse table, cf. the viewValue component; the default is DatabaseAttributeValueDefault (A.1.6) (Note that using individual attributeValue functions, one can deal with database attributes independent of actually stored data, for example without precomputed values, such that the values are computed on demand and afterwards are cached.),

dataDefault
: a GAP object that is regarded as the attribute value for those database entries for which data, datafile, and name do not provide values; the default value is an empty string "",

eval
: if this component is bound, the value is assumed to be a function that takes a and a value from its data component, and returns the actual attribute value; this can be useful if one does not want to create all attribute values in advance, because this would be space or time consuming; another possible aspect of the eval component is that it may be used to strip off comments that are perhaps contained in data entries,

`isSorted`
: if this component is bound to `true` and if `type` is `"pairs"` then it is assumed that the two lists in the `data` record of `a` are sorted w.r.t. GAP's ordering `\<`; the default is `false`,

The following optional components are needed for *computing* (or recomputing) data with `DatabaseAttributeCompute` (A.1.8). This is useful mainly for databases which can change over the time.

`version`
: the GAP object that is the `version` component of the `idenumerator` component at the time when the stored data were entered; this value is used by `DatabaseIdEnumeratorUpdate` (A.1.7) for deciding whether the attribute values are outdated; if `a.datafile` is bound then it is assumed that the `version` component is set when this file is read, for example in the function `DatabaseAttributeSetData` (A.1.11),

`update`
: a function that takes `a` as its argument, adjusts its data components to the current values of `a.dbidenum` if necessary, sets the `version` component to that of `a.dbidenum`, and returns `true` or `false`, depending on whether the update process was successful or not; the default value is `ReturnTrue` (**Reference: ReturnTrue**),

`neededAttributes`
: a list of attribute `identifier` strings such that the values of these attributes are needed in the computations for the current one, and therefore these should be updated/recomputed in advance; it is assumed that the `neededAttributes` components of all database attributes of `a.idenumerator` define a partial ordering; the default is an empty list,

`prepareAttributeComputation`
: a function with argument `a` that must be called before the computations for the current attribute are started; the default value is `ReturnTrue` (**Reference: ReturnTrue**),

`cleanupAfterAttibuteComputation`
: a function with argument `a` that must be called after the computations for the current attribute are finished; the default value is `ReturnTrue` (**Reference: ReturnTrue**), and

`create`
: a function that takes a database attribute and an entry in the `identifiers` list of its database id enumerator, and returns either the entry that shall be stored in the `data` component, as the value for the given identifier (if this value shall be stored in the `data` component of `a`) or the `dataDefault` component of `a` (if this value shall *not* be stored); in order to get the actual attribute value, the `eval` function of `a`, if bound, must be called with the return value. This function may assume that the `prepareAttributeComputation` function has been called in advance, and that the `cleanupAfterAttibuteComputation` function will be called later. The `create` function is *not* intended to compute an individual attribute value on demand, use a `name` component for that. (A stored `name` function is used to provide a default for the `create` function; without `name` component, there is no default for `create`.)

The following optional component is needed for *storing* data on files.

`string`
: if bound, a function that takes the pair consisting of an identifier and the return value of the `create` function for this identifier, and returns a string that represents this value when the data are printed to a file with `DatabaseAttributeString` (A.1.9); the default function returns the `String` (**Reference: String**) value of the second argument.

The following optional component is needed for *checking* stored data.

`check`
: a function that takes a string that occurs in the `identifiers` list of the `idenumerator` record, and returns `true` if the attribute value stored for this string is reasonable, and something different from `true` if an error was detected. (One could argue that these tests can be performed also when the values are computed, but consistency checks may involve several entries; besides that, checking may be cheaper than recomputing.)

### A.1.3 How to Deal with Database Id Enumerators and Database Attributes

The idea is to start with a database id enumerator (see A.1.1), constructed with `DatabaseIdEnumerator` (A.1.4), and to define database attributes for it (see A.1.2), using `DatabaseAttributeAdd` (A.1.5). The attribute values can be precomputed and stored on files, or they are computed when the attribute gets defined, or they are computed on demand.

The function `DatabaseAttributeCompute` (A.1.8) can be used to "refresh" the attribute values, that is, all values or selected values can be recomputed; this can be necessary for example when the underlying database id enumerator gets extended.

In data files, the function `DatabaseAttributeSetData` (A.1.11) can be used to fill the `data` component of the attribute. The contents of a data file can be produced with `DatabaseAttributeString` (A.1.9).

### A.1.4 DatabaseIdEnumerator

▷ `DatabaseIdEnumerator(`*`arec`*`)` (function)

**Returns:** a shallow copy of the record *`arec`*, extended by default values.

For a record *`arec`*, `DatabaseIdEnumerator` checks whether the mandatory components of a database id enumerator (see Section A.1.1) are present, initializes the `attributes` component, sets the defaults for unbound optional components (see A.2.1), and returns the resulting record.

A special database attribute (see Section A.1.2) with `identifier` value `"self"` is constructed automatically for the returned record by `DatabaseIdEnumerator`; its `attributeValue` function simply returns its second argument (the identifier). The optional components of this attribute are derived from components of the database id enumerator, so these components (see A.2.1) are supported for *`arec`*. A typical use of the `"self"` attribute is to provide the first column in browse tables constructed by `BrowseTableFromDatabaseIdEnumerator` (A.2.2).

### A.1.5 DatabaseAttributeAdd

▷ `DatabaseAttributeAdd(`*`dbidenum, arec`*`)` (function)

For a database id enumerator *`dbidenum`* and a record *`arec`*, `DatabaseAttributeAdd` checks whether the mandatory components of a database attribute, except `idenumerator`, are present in *`arec`*

(see Section A.1.2), sets the idenumerator component, and sets the defaults for unbound optional components (see A.2.1).

### A.1.6 DatabaseAttributeValueDefault

▷ DatabaseAttributeValueDefault(*attr, id*) (function)

**Returns:** the value of the database attribute *attr* at *id*.

For a database attribute *attr* and an entry *id* of the identifiers list of the underlying database id enumerator, DatabaseAttributeValueDefault takes the data entry for *id*, applies the eval function of *attr* to it if available and returns the result.

So the question is how to get the data entry.

First, if the data component of *attr* is not bound then the file given by the datafile component of *attr*, if available, is read, and otherwise DatabaseAttributeCompute (A.1.8) is called; afterwards it is assumed that the data component is bound.

The further steps depend on the type value of *attr*.

If the type value of *attr* is "pairs" then the data entry for *id* is either contained in the automatic or in the nonautomatic list of *attr*.data, or it is given by the dataDefault value of *attr*. (So a perhaps available name function is *not* used to compute the value for a missing data entry.)

If the type value of *attr* is "values" then the data entry for *id* is computed as follows. Let $n$ be the position of *id* in the identifiers component of the database id enumerator. If the $n$-th entry of the data component of *attr* is bound then take it; otherwise if the name component is bound then apply it to *id* and take the return value; otherwise take the dataDefault value.

If one wants to introduce a database attribute where this functionality is not suitable then another –more specific– function must be entered as the component attributeValue of such an attribute.

### A.1.7 DatabaseIdEnumeratorUpdate

▷ DatabaseIdEnumeratorUpdate(*dbidenum*) (function)

**Returns:** true or false.

For a database id enumerator *dbidenum* (see Section A.1.1), DatabaseIdEnumeratorUpdate first calls the update function of *dbidenum*. Afterwards, the update components of those of its attributes records are called for which the version component differs from that of *dbidenum*.

The order in which the database attributes are updates is determined by the neededAttributes component.

The return value is true if all these functions return true, and false otherwise.

When DatabaseIdEnumeratorUpdate has returned true, the data described by *dbidenum* and its database attributes are consistent and up to date.

### A.1.8 DatabaseAttributeCompute

▷ DatabaseAttributeCompute(*dbidenum, attridentifier[, what]*) (function)

**Returns:** true or false.

This function returns false if *dbidenum* is not a database id enumerator, or if it does not have a database attribute with identifier value *attridentifier*, or if this attribute does not have a create function.

Otherwise the prepareAttributeComputation function is called, the data entries for the database attribute are (re)computed, the cleanupAfterAttibuteComputation function is called, and true is returned.

The optional argument *what* determines which values are computed. Admissible values are

"all"
: all identifiers entries of *dbidenum*,

"automatic" **(the default)**
: the same as "all" if the type value of the database attribute is "values", otherwise only the values for the "automatic" component are computed,

"new"
: stored values are not recomputed.

### A.1.9 DatabaseAttributeString

▷ DatabaseAttributeString(*idenum, idenumname, attridentifier, format*) (function)

**Returns:** a string that describes the values of the attribute.

Let *idenum* be a database id enumerator (see Section A.1.1), *idenumname* be a string that denotes the variable to which the enumerator is bound, *attridentifier* be the name of an attribute of type "pairs", and *format* be one of "GAP", "JSON". DatabaseAttributeString returns a string that can be used to set the attribute values, using DatabaseAttributeLoadData (A.1.10). In the "JSON" case, it is not checked whether the string function of the attribute creates valid JSON (e.g., whether the lists are dense).

### A.1.10 DatabaseAttributeLoadData

▷ DatabaseAttributeLoadData(*attr*) (function)

**Returns:** true or false.

If the data of the attribute *attr* are stored in a file then this function loads the data. The data file is expected to be either in JSON format (which can be produced with DatabaseAttributeString (A.1.9), with fourth argument the string "JSON") and have filename extension .json, or to contain a call to DatabaseAttributeSetData (A.1.11) such that reading the file with Read (**Reference: Read**) sets the data.

### A.1.11 DatabaseAttributeSetData

▷ DatabaseAttributeSetData(*dbidenum, attridentifier, version, data*) (function)

Let *dbidenum* be a database id enumerator (see Section A.1.1), *attridentifier* be a string that is the identifier value of a database attribute of *dbidenum*, *data* be the data list or record for the database attribute (depending on its type value), and *version* be the corresponding version value.

DatabaseAttributeSetData sets the data and version components of the attribute. This function is called when data files are loaded with DatabaseAttributeLoadData (A.1.10).

## A.2 Using Database Attributes for Browse Tables

### A.2.1 Browse Relevant Components of Database Attributes

The following optional components of database id enumerators and database attributes are used by `BrowseTableFromDatabaseIdEnumerator` (A.2.2).

`viewLabel`
: if bound, a table cell data object (see `BrowseData.IsBrowseTableCellData` (4.2.1)) that gives a *short* description of the attribute, which is used as the column label in browse tables created with `BrowseTableFromDatabaseIdEnumerator` (A.2.2); the default for database attributes is the `name` component, if bound, and otherwise the `identifier` component; the default for database id enumerators is the string `"name"`,

`viewValue`
: if bound, a function that takes the output of the `attributeValue` function and returns a table cell data object (see `BrowseData.IsBrowseTableCellData` (4.2.1)) that is used as the entry of the corresponding column in browse tables created with `BrowseTableFromDatabaseIdEnumerator` (A.2.2); the default is `String` (**Reference: String**),

`viewSort`
: if bound, a comparison function that takes two database attribute values and returns `true` if the first value is regarded as smaller than the second when the column corresponding to the attribute in the browse table constructed by `BrowseTableFromDatabaseIdEnumerator` (A.2.2) gets sorted, and `false` otherwise; the default is GAP's `\<` operation,

`sortParameters`
: if bound, a list in the same format as the last argument of `BrowseData.SetSortParameters`, which is used for the column corresponding to the attribute in the browse table constructed by `BrowseTableFromDatabaseIdEnumerator` (A.2.2); the default is an empty list,

`widthCol`
: if bound, the width of the column in the browse table constructed by `BrowseTableFromDatabaseIdEnumerator` (A.2.2); if a column width is prescribed this way then the function stored in the `attributeValue` component must return either a list of attribute lines that fit into the column or a plain string (which then gets formatted as required); there is no default for this component, meaning that the column width is computed as the maximum of the widths of the column label and of all entries in the column if no value is bound,

`align`
: if bound, the alignment of the values in the column of the browse table constructed by `BrowseTableFromDatabaseIdEnumerator` (A.2.2); admissible values are substrings of `"bclt"`, see `BrowseData.IsBrowseTableCellData` (4.2.1); the default is right and vertically centered, but note that if the `viewValues` function returns a record (see `BrowseData.IsBrowseTableCellData` (4.2.1)) then the alignment prescribed by this record is preferred,

`categoryValue`
: if bound, a function that is similar to the `viewValue` component but may return a different value; for example if the column in the browse table belongs to a property and the `viewValue` function returns something like `"+"` or `"-"`, it may be useful that the category rows show a textual description of the property values; the default value is the `viewValue` component; if the value is a record then its `rows` component is taken for forming category rows, if the value is an attribute line (see `NCurses.IsAttributeLine` (2.2.3)) then there is exactly this category row, and otherwise the value is regarded as a list of attribute lines, which is either concatenated to one category row or turned into individual category rows, depending on the `sortParameters` value.

### A.2.2 BrowseTableFromDatabaseIdEnumerator

▷ `BrowseTableFromDatabaseIdEnumerator(`*`dbidenum, labelids, columnids[, header[, footer[, choice]]]`*`)` (function)

**Returns:** a record that can be used as the input of `NCurses.BrowseGeneric` (4.3.1).

For a database id enumerator *`dbidenum`* (see Section A.1.1) and two lists *`labelids`* and *`columnids`* of `identifier` values of database attributes stored in *`dbidenum`*, `BrowseTableFromDatabaseIdEnumerator` returns a browse table (see `BrowseData.IsBrowseTable` (4.2.3)) whose columns are given by the values of the specified database attributes. The columns listed in *`labelids`* are used to provide row label columns of the browse table, the columns listed in *`columnids`* yield main table columns. *`columnids`* must be nonempty.

If the optional arguments *`header`* and *`footer`* are given then they must be lists or functions or records that are admissible for the `header` and `footer` components of the `work` record of the browse table, see `BrowseData.IsBrowseTable` (4.2.3).

The optional argument *`choice`*, if given, must be a subset of *`dbidenum`*`.identifiers`. The rows of the returned browse table are then restricted to this subset.

The returned browse table does not support "Click" events or return values.

## A.3 Example: Database Id Enumerators and Database Attributes

As an example for the functions introduced in this appendix, we introduce the *database of small integers*. For that, we fix a positive integer $n$ and consider the integers from 1 to $n$ as the entries of our database. Using these integers as their own identifiers, we construct the database id enumerator.

Example

```
gap> n:= 100;;
gap> smallintenum1:= DatabaseIdEnumerator( rec(
>      identifiers:= [ 1 .. n ],
>      entry:= function( dbidenum, id ) return id; end,
>    ) );;
```

Examples of attributes for this database are the properties whether or not an integer is a prime or a prime power. There are global GAP functions `IsPrimeInt` (**Reference: IsPrimeInt**) and `IsPrimePowerInt` (**Reference: IsPrimePowerInt**) for computing these properties, so we can define these database attributes via a `name` component; we choose `"values"` as the `type` value, so the values (`true` or `false`) are stored in a list of length $n$ for each of the two database attributes.

Example

```
gap> DatabaseAttributeAdd( smallintenum1, rec(
>      identifier:= "primes",
>      type:= "values",
>      name:= "IsPrimeInt",
>    ) );
gap> DatabaseAttributeAdd( smallintenum1, rec(
>      identifier:= "prime powers",
>      type:= "values",
>      name:= "IsPrimePowerInt",
>    ) );
```

Similarly, we consider the prime factors as a database attribute.

Example

```
gap> DatabaseAttributeAdd( smallintenum1, rec(
>      identifier:= "factors",
>      type:= "values",
>      name:= "Factors",
>    ) );
```

Another example of an attribute of integers is the residue modulo 11. We do not want to introduce a global GAP function for computing the value, so we use the `create` component in order to define the attribute; again, the values (integers from 0 to 10) are stored in a list of length $n$.

Example

```
gap> DatabaseAttributeAdd( smallintenum1, rec(
>      identifier:= "residue mod 11",
>      type:= "values",
>      create:= function( attr, id ) return id mod 11; end,
>    ) );
```

Some integers are values of `Factorial` (**Reference: Factorial**), and we want to record this information and show it in a browse table. For most integers, nothing is stored and shown for this attribute, so we choose the `type` value `"pairs"` and precompute the information for the `data` component. (The default for the `dataDefault` component is an empty string, which is fine; so we need not prescribe this component.)

Example

```
gap> factorialdata:= function( n )
>      local result, i, f;
>      result:= [];  i:= 1;  f:= 1;;
>      while f <= n do
>        Add( result, [ f, i ] );  i:= i + 1;  f:= f * i;
>      od;
>      return result;
>    end;;
gap> DatabaseAttributeAdd( smallintenum1, rec(
>      identifier:= "inverse factorial",
>      type:= "pairs",
>      data:= rec( automatic:= factorialdata( n ), nonautomatic:= [] ),
>      isSorted:= true,
>    ) );
```

We use this setup for creating a browse table. The integers are shown as the first column, using the "self" attribute. This attribute can be used as a column of row labels (useful if we want to keep the column visible when one scrolls the table to the right) or as a column in the main table (useful if we want to search for the values); here we choose the former possibility.

Example

```
gap> t1:= BrowseTableFromDatabaseIdEnumerator( smallintenum1,
>           [ "self" ],
>           [ "primes", "prime powers", "factors", "residue mod 11",
>             "inverse factorial" ] );;
```

The following session shows some of the features of the browse table.

Example

```
gap> nop:= [ 14, 14, 14, 14, 14, 14 ];;  # ‘‘do nothing’’
gap> sample_session:= Concatenation(
>         # categorize by the first column, expand categories, wait, reset
>         nop, "scsc", nop, "X", nop, "!",
>         # sort the residue column, wait, reset
>         "scrrrso", nop, "!",
>         # categorize by the inverse factorial column
>         "rscsrdx", nop, "!",
>         # and quit the application
>         "qQ" );;
gap> BrowseData.SetReplay( sample_session );
gap> NCurses.BrowseGeneric( t1 );
gap> BrowseData.SetReplay( false );
gap> Unbind( t1.dynamic.replay );
```

(Note that the last statement above is necessary to run the session more than once.) The result is not too bad but we can improve the table, using the optional components of database attributes, as follows.

- The strings "true" and "false" shown for the Boolean valued database attributes can be replaced by the perhaps more suggestive strings "+" and "-" (or perhaps an empty string instead of "-").
- The alignment of values inside their columns can be customized.
- When the browse table is categorized by a column then the values in this column do usually not provide suitable category rows; we can prescribe individual category values.
- The column labels can be customized.
- Where the lexicographic order is not appropriate for sorting table entries, we can prescribe an individual comparison function.
- Sort parameters can be customized.
- We can prescribe the width of a column, and thus distribute the attribute values for this column to several rows when the values are too long.
- Finally, in the call of BrowseTableFromDatabaseIdEnumerator (A.2.2), we can add a header to the browse table.

We create a new database id enumerator and the corresponding browse table, in order to be able to compare the behaviour of the two objects. However, we assume that the variables `n` and `factorialdata` are already available.

Example

```
gap> smallintenum2:= DatabaseIdEnumerator( rec(
>      identifiers:= [ 1 .. n ],
>      entry:= function( dbidenum, id ) return id; end,
>      viewLabel:= "",
>    ) );;
gap> DatabaseAttributeAdd( smallintenum2, rec(
>      identifier:= "primes",
>      type:= "values",
>      name:= "IsPrimeInt",
>      viewLabel:= "prime?",
>      viewValue:= value -> BrowseData.ReplacedEntry( value,
>          [ true, false ], [ "+", "-" ] ),
>      sortParameters:= [ "add counter on categorizing", "yes" ],
>      align:= "c",
>      categoryValue:= value -> BrowseData.ReplacedEntry( value,
>          [ true, false ], [ "prime", "nonprime" ] ),
>    ) );
gap> DatabaseAttributeAdd( smallintenum2, rec(
>      identifier:= "prime powers",
>      type:= "values",
>      name:= "IsPrimePowerInt",
>      viewLabel:= "prime power?",
>      viewValue:= value -> BrowseData.ReplacedEntry( value,
>          [ true, false ], [ "+", "-" ] ),
>      sortParameters:= [ "add counter on categorizing", "yes" ],
>      align:= "c",
>      categoryValue:= value -> BrowseData.ReplacedEntry( value,
>          [ true, false ], [ "prime power", "not prime power" ] ),
>    ) );
gap> DatabaseAttributeAdd( smallintenum2, rec(
>      identifier:= "factors",
>      type:= "values",
>      name:= "Factors",
>      viewLabel:= "factors",
>      viewValue:= value -> JoinStringsWithSeparator( List( value, String ),
>                                                     " * "),
>      widthCol:= 10,
>    ) );
gap> DatabaseAttributeAdd( smallintenum2, rec(
>      identifier:= "residue mod 11",
>      type:= "values",
>      create:= function( attr, id ) return id mod 11; end,
>      viewSort:= BrowseData.SortAsIntegers,
>      categoryValue:= res -> Concatenation( String( res ), " mod 11" ),
>    ) );
gap> DatabaseAttributeAdd( smallintenum2, rec(
>      identifier:= "inverse factorial",
>      type:= "pairs",
>      data:= rec( automatic:= factorialdata( n ), nonautomatic:= [] ),
```

```
>      isSorted:= true,
>      categoryValue:= function( k )
>        if k = "" then
>          return "(no factorial)";
>        else
>          return Concatenation( String( k ), "!" );
>        fi;
>      end,
>    ) );
gap> t2:= BrowseTableFromDatabaseIdEnumerator( smallintenum2,
>           [ "self" ],
>           [ "primes", "prime powers", "factors", "residue mod 11",
>             "inverse factorial" ],
>           t -> BrowseData.HeaderWithRowCounter( t, "Small integers", n ) );;
```

We run the same session as with the browse table for `smallintenum1`.

Example

```
gap> BrowseData.SetReplay( sample_session );
gap> NCurses.BrowseGeneric( t2 );
gap> BrowseData.SetReplay( false );
gap> Unbind( t2.dynamic.replay );
```

Another possibility to change the look of the table is to combine the columns for the two Boolean valued database attributes in one column, by showing the string `"+"` for prime powers, as before, and showing this string in boldface red if the number in question is a prime. We implement this idea in the following database attribute. However, note that this can be a bad idea because text attributes may be not supported in the user's terminal (see Section 2.1.7), or the user may have difficulties to see or to distinguish colors; also, it must be documented which information is encoded in the table, and the column label might be not sufficient for explaining what the text attributes mean. Alternatively, we could show for example combined symbols such as `++`, `+-`, `--` for primes, prime powers, and non-prime-powers, respectively. (We see that besides these issues, the required GAP code is more involved than what is needed for the examples above.)

Example

```
gap> DatabaseAttributeAdd( smallintenum2, rec(
>      identifier:= "primes & prime powers",
>      type:= "values",
>      create:= function( attr, id )
>        if   IsPrimeInt( id ) then
>          return 2;
>        elif IsPrimePowerInt( id ) then
>          return 1;
>        else
>          return 0;
>        fi;
>      end,
>      viewLabel:= [ NCurses.attrs.BOLD + NCurses.ColorAttr( "red", -1 ),
>                    "prime", NCurses.attrs.NORMAL, " power?" ],
>      viewValue:= value -> BrowseData.ReplacedEntry( value,
>          [ 0, 1, 2 ], [ "-", "+",
>                         [ NCurses.attrs.BOLD + NCurses.ColorAttr( "red", -1 ),
>                           true, "+",
```

```
>                          NCurses.ColorAttr( "red", -1 ), false ] ] ),
>      sortParameters:= [ "add counter on categorizing", "yes" ],
>      align:= "c",
>      categoryValue:= value -> BrowseData.ReplacedEntry( value,
>          [ 0, 1, 2 ],
>          [ "not prime power", "prime power, not prime", "prime" ] ),
>    ) );
gap> t3:= BrowseTableFromDatabaseIdEnumerator( smallintenum2,
>           [ "self" ],
>           [ "primes & prime powers", "residue mod 11",
>             "inverse factorial" ],
>           t -> BrowseData.HeaderWithRowCounter( t, "Small integers", n ) );;
gap> sample_session2:= Concatenation(
>         # categorize by the first column, expand categories, wait, reset
>         nop, "scsc", nop, "X", nop, "!", "Q" );;
gap> BrowseData.SetReplay( sample_session2 );
gap> NCurses.BrowseGeneric( t3 );
gap> BrowseData.SetReplay( false );
gap> Unbind( t3.dynamic.replay );
```

Now we want to consider the database as extendible, that is, we want to be able to increase $n$ after constructing the database attributes. For that, we use $n$ as the `version` value of the database id enumerator, and provide `version` and `update` components for all attributes.

Again, we start the construction from scratch.

Example

```
gap> smallintenum3:= DatabaseIdEnumerator( rec(
>     identifiers:= [ 1 .. n ],
>     entry:= function( dbidenum, id ) return id; end,
>     viewLabel:= "",
>     version:= n,
>     update:= function( dbidenum )
>         dbidenum.identifiers:= [ 1 .. n ];
>         dbidenum.version:= n;
>         return true;
>       end,
>     ) );;
gap> updateByUnbindData:= function( attr )
>      Unbind( attr.data );
>      return true;
>    end;;
gap> DatabaseAttributeAdd( smallintenum3, rec(
>     identifier:= "primes",
>     type:= "values",
>     name:= "IsPrimeInt",
>     viewLabel:= "prime?",
>     viewValue:= value -> BrowseData.ReplacedEntry( value,
>         [ true, false ], [ "+", "-" ] ),
>     sortParameters:= [ "add counter on categorizing", "yes" ],
>     align:= "c",
>     categoryValue:= value -> BrowseData.ReplacedEntry( value,
>         [ true, false ], [ "prime", "nonprime" ] ),
>     version:= n,
```

```
>     update:= updateByUnbindData,
>     ) );
gap> DatabaseAttributeAdd( smallintenum3, rec(
>     identifier:= "prime powers",
>     type:= "values",
>     name:= "IsPrimePowerInt",
>     viewLabel:= "prime power?",
>     viewValue:= value -> BrowseData.ReplacedEntry( value,
>         [ true, false ], [ "+", "-" ] ),
>     sortParameters:= [ "add counter on categorizing", "yes" ],
>     align:= "c",
>     categoryValue:= value -> BrowseData.ReplacedEntry( value,
>         [ true, false ], [ "prime power", "not prime power" ] ),
>     version:= n,
>     update:= updateByUnbindData,
>     ) );
gap> DatabaseAttributeAdd( smallintenum3, rec(
>     identifier:= "factors",
>     type:= "values",
>     name:= "Factors",
>     viewLabel:= "factors",
>     viewValue:= value -> JoinStringsWithSeparator( List( value, String ),
>                                                    " * "),
>     widthCol:= 10,
>     version:= n,
>     update:= updateByUnbindData,
>     ) );
gap> DatabaseAttributeAdd( smallintenum3, rec(
>     identifier:= "residue mod 11",
>     type:= "values",
>     create:= function( attr, id ) return id mod 11; end,
>     viewSort:= BrowseData.SortAsIntegers,
>     categoryValue:= res -> Concatenation( String( res ), " mod 11" ),
>     version:= n,
>     update:= updateByUnbindData,
>     ) );
gap> DatabaseAttributeAdd( smallintenum3, rec(
>     identifier:= "inverse factorial",
>     type:= "pairs",
>     data:= rec( automatic:= factorialdata( n ), nonautomatic:= [] ),
>     isSorted:= true,
>     categoryValue:= function( k )
>       if k = "" then
>         return "(no factorial)";
>       else
>         return Concatenation( String( k ), "!" );
>       fi;
>     end,
>     version:= n,
>     update:= function( attr )
>         attr.data.automatic:= factorialdata( n );
>         return true;
```

```
>       end,
>     ) );
```

Now we can change the set of database entries by assigning a new value to the variable `n`, and then calling `DatabaseIdEnumeratorUpdate` (A.1.7).

Example

```
gap> n:= 200;;
gap> DatabaseIdEnumeratorUpdate( smallintenum3 );
true
gap> t4:= BrowseTableFromDatabaseIdEnumerator( smallintenum3,
>   [ "self" ], [ "primes", "prime powers", "factors", "residue mod 11",
>     "inverse factorial" ],
>   t -> BrowseData.HeaderWithRowCounter( t, "Small integers", n ) );;
gap> BrowseData.SetReplay( sample_session );
gap> NCurses.BrowseGeneric( t4 );
gap> BrowseData.SetReplay( false );
gap> Unbind( t4.dynamic.replay );
```

## A.4 Example: An Overview of the GAP Library of Tables of Marks

The example shown in this section deals with GAP's Library of Tables of Marks (the TomLib package [NMP13]).

### A.4.1 BrowseTomLibInfo

▷ `BrowseTomLibInfo()` (function)

**Returns:** nothing.

This function shows the contents of the GAP Library of Tables of Marks (the TomLib package, see [NMP13]) in a browse table.

The first call may take substantial time (about 40 seconds), because the data files of the TomLib package are evaluated. This could be improved by precomputing and caching the values. Another possibility would be to call `BrowseTomLibInfo` once before creating a GAP workspace. The subsequent calls are not expensive.

The table rows correspond to the tables of marks, one column of row labels shows the identifier of the table. The columns of the table contain information about the group order, the number of conjugacy classes of subgroups, the identifiers of tables of marks with fusions to and from the given table, and the name of the file that contains the table of marks data.

The full functionality of the function `NCurses.BrowseGeneric` (4.3.1) is available.

Example

```
gap> c:= [ NCurses.keys.ENTER ];;
gap> n:= [ 14, 14, 14 ];;  # ‘‘do nothing’’
gap> BrowseData.SetReplay( Concatenation(
>       "scrrsc",   # categorize the list by source tables of fusions,
>       "srdd",     # choose a source table,
>       "x",        # expand the list of targets of fusions
>       n,
>       "!",        # revert the categorization
>       "q",        # leave the mode in which a row is selected
>       "scrrrrsc", # categorize the list by filenames
```

```
>        "X",        # expand all categories
>        n,
>        "!",        # revert the categorization
>        "scso",     # sort the list by group order
>        n,
>        "!q",       # revert the sorting and selection
>        "?",        # open the help window
>        n,
>        "Q",        # close the help window
>        "/A5", c,   # search for the first occurrence of "A5"
>        n,
>        "Q" ) );;   # and quit the browse table
gap> BrowseTomLibInfo();
gap> BrowseData.SetReplay( false );
```

# References


[Bog] A. Bogomolny. Sam Loyd's Fifteen. http://www.cut-the-knot.org/pythagoras/fifteen.shtml. 65

[CCN+85] J. H. Conway, R. T. Curtis, S. P. Norton, R. A. Parker, and R. A. Wilson. *Atlas of finite groups*. Oxford University Press, Eynsham, 1985. Maximal subgroups and ordinary characters for simple groups, With computational assistance from J. G. Thackray. 63

[CN04] F. Celler and M. Neunhöffer. XGAP, a graphical user interface for GAP, Version 4.21. http://www-groups.mcs.st-and.ac.uk/~neunhoef/Computer/Software/Gap/xgap4.html, May 2004. Refereed GAP package. 66

[GAP] A Bibliography of GAP related publications. http://www.gap-system.org/Doc/Bib/gap-publishednicer.bib. 55

[JLPW95] C. Jansen, K. Lux, R. Parker, and R. Wilson. *An atlas of Brauer characters*, volume 11 of *London Mathematical Society Monographs. New Series*. The Clarendon Press Oxford University Press, New York, 1995. Appendix 2 by T. Breuer and S. Norton, Oxford Science Publications. 63

[Köla] J. Köller. Peg Solitaire. http://www.mathematische-basteleien.de/solitaire.htm. 67

[Kölb] J. Köller. Rubik's Cube. http://www.mathematische-basteleien.de/rubikscube.htm. 68

[LN07] F. Lübeck and M. Neunhöffer. GAPDoc, a Meta Package for GAP Documentation, Version 1.0. http://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc, May 2007. Refereed GAP package. 57

[NCu] The ncurses C library. https://invisible-island.net/ncurses/ncurses.faq.html. 5, 8

[Neu07] M. Neunhöffer. IO, bindings for low level C library IO, Version 2.2. http://www-groups.mcs.st-and.ac.uk/~neunhoef/Computer/Software/Gap/io.html, Apr 2007. GAP package. 65

[NMP13] L. Naughton, T. Merkwitz, and G. Pfeiffer. TomLib, the GAP library of tables of marks, Version 1.2.4. http://schmidt.nuigalway.ie/tomlib/tomlib, Nov 2013. GAP package. 63, 94

[OR] J. J. O'Connor and E. F. Robertson. http://www-history.mcs.st-and.ac.uk/HistTopics/Mathematical_games.html. 65

[OR00] J. J. O'Connor and E. F. Robertson. Mathematical MacTutor. http://www-groups.dcs.st-and.ac.uk/~edmund/mactutor.html, 2000. 69

[Sch] M. Schönert. Analyzing Rubik's Cube with GAP. http://www.gap-system.org/Doc/Examples/rubik.html. 68

[WPN+19] R. A. Wilson, R. A. Parker, S. Nickerson, J. N. Bray, and T. Breuer. AtlasRep, a GAP Interface to the Atlas of Group Representations, Version 2.1. http://www.math.rwth-aachen.de/~Thomas.Breuer/atlasrep, May 2019. GAP package. 52, 63

# Index