Adapting Proof General

Proof General — Organize your proofs!

Adapting Proof General 3.7 to new provers
January 2008
proofgeneral.inf.ed.ac.uk

David Aspinall with T. Kleymann

This manual documents Proof General, Version 3.7, for use with XEmacs 21.5.28 and GNU Emacs 22.1.1 or later versions. Proof General is distributed under the terms of the GNU General Public License (GPL); please check the accompanying file 'COPYING' for more details.

Visit Proof General on the web at `http://proofgeneral.inf.ed.ac.uk`

Version control: `PG-adapting.texi,v 9.1 2008/07/12 14:00:46 da Exp`

Introduction

Welcome to Proof General!

Proof General a generic Emacs-based interface for proof assistants.

This manual contains information for adapting Proof General to new proof assistants, and some sketches of the internal implementation. It is not intended for most ordinary users of the system. For full details about how to use Proof General, and information on its availability and installation, please see the main Proof General manual which should accompany this one.

We positively encourage the support of new systems. Proof General has grown more flexible and useful as it has been adapted to more proof assistants.

Typically, adding support for a new prover improves support for others, both because the code becomes more robust, and because new ideas are brought into the generic setting. Notice that the Proof General framework has been built as a "product line architecture": generality has been introduced step-by-step in a demand-driven way, rather than at the outset as a grand design. Despite this strategy, the interface has a surprisingly clean structure. The approach means that we fully expect hiccups when adding support for new assistants, so the generic core may need extension or modification. To support this we have an open development method: if you require changes in the generic support, please contact us (or make adjustments yourself and send them to us).

Proof General has a home page at `http://proofgeneral.inf.ed.ac.uk`. Visit this page for the latest version of the manuals, other documentation, system downloads, etc.

Future

The aim of the Proof General project is to provide a powerful and configurable interfaces which help user-interaction with interactive proof assistants.

The strategy Proof General uses is to targets power users rather than novices; other interfaces have often neglected this class of users. But we do include general user interface niceties, such as toolbar and menus, which make use easier for all.

Proof General has been Emacs based so far, but plans are afoot to liberate it from the points and parentheses of Emacs Lisp. The successor project Proof General Kit proposes that proof assistants use a *standard* XML-based protocol for interactive proof, dubbed **PGIP**.

PGIP enables middleware for interactive proof tools and interface components. Rather than configuring Proof General for your proof assistant, you will need to configure your proof assistant to understand PGIP. There is a similarity however; the design of PGIP was based heavily on the Emacs Proof General framework. This means that effort on customizing Emacs Proof General to a new proof assistant is worthwhile even in the light of PGIP: it will help you to understand Proof General's model of interaction, and moreover, we hope to use the Emacs customizations to provide experimental filters which allow supported provers to communicate using PGIP.

At the time of writing, these ideas are in early stages. For latest details, or to become involved, see the Proof General Kit webpage.

Credits

David Aspinall put together and wrote most of this manual. Thomas Kleymann wrote some of the text in Chapter 8. Much of the content is generated automatically from Emacs docstrings, some of which have been written by other Proof General developers.

1 Beginning with a New Prover

Proof General has about 100 configuration variables which are set on a per-prover basis to configure the various features. It may sound like a lot but don't worry! Many of the variables occur in pairs (typically regular expressions matching the start and end of some text), and you can begin by setting just a fraction of the variables to get the basic features of script management working. The bare minimum for a working prototype is about 25 simple settings.

For more advanced features you may need (or want) to write some Emacs Lisp. If you're adding new functionality please consider making it generic for different proof assistants, if appropriate. When writing your modes, please follow the Emacs Lisp conventions See Info file '`lispref`', node '`Style Tips`'.

The configuration variables are declared in the file '`generic/proof-config.el`'. The details in the central part of this manual are based on the contents of that file, beginning in Chapter 2 [Menus and Toolbar and User-level Commands], page 7, and continuing until Chapter 7 [Global Constants], page 31. Other chapters cover the details of configuring for multiple files and for supporting the other Emacs packages mentioned in the user manual (*Support for other Packages*). If you write additional Elisp code interfacing to Proof General, you can find out about some useful functions by reading Chapter 12 [Writing More Lisp Code], page 41. The last chapter of this manual describes some of the internals of Proof General, in case you are interested, maybe because you need to extend the generic core to do something new.

In the rest of this chapter we describe the general mechanisms for instantiating Proof General. We assume some knowledge of the content of the main Proof General manual.

1.1 Overview of adding a new prover

Each proof assistant supported has its own subdirectory under `proof-home-directory`, used to store a root elisp file and any other files needed to adapt the proof assistant for Proof General.

Here is how to go about adding support for a new prover.

1. Make a directory called '`myassistant/`' under the Proof General home directory `proof-home-directory`, to put the specific customization and associated files in.
2. Add a file '`myassistant.el`' to the new directory.
3. Edit '`proof-site.el`' to add a new entry to the `proof-assistants-table` variable. The new entry should look like this:

   ```
   (myassistant "My Proof Assistant" "\\.myasst$")
   ```

   The first item is used to form the name of the internal variables for the new mode as well as the directory and file where it loads from. The second is a string, naming the proof assistant. The third item is a regular expression to match names of proof script files for this assistant. See the documentation of `proof-assistant-table` for more details.
4. Define the new Proof General modes in '`myassistant.el`', by setting configuration variables to customize the behaviour of the generic modes.

`proof-assistant-table` [User Option]

Proof General's table of supported proof assistants.

This is copied from '`proof-assistant-table-default`' at load time, removing any entries that do not have a corresponding directory under '`proof-home-directory`'.

Each entry is a list of the form

```
(symbol name automode-regexp)
```

The *name* is a string, naming the proof assistant. The *symbol* is used to form the name of the mode for the assistant, '`SYMBOL-mode`', run when files with *automode-regexp* are visited. *symbol* is also used to form the name of the directory and elisp file for the mode, which will be

proof-home-directory`/`*symbol*`/`*symbol*`.el`

where *proof-home-directory* is the value of the variable '`proof-home-directory`'.

The default value is `((isar "Isabelle" "\\.thy$") (coq "Coq" "\\.v$\\|\\.v8$\\|\\.v7$") (phox "PhoX" "\\.phx$") (lego "LEGO" "\\.l$") (plastic "Plastic" "\\.lf$"))`.

The final step of the description above is where the work lies. There are two basic methods. You can write some Emacs lisp functions and define the modes using the macro `define-derived-mode`. Or you can use the new easy configuration mechanism of Proof General 3.0 described in the next section, which calls `define-derived-mode` for you. You still need to know which configuration variables should be set, and how to set them.

The documentation below (and inside Emacs) should help with that, but the best way to begin might be to use an existing Proof General instance as an example.

1.2 Demonstration instance and easy configuration

Proof General is supplied with a demonstration instance for Isabelle which configures the basic features. This is a whittled down version of Isabelle Proof General, which you can use as a template to get support for a new assistant going. Check the directory '`demoisa`' for the two files '`demoisa.el`' and '`demoisa-easy.el`'.

The file '`demoisa.el`' follows the scheme described in Section 1.3 [Major modes used by Proof General], page 5. It uses the Emacs Lisp macro `define-derived-mode` to define the four modes for a Proof General instance, by inheriting from the generic code. Settings which configure Proof General are made by functions called from within each mode, as appropriate.

The file '`demoisa-easy.el`' uses a new simplified mechanism to achieve (virtually) the same result. It uses the macro `proof-easy-config` defined in '`proof-easy-configl.el`' to make all of the settings for the Proof General instance in one go, defining the derived modes automatically using a regular naming scheme. No lisp code is used in this file except the call to this macro. The minor difference in the end result is that all the variables are set at once, rather than inside each mode. But since the configuration variables are all global variables anyway, this makes no real difference.

The macro `proof-easy-config` is called like this:

```
(proof-easy-config myprover "MyProver"
     config_1 val_1
     ...
     config_n val_n)
```

The main body of the macro call is like the body of a `setq`. It contains pairs of variables and value settings. The first argument to the macro is a symbol defining the mode root, the second argument is a string defining the mode name. These should be the same as the first part of the entry in `proof-assistant-table` for your prover. See Section 1.1 [Overview of adding a new prover], page 3. After the call to `proof-easy-config`, the new modes *myprover*`-mode`, *myprover*`-shell-mode`, *myprover*`-response-mode`, and *myprover*`-goals-mode` will be defined. The configuration variables in the body will be set immediately.

This mechanism is in fact recommended for new instantiations of Proof General since it follows a regular pattern, and we can more easily adapt it in the future to new versions of Proof General.

Even Emacs Lisp experts should prefer the simplified mechanism. If you want to set some buffer-local variables in your Proof General modes, or invoke supporting lisp code, this can easily be done by adding functions to the appropriate mode hooks after the `proof-easy-config` call. For example, to add extra settings for the shell mode for `demoisa`, we could do this:

```
(defun demoisa-shell-extra-config ()
   extra configuration ...
 )
(add-hook 'demoisa-shell-mode-hook 'demoisa-shell-extra-config)
```

The function to do extra configuration `demoisa-shell-extra-config` is then called as the final step when `demoisa-shell-mode` is entered (be wary, this will be after the generic `proof-shell-config-done` is called, so it will be too late to set normal configuration variables which may be examined by `proof-shell-config-done`).

1.3 Major modes used by Proof General

There are four major modes used by Proof General, one for each type of buffer it handles. The buffer types are: script, shell, response and goals. Each of these has a generic mode, respectively: `proof-mode`, `proof-shell-mode`, `proof-response-mode`, and `proof-goals-mode`.

The pattern for defining the major mode for an instance of Proof General is to use `define-derived-mode` to define a specific mode to inherit from each generic one, like this:

```
(define-derived-mode myass-shell-mode proof-shell-mode
   "MyAss shell" nil
   (myass-shell-config)
   (proof-shell-config-done))
```

Where `myass-shell-config` is a function which sets the configuration variables for the shell (see Chapter 4 [Proof Shell Settings], page 17).

It's important that each of your modes invokes one of the functions `proof-config-done`, `proof-shell-config-done`, `proof-response-config-done`, or `proof-goals-config-done` once it has set its configuration variables. These functions finalize the configuration of the mode.

The modes must be named standardly, replacing `proof-` with the prover's symbol name, *`PA-`*. See the file '`demoisa.el`' for an example of the four calls to `define-derived-mode`.

2 Menus, toolbar, and user-level commands

The variables described in this chapter configure the menus, toolbar, and user-level commands. They should be set in the script mode before `proof-config-done` is called. (Toolbar configuration must be made before '`proof-toolbar.el`' is loaded, which usually is triggered automatically by an attempt to display the toolbar).

2.1 Settings for generic user-level commands

`proof-assistant-home-page` [Variable]
Web address for information on proof assistant.
Used for Proof General's help menu.

`proof-context-command` [Variable]
Command to display the context in proof assistant.

`proof-info-command` [Variable]
Command to ask for help or information in the proof assistant.
String or fn. If a string, the command to use. If a function, it should return the command string to insert.

`proof-showproof-command` [Variable]
Command to display proof state in proof assistant.

`proof-goal-command` [Variable]
Command to set a goal in the proof assistant. String or fn.
If a string, the format character '`%s`' will be replaced by the goal string. If a function, it should return the command string to insert.

`proof-save-command` [Variable]
Command to save a proved theorem in the proof assistant. String or fn.
If a string, the format character '`%s`' will be replaced by the theorem name. If a function, it should return the command string to insert.

`proof-find-theorems-command` [Variable]
Command to search for a theorem containing a given term. String or fn.
If a string, the format character '`%s`' will be replaced by the term. If a function, it should return the command string to insert.

2.2 Menu configuration

As well as the generic Proof General menu, each proof assistant is provided with a specific menu which can have prover-specific commands. Proof General puts some default things on this menu, including the commands to start/stop the prover, and the user-extensible "Favourites" menu.

`PA-menu-entries` [Variable]
Extra entries for proof assistant specific menu.
A list of menu items [*name callback enabler* ...]. See the documentation of '`easy-menu-define`' for more details.

`PA-help-menu-entries` [Variable]
Extra entries for help submenu for proof assistant specific help menu.
A list of menu items [*name callback enabler* ...]. See the documentation of '`easy-menu-define`' for more details.

2.3 Toolbar configuration

Unlike the menus, Proof General has only one toolbar. For the "generic" aspect of Proof General to work well, we shouldn't change (the meaning of) the existing toolbar buttons too far. This would discourage people from experimenting with different proof assistants when they don't really know them, which is one of the advantages that Proof General brings.

But in case it is hard to map some of the generic buttons onto functions in particular provers, and to allow extra buttons, there is a mechanism for adjustment.

I used The Gimp to create the buttons for Proof General. The development distribution includes a button blank and some notes in '`etc/notes.txt`' about making new buttons.

`proof-toolbar-entries-default` [Variable]

Example value for proof-toolbar-entries. Also used to define scripting menu.
This gives a bare toolbar that works for any prover, providing the appropriate configuration variables are set. To add/remove prover specific buttons, adjust the '`<PA>-toolbar-entries`' variable, and follow the pattern in '`proof-toolbar.el`' for defining functions, images.

`PA-toolbar-entries` [Variable]

List of entries for Proof General toolbar and Scripting menu.
Format of each entry is (*token menuname tooltip dynamic-enabler-p enable*).

For each *token*, we expect an icon with base filename *token*, a function proof-toolbar-<TOKEN>, and (optionally) a dynamic enabler proof-toolbar-<TOKEN>-enable-p.

If *enablep* is absent, item is enabled; if *enablep* is present, item is only added to menubar and toolbar if *enablep* is non-null.

If *menuname* is nil, item will not appear on the scripting menu.

If *tooltip* is nil, item will not appear on the toolbar.

The default value is '`proof-toolbar-entries-default`' which contains the standard Proof General buttons.

Here's an example of how to remove a button, from '`af2.el`':

```
(setq af2-toolbar-entries
      (remassoc 'state af2-toolbar-entries))
```

3 Proof Script Settings

The variables described in this chapter should be set in the script mode before `proof-config-done` is called. These variables configure recognition of commands in the proof script, and also control some of the behaviour of script management.

3.1 Recognizing commands and comments

The first four settings configure the generic parsing strategy for commands in the proof script. Usually only one of these three needs to be set. If the generic parsing functions are not flexible for your needs, you can supply a value for `proof-script-parse-function`.

Note that for the generic functions to work properly, it is **essential** that you set the syntax table for your mode properly, so that comments and strings are recognized. See the Emacs documentation to discover how to do this (particularly for the function `modify-syntax-entry`).

See Section 13.5 [Proof script mode], page 48, for more details of the parsing functions.

`proof-terminal-char` [Variable]
Character that terminates commands sent to prover; nil if none.

To configure command recognition properly, you must set at least one of these: '`proof-script-sexp-commands`', '`proof-script-command-end-regexp`', '`proof-script-command-start-regexp`', '`proof-terminal-char`', or '`proof-script-parse-function`'.

`proof-script-sexp-commands` [Variable]
Non-nil if script has LISP-like syntax: commands are `top-level` sexps.
You should set this variable in script mode configuration.

To configure command recognition properly, you must set at least one of these: '`proof-script-sexp-commands`', '`proof-script-command-end-regexp`', '`proof-script-command-start-regexp`', '`proof-terminal-char`', or '`proof-script-parse-function`'.

`proof-script-command-start-regexp` [Variable]
Regular expression which matches start of commands in proof script.
You should set this variable in script mode configuration.

To configure command recognition properly, you must set at least one of these: '`proof-script-sexp-commands`', '`proof-script-command-end-regexp`', '`proof-script-command-start-regexp`', '`proof-terminal-char`', or '`proof-script-parse-function`'.

`proof-script-command-end-regexp` [Variable]
Regular expression which matches end of commands in proof script.
You should set this variable in script mode configuration.

The end of the command is considered to be the end of the match of this regexp. The regexp may include a nested group, which can be used to recognize the start of the following command (or white space). If there is a nested group, the end of the command is considered to be the start of the nested group, i.e. (`match-beginning` 1), rather than (`match-end` 0).

To configure command recognition properly, you must set at least one of these: '`proof-script-sexp-commands`', '`proof-script-command-end-regexp`', '`proof-script-command-start-regexp`', '`proof-terminal-char`', or '`proof-script-parse-function`'.

The next four settings configure the comment syntax. Notice that to get reliable behaviour of the parsing functions, you may need to modify the syntax table for your prover's mode. Read the Elisp manual for details about that.

`proof-script-comment-start` [Variable]
String which starts a comment in the proof assistant command language.
The script buffer's '`comment-start`' is set to this string plus a space. Moreover, comments are usually ignored during script management, and not sent to the proof process.

You should set this variable for reliable working of Proof General, as well as '`proof-script-comment-end`'.

`proof-script-comment-start-regexp` [Variable]
Regexp which matches a comment start in the proof command language.

The default value for this is set as (`regexp-quote proof-script-comment-start`) but you can set this variable to something else more precise if necessary.

`proof-script-comment-end` [Variable]
String which ends a comment in the proof assistant command language.
Should be an empty string if comments are terminated by '`end-of-line`' The script buffer's '`comment-end`' is set to a space plus this string, if it is non-empty.

See also '`proof-script-comment-start`'.

You should set this variable for reliable working of Proof General.

`proof-script-comment-end-regexp` [Variable]
Regexp which matches a comment end in the proof command language.

The default value for this is set as (`regexp-quote proof-script-comment-end`) but you can set this variable to something else more precise if necessary.

`proof-case-fold-search` [Variable]
Value for '`case-fold-search`' when recognizing portions of proof scripts.
Also used for completion, via '`proof-script-complete`'. The default value is nil. If your prover has a case **insensitive** input syntax, `proof-case-fold-search` should be set to t instead. NB: This setting is not used for matching output from the prover.

3.2 Recognizing proofs

Up to three settings each may be supplied for recognizing goal-like and save-like commands. The `-with-hole-` settings are used to make a record of the name of the theorem proved, and also to build a default value for the rather complicated setting `proof-script-next-entity-regexps`, which activates the *function menu* feature.

The `-p` subsidiary predicates were added to allow more discriminating behaviour for particular proof assistants. (This is a typical example of where the core framework needs some additional generalization, to simplify matters, and allow for a smooth handling of nested proofs; the present state is only part of the way there).

`proof-goal-command-regexp` [Variable]
Matches a goal command in the proof script.
This is used (1) to make the default value for '`proof-goal-command-p`', used as an important part of script management to find the start of an atomic undo block, and (2) to construct the default for '`proof-script-next-entity-regexps`' used for function menus.

`proof-goal-command-p` [Variable]
A function to test: is this really a goal command span?

This is added as a more refined addition to `proof-goal-command-regexp`, to solve the problem that Coq and some other provers can have goals which look like definitions, etc. (In the future we may generalize `proof-goal-command-regexp` instead).

`proof-goal-with-hole-regexp` [Variable]
Regexp which matches a command used to issue and name a goal.
The name of the theorem is built from the variable `proof-goal-with-hole-result` using the same convention as for '`query-replace-regexp`'. Used for setting names of goal..save regions and for default configuration of other modes (function menu, imenu).

It's safe to leave this setting as nil.

`proof-goal-with-hole-result` [Variable]
How to build theorem name after matching with '`proof-goal-with-hole-regexp`'.
String or Int. If an int N use `match-string` to recover the value of the Nth parenthesis matched. If it is a string use `replace-match`. In this case, `proof-save-with-hole-regexp` should match the entire command

`proof-save-command-regexp` [Variable]
Matches a save command.

`proof-save-with-hole-regexp` [Variable]
Regexp which matches a command to save a named theorem.
The name of the theorem is build from the variable '`proof-save-with-hole-result`' using the same convention as '`query-replace-regexp`'. Used for setting names of goal..save and proof regions and for default function-menu configuration in '`proof-script-find-next-entity`'.

It's safe to leave this setting as nil.

`proof-completed-proof-behaviour` [Variable]
Indicates how Proof General treats commands beyond the end of a proof.
Normally goal...save regions are "closed", i.e. made atomic for undo. But once a proof has been completed, there may be a delay before the "save" command appears — or it may not appear at all. Unless nested proofs are supported, this can spoil the undo-behaviour in script management since once a new goal arrives the old undo history may be lost in the prover. So we allow Proof General to close off the goal..[save] region in more flexible ways. The possibilities are:

```
           nil  -  nothing special; close only when a save arrives
     'closeany  -  close as soon as the next command arrives, save or not
    'closegoal  -  close when the next "goal" command arrives
      'extend   -  keep extending the closed region until a save or goal.
```

If your proof assistant allows nested goals, it will be wrong to close off the portion of proof so far, so this variable should be set to nil.

NB: `'extend` behaviour is not currently compatible with appearance of save commands, so don't use that if your prover has save commands.

`proof-really-save-command-p` [Variable]
Is this really a save command?

This is a more refined addition to `proof-save-command-regexp`. It should be a function taking a span and command as argument, and can be used to track nested proofs. (See what is done in isar/ for example). In the future, this setting should be removed when the generic core is extended to handle nested proofs smoothly.

3.3 Recognizing other elements

To configure *Imenu* (which in turn configures *Speedbar*), you may use the following setting. If this is unset, a generic setting based on `proof-goal-with-hole-regexp` is configured.

`proof-script-imenu-generic-expression` [Variable]
Regular expressions to help find definitions and proofs in a script.
Value for '`imenu-generic-expression`', see documentation of Imenu and that variable for details.

`imenu-generic-expression` [Variable]
The regex pattern to use for creating a buffer index.

If non-nil this pattern is passed to '`imenu--generic-function`' to create a buffer index.

The value should be an alist with elements that look like this:

```
(menu-title regexp index)
```

or like this:

```
(menu-title regexp index function ARGUMENTS...)
```

with zero or more ARGUMENTS. The former format creates a simple element in the index alist when it matches; the latter creates a special element of the form (*name function position-marker* ARGUMENTS...) with *function* and *arguments* beiong copied from '`imenu-generic-expression`'.

menu-title is a string used as the title for the submenu or nil if the entries are not nested.

regexp is a regexp that should match a construct in the buffer that is to be displayed in the menu; i.e., function or variable definitions, etc. It contains a substring which is the name to appear in the menu. See the info section on Regexps for more information.

index points to the substring in *regexp* that contains the name (of the function, variable or type) that is to appear in the menu.

The variable is buffer-local.

The variable '`imenu-case-fold-search`' determines whether or not the regexp matches are case sensitive. and '`imenu-syntax-alist`' can be used to alter the syntax table for the search.

For example, see the value of '`lisp-imenu-generic-expression`' used by '`lisp-mode`' and '`emacs-lisp-mode`' with '`imenu-syntax-alist`' set locally to give the characters which normally have \"punctuation\" syntax \"word\" syntax during matching."

To configure the *function menu* feature, there are a couple of settings. These can be used to recognize named proofs, and other elements in the proof script as well. (**NOTE**: it is likely that function menu support will be removed in favour of Imenu only in the next release of Proof General).

`proof-script-next-entity-regexps` [Variable]
Regular expressions to help find definitions and proofs in a script.
This is the list of the form

```
(anyentity-regexp
 discriminator-regexp  ... discriminator-regexp)
```

The idea is that *anyentity-regexp* matches any named entity in the proof script, on a line where the name appears. This is assumed to be the start or the end of the entity. The discriminators then test which kind of entity has been found, to get its name. A *discriminator-regexp* has one of the forms

```
(regexp matchnos)
(regexp matchnos 'backward backregexp)
(regexp matchnos 'forward forwardregexp)
```

If *regexp* matches the string captured by *anyentity-regexp*, then *matchnos* are the match numbers for the substrings which name the entity (these may be either a single number or a list of numbers).

If `'backward` *backregexp* is present, then the start of the entity is found by searching backwards for *backregexp*.

Conversely, if `'forward` *forwardregexp* is found, then the end of the entity is found by searching forwards for *forwardregexp*.

Otherwise, the start and end of the entity will be the region matched by *anyentity-regexp*.

This mechanism allows fairly complex parsing of the buffer, in particular, it allows for goal..save regions which are named only at the end. However, it does not parse strings, comments, or parentheses.

This variable may not need to be set: a default value which should work for goal..saves is calculated from `proof-goal-with-hole-regexp`, `proof-goal-command-regexp`, and `proof-save-with-hole-regexp`.

`proof-script-find-next-entity-fn` [Variable]

Name of function to find next interesting entity in a script buffer.

This is used to configure `func-menu`. The default value is '`proof-script-find-next-entity`', which searches for the next entity based on fume-function-name-regexp which by default is set from '`proof-script-next-entity-regexps`'.

The function should move point forward in a buffer, and return a cons cell of the name and the beginning of the entity's region.

Note that '`proof-script-next-entity-regexps`' is set to a default value from '`proof-goal-with-hole-regexp`' and '`proof-save-with-hole-regexp`' in the function `proof-config-done`, so you may not need to worry about any of this. See whether function menu does something sensible by default.

3.4 Configuring undo behaviour

The settings here are used to configure the way "undo" commands are calculated.

`proof-non-undoables-regexp` [Variable]

Regular expression matching commands which are **not** undoable.

These are commands which should not appear in proof scripts, for example, undo commands themselves (if the proof assistant cannot "redo" an "undo"). Used in default functions '`proof-generic-state-preserving-p`' and '`proof-generic-count-undos`'. If you don't use those, may be left as nil.

`proof-undo-n-times-cmd` [Variable]

Command to undo n steps of the currently open goal.

String or function. If this is set to a string, '`%s`' will be replaced by the number of undo steps to issue. If this is set to a function, it should return the appropriate command when called with an integer (the number of undo steps).

This setting is used for the default '`proof-generic-count-undos`'. If you set '`proof-count-undos-fn`' to some other function, there is no need to set this variable.

`proof-ignore-for-undo-count` [Variable]

Matcher for script commands to be ignored in undo count.

May be left as nil, in which case it will be set to '`proof-non-undoables-regexp`'. Used in default function '`proof-generic-count-undos`'.

`proof-count-undos-fn` [Variable]
Function to calculate a command to issue undos to reach a target span.
The function takes a span as an argument, and should return a string which is the command to undo to the target span. The target is guaranteed to be within the current (open) proof. This is an important function for script management. The default setting '`proof-generic-count-undos`' is based on the settings '`proof-non-undoables-regexp`' and '`proof-non-undoables-regexp`'.

`proof-generic-count-undos` *span* [Function]
Count number of undos in a span, return command needed to undo that far.
Command is set using '`proof-undo-n-times-cmd`'.

A default value for '`proof-count-undos-fn`'.

For this function to work properly, you must configure '`proof-undo-n-times-cmd`' and '`proof-ignore-for-undo-count`'.

`proof-find-and-forget-fn` [Variable]
Function that returns a command to forget back to before its argument span.
This setting is used to for retraction (undoing) in proof scripts.

It should undo the effect of all settings between its target span up to (`proof-locked-end`). This may involve forgetting a number of definitions, declarations, or whatever.

The special string `proof-no-command` means there is nothing to do.

This is an important function for script management. Study one of the existing instantiations for examples of how to write it, or leave it set to the default function '`proof-generic-find-and-forget`' (which see).

`proof-generic-find-and-forget` *span* [Function]
Calculate a forget/undo command to forget back to *span*.
This is a long-range forget: we know that there is no open goal at the moment, so forgetting involves unbinding declarations, etc, rather than undoing proof steps.

This generic implementation assumes it is enough to find the nearest following span with a '`name`' property, and retract that using '`proof-forget-id-command`' with the given name.

If this behaviour is not correct, you must customize the function with something different.

`proof-forget-id-command` [Variable]
Command to forget back to a given named span.
A string; '`%s`' will be replaced by the name of the span.

This is only used in the implementation of '`proof-generic-find-and-forget`', you only need to set if you use that function (by not customizing '`proof-find-and-forget-fn`'.

`pg-topterm-goalhyplit-fn` [Variable]
Function to return cons if point is at a goal/hypothesis/literal.
This is used to parse the proofstate output to mark it up for proof-by-pointing or literal command insertion. It should return a cons or nil. First element of the cons is a symbol, `'goal'`, `'hyp'` or `'lit'`. The second element is a string: the goal, hypothesis, or literal command itself.

If you leave this variable unset, no proof-by-pointing markup will be attempted.

`proof-kill-goal-command` [Variable]
Command to kill the currently open goal.

If this is set to nil, PG will expect `proof-find-and-forget-fn` to do all the work of retracting to an arbitrary point in a file. Otherwise, the generic split-phase mechanism will be used:

1. If inside an unclosed proof, use '`proof-count-undos`'. 2. If retracting to before an unclosed proof, use '`proof-kill-goal-command`', followed by '`proof-find-and-forget-fn`' if necessary.

3.5 Nested proofs

Proof General allows configuration for provers which have particular notions of nested proofs. The right thing may happen automatically, or you may need to adjust some of the following settings.

First, you should alter the next setting if the prover retains history for nested proofs.

`proof-nested-goals-history-p` [Variable]

Whether the prover supports recovery of history for nested proofs.
If it does (non-nil), Proof General will retain history inside nested proofs. If it does not, Proof General will amalgamate nested proofs into single steps within the outer proof.

Second, it may happen (i.e. it does for Coq) that the prover has a history mechanism which necessitates keeping track of the number of nested "undoable" commands, even if the history of the proof itself is lost.

`proof-nested-undo-regexp` [Variable]

Regexp for commands that must be counted in nested goal-save regions.

Used for provers which allow nested atomic goal-saves, but with some nested history that must be undone specially.

At the moment, the behaviour is that a goal-save span has a `'nestedundos` property which is set to the number of commands within it which match this regexp. The idea is that the prover-specific code can create a customized undo command to retract the goal-save region, based on the `'nestedundos` setting. Coq uses this to forget declarations, since declarations in Coq reside in a separate context with its own (flat) history.

3.6 Safe (state-preserving) commands

A proof command is "safe" if it can be issued away from the proof script. For this to work it should be state-preserving in the proof assistant (with respect to an on-going proof).

`proof-state-preserving-p` [Variable]

A predicate, non-nil if its argument (a command) preserves the proof state.
This is a safety-test used by `proof-minibuffer-cmd` to filter out scripting commands which should be entered directly into the script itself.

The default setting for this function, '`proof-generic-state-preserving-p`' tests by negating the match on '`proof-non-undoables-regexp`'.

`proof-generic-state-preserving-p` *cmd* [Function]

Is *cmd* state preserving? Match on '`proof-non-undoables-regexp`'.

3.7 Activate scripting hook

`proof-activate-scripting-hook` [Variable]

Hook run when a buffer is switched into scripting mode.
The current buffer will be the newly active scripting buffer.

This hook may be useful for synchronizing with the proof assistant, for example, to switch to a new theory (in case that isn't already done by commands in the proof script).

When functions in this hook are called, the variable '`activated-interactively`' will be non-nil if `proof-activate-scripting` was called interactively (rather than as a side-effect of some other action). If a hook function sends commands to the proof process, it should wait for them to complete (so the queue is cleared for scripting commands), unless activated-interactively is set.

3.8 Automatic multiple files

See Chapter 8 [Handling Multiple Files], page 33, for more details about this setting.

`proof-auto-multiple-files` [Variable]

Whether to use automatic multiple file management.
If non-nil, Proof General will automatically retract a script file whenever another one is retracted which it depends on. It assumes a simple linear dependency between files in the order which they were processed.

If your proof assistant has no management of file dependencies, or one which depends on a simple linear context, you may be able to use this setting to good effect. If the proof assistant has more complex file dependencies then you should configure it to communicate with Proof General about the dependencies rather than using this setting.

3.9 Completions

Proof General allows provers to create a *completion table* to help writing keywords and identifiers in proof scripts. This is documented in the main *Proof General* user manual but summarized here for (a different kind of) completion.

Completions are filled in according to what has been recently typed, from a database of symbols. The database is automatically saved at the end of a session. Completion is usually a hand-wavy thing, so we don't make any attempt to maintain a precise completion table or anything.

The completion table maintained by '`complete.el`' is initialized from `PA-completion-table` when '`proof-script.el`' is loaded. This is done with the function `proof-add-completions` which you may want to call at other times.

`PA-completion-table` [Variable]

List of identifiers to use for completion for this proof assistant.
Completion is activated with *c-ret*.

If this table is empty or needs adjusting, please make changes using '`customize-variable`' and post suggestions at http://proofgeneral.inf.ed.ac.uk/trac

`proof-add-completions` [Command]

Add completions from <PA>-completion-table to completion database.
Uses '`add-completion`' with a negative number of uses and ancient last use time, to discourage saving these into the users database.

4 Proof Shell Settings

The variables in this chapter concern the proof shell mode, and are the largest group. They are split into several subgroups. The first subgroup are commands invoked at various points. The second subgroup of variables are concerned with matching the output from the proof assistant. The final subgroup contains various hooks which you can set to add lisp customization to Proof General in various points (some of them are also used internally for behaviour you may wish to adjust).

Variables for configuring the proof shell are put into the customize group `proof-shell`.

These should be set in the shell mode configuration, before `proof-shell-config-done` is called.

To understand the way the proof assistant runs inside Emacs, you may want to refer to the `comint.el` (Command interpreter) package distributed with Emacs. This package controls several shell-like modes available in Emacs, including the `proof-shell-mode` and all specific shell modes derived from it.

4.1 Commands

Settings in this section configure Proof General with commands to send to the prover to activate certain actions.

`proof-prog-name` [Variable]

System command to run the proof assistant in the proof shell.
May contain arguments separated by spaces, but see also '`proof-prog-args`'.

Remark: if '`proof-prog-args`' is non-nil, then `proof-prog-name` is considered strictly: it must contain **only** the program name with no option, spaces are interpreted literally as part of the program name.

`proof-shell-auto-terminate-commands` [Variable]

Non-nil if Proof General should try to add terminator to every command.
If non-nil, whenever a command is sent to the prover using '`proof-shell-invisible-command`', Proof General will check to see if it ends with `proof-terminal-char`, and add it if not. If `proof-terminal-char` is nil, this has no effect.

`proof-shell-pre-sync-init-cmd` [Variable]

The command for configuring the proof process to gain synchronization.
This command is sent before Proof General's synchronization mechanism is engaged, to allow customization inside the process to help gain syncrhonization (e.g. engaging special markup).

It is better to configure the proof assistant for this purpose via command line options if possible, in which case this variable does not need to be set.

See also '`proof-shell-init-cmd`'.

`proof-shell-init-cmd` [Variable]

The command for initially configuring the proof process.
This command is sent to the process as soon as synchronization is gained (when an annotated prompt is first recognized). It can be used to configure the proof assistant in some way, or print a welcome message (since output before the first prompt is discarded).

See also '`proof-shell-pre-sync-init-cmd`'.

`proof-shell-restart-cmd` [Variable]

A command for re-initialising the proof process.

`proof-shell-quit-cmd` [Variable]
A command to quit the proof process. If nil, send EOF instead.

`proof-shell-quit-timeout` [Variable]
The number of seconds to wait after sending `proof-shell-quit-cmd`.
After this timeout, the proof shell will be killed off more rudely. If your proof assistant takes a long time to clean up (for example writing persistent databases out or the like), you may need to bump up this value.

`proof-shell-cd-cmd` [Variable]
Command to the proof assistant to change the working directory.
The format character '`%s`' is replaced with the directory, and the escape sequences in '`proof-shell-filename-escapes`' are applied to the filename.

This setting is used to define the function `proof-cd` which changes to the value of (`default-directory`) for script buffers. For files, the value of (`default-directory`) is simply the directory the file resides in.

NB: By default, `proof-cd` is called from `proof-activate-scripting-hook`, so that the prover switches to the directory of a proof script every time scripting begins.

`proof-shell-start-silent-cmd` [Variable]
Command to turn prover goals output off when sending many script commands.
If non-nil, Proof General will automatically issue this command to help speed up processing of long proof scripts. See also `proof-shell-stop-silent-cmd`. NB: terminator not added to command.

`proof-shell-stop-silent-cmd` [Variable]
Command to turn prover output on.
If non-nil, Proof General will automatically issue this command to help speed up processing of long proof scripts. See also `proof-shell-start-silent-cmd`. NB: Terminator not added to command.

`proof-shell-silent-threshold` [Variable]
Number of waiting commands in the proof queue needed to trigger silent mode.
Default is 2, but you can raise this in case switching silent mode on or off is particularly expensive (or make it ridiculously large to disable silent mode altogether).

See Chapter 8 [Handling Multiple Files], page 33, for more details about the final two settings in this group,

`proof-shell-inform-file-processed-cmd` [Variable]
Command to the proof assistant to tell it that a file has been processed.
The format character '`%s`' is replaced by a complete filename for a script file which has been fully processed interactively with Proof General. See '`proof-format-filename`' for other possibilities to process the filename.

This setting used to interface with the proof assistant's internal management of multiple files, so the proof assistant is kept aware of which files have been processed. Specifically, when scripting is deactivated in a completed buffer, it is added to Proof General's list of processed files, and the prover is told about it by issuing this command.

If this is set to nil, no command is issued.

See also: `proof-shell-inform-file-retracted-cmd`, `proof-shell-process-file`, `proof-shell-compute-new-files-list`.

`proof-shell-inform-file-retracted-cmd` [Variable]

Command to the proof assistant to tell it that a file has been retracted.
The format character '`%s`' is replaced by a complete filename for a script file which Proof General wants the prover to consider as not completely processed. See '`proof-format-filename`' for other possibilities to process the filename.

This is used to interface with the proof assistant's internal management of multiple files, so the proof assistant is kept aware of which files have been processed. Specifically, when scripting is activated, the file is removed from Proof General's list of processed files, and the prover is told about it by issuing this command. The action may cause the prover in turn to suggest to Proof General that files depending on this one are also unlocked.

If this is set to nil, no command is issued.

It is also possible to set this value to a function which will be invoked on the name of the retracted file, and should remove the ancestor files from '`proof-included-files-list`' by some other calculation.

See also: `proof-shell-inform-file-processed-cmd`, `proof-shell-process-file`, `proof-shell-compute-new-files-list`.

4.2 Script input to the shell

Generally, commands from the proof script are sent verbatim to the proof process running in the proof shell. For historical reasons, carriage returns are stripped by default. You can set `proof-shell-strip-crs-from-input` to adjust that. For more sophisticated pre-processing of the sent string, you may like to set `proof-shell-insert-hook`.

`proof-shell-strip-crs-from-input` [Variable]

If non-nil, replace carriage returns in every input with spaces.
This is enabled by default: it is appropriate for many systems based on human input, because several CR's can result in several prompts, which may mess up the display (or even worse, the synchronization).

If the prover can be set to output only one prompt for every chunk of input, then newlines can be retained in the input.

`proof-shell-insert-hook` [Variable]

Hooks run by '`proof-shell-insert`' before inserting a command.
Can be used to configure the proof assistant to the interface in various ways – for example, to observe or alter the commands sent to the prover, or to sneak in extra commands to configure the prover.

This hook is called inside a '`save-excursion`' with the `proof-shell-buffer` current, just before inserting and sending the text in the variable '`string`'. The hook can massage '`string`' or insert additional text directly into the `proof-shell-buffer`. Before sending '`string`', it will be stripped of carriage returns.

Additionally, the hook can examine the variable '`action`'. It will be a symbol, set to the callback command which is executed in the proof shell filter once '`string`' has been processed. The '`action`' variable suggests what class of command is about to be inserted:

```
'proof-done-invisible       A non-scripting command
'proof-done-advancing       A "forward" scripting command
'proof-done-retracting      A "backward" scripting command
'init-cmd                   Initialization command sent to prover
'interactive-input          Special interactive input direct to prover
```

Caveats: You should be very careful about setting this hook. Proof General relies on a careful synchronization with the process between inputs and outputs. It expects to see a prompt for

each input it sends from the queue. If you add extra input here and it causes more prompts than expected, things will break! Extending the variable '`string`' may be safer than inserting text directly, since it is stripped of carriage returns before being sent.

Example uses: *lego* uses this hook for setting the pretty printer width if the window width has changed; Plastic uses it to remove literate-style markup from '`string`'. The x-symbol support uses this hook to convert special characters into tokens for the proof assistant.

4.3 Settings for matching various output from proof process

These settings control the way Proof General reacts to process output. The single most important setting is `proof-shell-annotated-prompt-regexp`, which **must** be set as part of the prover configuraton. This is used to configure the communication with the prover process.

`proof-shell-wakeup-char` [Variable]
A special character which terminates an annotated prompt.
Set to nil if proof assistant does not support annotated prompts.

`pg-subterm-first-special-char` [Variable]
First special character.
Codes above this character can have special meaning to Proof General, and are stripped from the prover's output strings. Leave unset if no special characters are being used.

`proof-shell-prompt-pattern` [Variable]
Proof shell's value for comint-prompt-pattern, which see.
NB!! This pattern is just for interaction in comint (shell buffer). You don't really need to set it. The important one to set is '`proof-shell-annotated-prompt-regexp`', which see.

`proof-shell-annotated-prompt-regexp` [Variable]
Regexp matching a (possibly annotated) prompt pattern.

this IS THE *most important setting* TO *configure*!!

Output is grabbed between pairs of lines matching this regexp, and the appearance of this regexp is used by Proof General to recognize when the prover has finished processing a command.

To help speed up matching you may be able to annotate the proof assistant prompt with a special character not appearing in ordinary output. The special character should appear in this regexp, and should be the value of `proof-shell-wakeup-char`.

`proof-shell-abort-goal-regexp` [Variable]
Regexp matching output from an aborted proof.

`proof-shell-error-regexp` [Variable]
Regexp matching an error report from the proof assistant.

We assume that an error message corresponds to a failure in the last proof command executed. So don't match mere warning messages with this regexp. Moreover, an error message should not be matched as an eager annotation (see `proof-shell-eager-annotation-start`) otherwise it will be lost.

Error messages are considered to begin from `proof-shell-error-regexp` and continue until the next prompt. The variable '`proof-shell-truncate-before-error`' controls whether text before the error message is displayed.

The engine matches interrupts before errors, see `proof-shell-interrupt-regexp`.

It is safe to leave this variable unset (as nil).

`proof-shell-interrupt-regexp` [Variable]
Regexp matching output indicating the assistant was interrupted.
We assume that an interrupt message corresponds to a failure in the last proof command executed. So don't match mere warning messages with this regexp. Moreover, an interrupt message should not be matched as an eager annotation (see `proof-shell-eager-annotation-start`) otherwise it will be lost.

The engine matches interrupts before errors, see `proof-shell-error-regexp`.

It is safe to leave this variable unset (as nil).

`proof-shell-truncate-before-error` [Variable]
Non-nil means truncate output that appears before error messages.
If nil, the whole output that the prover generated before the last error message will be shown.

NB: the default setting for this is '`t`' to be compatible with behaviour in Proof General before version 3.4. The more obvious setting for new instances is probably '`nil`'.

Interrupt messages are treated in the same way. See '`proof-shell-error-regexp`' and '`proof-shell-interrupt-regexp`'.

`proof-shell-proof-completed-regexp` [Variable]
Regexp matching output indicating a finished proof.

When output which matches this regexp is seen, we clear the goals buffer in case this is not also marked up as a '`goals`' type of message.

We also enable the QED function (save a proof) and we may automatically close off the proof region if another goal appears before a save command, depending on whether the prover supports nested proofs or not.

`proof-shell-start-goals-regexp` [Variable]
Regexp matching the start of the proof state output.
This is an important setting. Output between '`proof-shell-start-goals-regexp`' and '`proof-shell-end-goals-regexp`' will be pasted into the goals buffer and possibly analysed further for proof-by-pointing markup.

`proof-shell-end-goals-regexp` [Variable]
Regexp matching the end of the proof state output, or nil.
If nil, just use the rest of the output following `proof-shell-start-goals-regexp`.

`proof-shell-assumption-regexp` [Variable]
A regular expression matching the name of assumptions.

At the moment, this setting is not used in the generic Proof General.

Future use may providea generic implementation for '`pg-topterm-goalhyplit-fn`', used to help parse the goals buffer to annotate it for proof by pointing.

4.4 Settings for matching urgent messages from proof process

Among the various dialogue messages that the proof assistant outputs during proof, Proof General can consider certain messages to be "urgent". When processing many lines of a proof, Proof General will normally supress the output, waiting until the final message appears before displaying anything to the user. Urgent messages escape this: typically they include messages that the prover wants the user to notice, for example, perhaps, file loading messages, or timing statistics.

So that Proof General notices, these urgent messages should be marked-up with "eager" annotations.

`proof-shell-eager-annotation-start` [Variable]
Eager annotation field start. A regular expression or nil.
An "eager annotation indicates" to Proof General that some following output should be displayed (or processed) immediately and not accumulated for parsing later. Note that this affects processing of output which is ordinarily accumulated: output which appears before the eager annotation start will be discarded.

The start/end annotations can be used to hilight the output, but are stripped from display of the message in the minibuffer.

It is useful to recognize (starts of) warnings or file-reading messages with this regexp. You must also recognize any special messages from the prover to PG with this regexp (e.g. '`proof-shell-clear-goals-regexp`', '`proof-shell-retract-files-regexp`', etc.)

See also '`proof-shell-eager-annotation-start-length`', '`proof-shell-eager-annotation-end`'.

Set to nil to disable this feature.

`proof-shell-eager-annotation-start-length` [Variable]
Maximum length of an eager annotation start.
Must be set to the maximum length of the text that may match '`proof-shell-eager-annotation-start`' (at least 1). If this value is too low, eager annotations may be lost!

This value is used internally by Proof General to optimize the process filter to avoid unnecessary searching.

`proof-shell-eager-annotation-end` [Variable]
Eager annotation field end. A regular expression or nil.
An eager annotation indicates to Emacs that some following output should be displayed or processed immediately.

See also '`proof-shell-eager-annotation-start`'.

It is nice to recognize (ends of) warnings or file-reading messages with this regexp. You must also recognize (ends of) any special messages from the prover to PG with this regexp (e.g. '`proof-shell-clear-goals-regexp`', '`proof-shell-retract-files-regexp`', etc.)

The default value is `"\n"` to match up to the end of the line.

The default action for urgent messages is to display them in the response buffer, highlighted. But we also allow for some control messages, issued from the proof assistant to Proof General and not intended for the user to see. These are recognized in the same way as urgent messages (marked with eager annotations), so they will be acted on as soon as they are issued by the prover.

`proof-shell-clear-response-regexp` [Variable]
Regexp matching output telling Proof General to clear the response buffer.
This feature is useful to give the prover more control over what output is shown to the user.
Set to nil to disable.

`proof-shell-clear-goals-regexp` [Variable]
Regexp matching output telling Proof General to clear the goals buffer.
This feature is useful to give the prover more control over what output is shown to the user.
Set to nil to disable.

`proof-shell-set-elisp-variable-regexp` [Variable]
Matches output telling Proof General to set some variable.
This allows the proof assistant to configure Proof General directly and dynamically. (It's also a fantastic backdoor security risk).

If the regexp matches output from the proof assistant, there should be two match strings: (`match-string` 1) should be the name of the elisp variable to be set, and (`match-string` 2) should be the value of the variable (which will be evaluated as a Lisp expression).

A good markup for the second string is to delimit with #'s, since these are not valid syntax for elisp evaluation.

Elisp errors will be trapped when evaluating; set `proof-general-debug` to be informed when this happens.

Example uses are to adjust PG's internal copies of proof assistant's settings, or to make automatic dynamic syntax adjustments in Emacs to match changes in theory, etc.

If you pick a dummy variable name (e.g. '`proof-dummy-setting`') you can just evaluation arbitrary elisp expressions for their side effects, to adjust menu entries, or even launch auxiliary programs. But use with care – there is no protection against catastrophic elisp!

This setting could also be used to move some configuration settings from PG to the prover, but this is not really supported (most settings must be made before this mechanism will work). In future, the PG standard protocol, *pgip*, will use this mechanism for making all settings.

`proof-shell-theorem-dependency-list-regexp` [Variable]

Matches output telling Proof General about dependencies.
This is to allow navigation and display of dependency information. The output from the prover should be a message with the form

dependencies `OF X Y Z ARE A B C`

with X Y Z, A B C separated by whitespace or somehow else (see '`proof-shell-theorem-dependency-list-split`'. This variable should be set to a regexp to match the overall message (which should be an urgent message), with two sub-matches for X Y Z and A B C.

This is an experimental feature, currently work-in-progress.

Two important control messages are recognized by `proof-shell-process-file` and `proof-shell-retract-files-regexp`, used for synchronizing Proof General with a file loading mechanism built into the proof assistant. See Chapter 8 [Handling Multiple Files], page 33, for more details about how to use the final four settings described here.

`proof-shell-process-file` [Variable]

A pair (*regexp* . *function*) to match a processed file name.

If *regexp* matches output, then the function *function* is invoked on the output string chunk. It must return the name of a script file (with complete path) that the system has successfully processed. In practice, *function* is likely to inspect the match data. If it returns the empty string, the file name of the scripting buffer is used instead. If it returns nil, no action is taken.

Care has to be taken in case the prover only reports on compiled versions of files it is processing. In this case, *function* needs to reconstruct the corresponding script file name. The new (true) file name is added to the front of '`proof-included-files-list`'.

`proof-shell-retract-files-regexp` [Variable]

Matches a message that the prover has retracted a file.

At this stage, Proof General's view of the processed files is out of date and needs to be updated with the help of the function '`proof-shell-compute-new-files-list`'.

`proof-shell-compute-new-files-list` [Variable]

Function to update '`proof-included-files list`'.

It needs to return an up to date list of all processed files. Its output is stored in '`proof-included-files-list`'. Its input is the string of which '`proof-shell-retract-files-regexp`' matched a substring. In practice, this function is likely to inspect the previous (global) variable '`proof-included-files-list`' and the match data triggered by '`proof-shell-retract-files-regexp`'.

`proof-cannot-reopen-processed-files` [Variable]

Non-nil if the prover allows re-opening of already processed files.

If the user has used Proof General to process a file incrementally, then PG will retain the spans recording undo history in the buffer corresponding to that file (provided it remains visited in Emacs).

If the prover allows, it will be possible to undo to a position within this file. If the prover does **not** allow this, this variable should be set non-nil, so that when a completed file is activated for scripting (to do undo operations), the whole history is discarded.

`proof-shell-require-command-regexp` [Variable]

A regular expression to match a Require-type of command, or nil.
If set to a regexp, then '`proof-done-advancing-require-function`' should also be set, and will be called immediately after the match.

This can be used to adjust '`proof-included-files-list`' based on the lines of script that have been processed/parsed, rather than relying on information from the prover.

`proof-done-advancing-require-function` [Variable]

Used in '`proof-done-advancing`', see '`proof-shell-require-command-regexp`'.
The function is passed the span and the command as arguments.

4.5 Hooks and other settings

`proof-shell-filename-escapes` [Variable]

A list of escapes that are applied to %s for filenames.
A list of cons cells, car of which is string to be replaced by the cdr. For example, when directories are sent to Isabelle, HOL, and Coq, they appear inside ML strings and the backslash character and quote characters must be escaped. The setting

```
'(("\\" . "\\\\")
  ("\"" . "\\\""))
```

achieves this. This does not apply to *lego*, which does not need backslash escapes and does not allow filenames with quote characters.

This setting is used inside the function '`proof-format-filename`'.

`proof-shell-process-connection-type` [Variable]

The value of '`process-connection-type`' for the proof shell.
Set non-nil for ptys, nil for pipes. The default (and preferred) option is to use pty communication. However there is a long-standing backslash/long line problem with Solaris which gives a mess of ^G characters when some input is sent which has a \ in the 256th position. So we select pipes by default if it seems like we're on Solaris. We do not force pipes everywhere because this risks loss of data.

`proof-shell-handle-error-or-interrupt-hook` [Variable]

Run after an error or interrupt has been reported in the response buffer.
Hook functions may inspect '`proof-shell-error-or-interrupt-seen`' to determine whether the cause was an error or interrupt. Possible values for this hook include:

`proof-goto-end-of-locked-on-error-if-pos-not-visible-in-window`
`proof-goto-end-of-locked-if-pos-not-visible-in-window`

which move the cursor in the scripting buffer on an error or error/interrupt.

Remark: This hook is called from shell buffer. If you want to do something in scripting buffer, '`save-excursion`' and/or '`set-buffer`'.

`proof-shell-pre-interrupt-hook` [Variable]

Run immediately after '`comint-interrupt-subjob`' is called. This
hook is added to allow customization for systems that query the user before returning to the top level.

`proof-shell-process-output-system-specific` [Variable]

Set this variable to handle system specific output.
Errors, start of proofs, abortions of proofs and completions of proofs are recognised in the function '`proof-shell-process-output`'. All other output from the proof engine is simply reported to the user in the *response* buffer.

To catch further special cases, set this variable to a pair of functions '(condf . actf). Both are given (cmd string) as arguments. '`cmd`' is a string containing the currently processed command. '`string`' is the response from the proof system. To change the behaviour of '`proof-shell-process-output`', (condf cmd string) must return a non-nil value. Then (actf cmd string) is invoked.

See the documentation of '`proof-shell-process-output`' for the required output format.

5 Goals Buffer Settings

The goals buffer settings allow configuration of Proof General for proof by pointing or similar features. See the Proof General documentation web page for a link to the technical report ECS-LFCS-97-368 which hints at how to use these settings.

`pg-goals-change-goal` [Variable]
Command to change to the goal '`%s`'.

`pbp-goal-command` [Variable]
Command sent when '`pg-goals-button-action`' is requested on a goal.

`pbp-hyp-command` [Variable]
Command sent when '`pg-goals-button-action`' is requested on an assumption.

`pg-goals-error-regexp` [Variable]
Regexp indicating that the proof process has identified an error.

`proof-shell-result-start` [Variable]
Regexp matching start of an output from the prover after pbp commands.
In particular, after a '`pbp-goal-command`' or a '`pbp-hyp-command`'.

`proof-shell-result-end` [Variable]
Regexp matching end of output from the prover after pbp commands.
In particular, after a '`pbp-goal-command`' or a '`pbp-hyp-command`'.

`pg-subterm-start-char` [Variable]
Opening special character for subterm markup.
Subsequent special characters with values **below** `pg-subterm-first-special-char` are assumed to be subterm position indicators. Annotations should be finished with `pg-subterm-sep-char`; the end of the concrete syntax is indicated by `pg-subterm-end-char`.

If '`pg-subterm-start-char`' is nil, subterm markup is disabled.

See doc of '`pg-assoc-analyse-structure`' for more details of subterm and proof-by-pointing markup mechanisms..

`pg-subterm-sep-char` [Variable]
Finishing special for a subterm markup.
See doc of '`pg-subterm-start-char`'.

`pg-topterm-regexp` [Variable]
Annotation regexp that indicates the beginning of a "top" element.
A "top" element may be a sub-goal to be proved or a named hypothesis, for example. It could also be a literal command to insert and send back to the prover.

The function '`pg-topterm-goalhyplit-fn`' examines text following this special character, to determine what kind of top element it is.

This setting is also used to see if proof-by-pointing features are configured. If it is unset, some of the code for parsing the prover output is disabled.

`pg-subterm-end-char` [Variable]
Closing special character for subterm markup.
See '`pg-subterm-start-char`'.

6 Splash Screen Settings

The splash screen can be configured, in a rather limited way.

`proof-splash-time` [Variable]
: Minimum number of seconds to display splash screen for.
The splash screen may be displayed for a wee while longer than this, depending on how long it takes the machine to initialise Proof General.

`proof-splash-contents` [Variable]
: Evaluated to configure splash screen displayed when entering Proof General.
A list of the screen contents. If an element is a string or an image specifier, it is displayed centred on the window on its own line. If it is nil, a new line is inserted.

7 Global Constants

The settings here are internal constants used by Proof General. You don't need to configure these for your proof assistant unless you want to modify or extend the defaults.

`proof-general-name` [Variable]
Proof General name used internally and in menu titles.

`proof-general-home-page` [User Option]
Web address for Proof General.

The default value is `"http://proofgeneral.inf.ed.ac.uk"`.

`proof-universal-keys` [Variable]
List of key bindings made for the script, goals and response buffer.
Elements of the list are tuples '`(k . f)`' where '`k`' is a key binding (vector) and '`f`' the designated function.

8 Handling Multiple Files

Large proof developments are typically spread across multiple files. Many provers support such developments by keeping track of dependencies and automatically processing scripts. Proof General supports this mechanism. The user's point of view is considered in the user manual. Here, we describe the more technical nitty gritty. This is what you need to know when you customise another proof assistant to work with Proof General.

Documentation for the configuration settings mentioned here appears in the previous sections, this section is intended to help explain the use of those settings.

Proof General maintains a list `proof-included-files-list` of files which it thinks have been processed by the proof assistant. When a file which is on this list is visited in Emacs, it will be coloured entirely blue to indicate that it has been processed. No editing of the file will be allowed (unless `proof-strict-read-only` allows it).

`proof-included-files-list` [Variable]

List of files currently included in proof process.
This list contains files in canonical truename format (see '`file-truename`').

Whenever a new file is being processed, it gets added to this list via the `proof-shell-process-file` configuration settings. When the prover retracts a file, this list is resynchronised via the `proof-shell-retract-files-regexp` and `proof-shell-compute-new-files-list` configuration settings.

Only files which have been **fully** processed should be included here. Proof General itself will automatically add the filenames of a script buffer which has been completely read when scripting is deactivated. It will automatically remove the filename of a script buffer which is completely unread when scripting is deactivated.

NB: Currently there is no generic provision for removing files which are only partly read-in due to an error, so ideally the proof assistant should only output a processed message when a file has been successfully read.

The way that `proof-included-files-list` is maintained is the key to multiple file management. (But you should not set this variable directly, it is managed via the configuration settings).

There is a range of strategies for managing multiple files. Ideally, file dependencies should be managed by the proof assistant. Proof General will use the prover's low-level commands to process a whole file and its requirements non-interactively, without going through script management. So that the user knows which files have been processed, the proof assistant should issue messages which Proof General can recognize ("file `foo` has been processed") — see `proof-shell-process-file`. When the user wants to edit a file which has been processed, the file must be retracted (unlocked). The proof assistant should provide a command corresponding to this action, which undoes a given file and all its dependencies. As each file is undone, a message should be issued which Proof General can recognize ("file `foo` has been undone") – see `proof-shell-retract-files-regexp`. (The function `proof-shell-compute-new-files-list` should be set to calculate the new value for `proof-included-files-list` after a retract message has been seen).

As well as this communication from the assistant to Proof General about processed or retracted files, Proof General can communicate the other way: it will tell the proof assistant when it has processed or retracted a file via script management. This is because during script management, the proof assistant may not be aware that it is actually dealing with a file of proof commands (rather than just terminal input).

Proof General will provide this information in two special instances. First, when scripting is turned off in a file that has been completely processed, Proof General will tell the proof assistant using `proof-shell-inform-file-processed-cmd`. Second, when scripting is turned on in a file

which is completely processed, Proof General will tell the proof assistant to reconsider: the file should not be considered completely processed yet. This uses the setting `proof-shell-inform-file-retracted-cmd`. This second, retracting, case might lead to a series of messages from the prover telling Proof General to unlock files which depend on the present one, again via `proof-shell-retract-files-regexp`.

The special case for retracting is the primary file the user wishes to edit: this is automatically removed from `proof-included-files-list`, but it depends on the proof assistant whether or not it is possible to revert to a partially processed version of the file (or "undo into" it). This is the reason for the setting `proof-cannot-reopen-processed-files`. If this is non-nil, any attempt to undo a fully processed file will unlock the entire file (whether or not Proof General itself has history information for the file).

What we have described so far is the ideal case, but it may require some support from the proof assistant to set up (for example, if file-level undo is not normally supported, or the messages during file processing are not suitable). Moreover, some proof assistants may not have file handling with dependencies, or may have a particularly simple case of a linear context: each file depends on all the ones processed before it. Proof General allows you a shortcut to get automatic management of multiple files in these cases by setting the flag `proof-auto-multiple-files`. This setting is probably an approximation to the right thing for any proof assistant. More files than necessary will be retracted if the prover has a tree-like file dependency rather than a linear one.

Finally, we should mention how Proof General recognizes file processing messages from the proof assistant. Proof General considers *output* delimited by the the two regular expressions `proof-shell-eager-annotation-start` and `proof-shell-eager-annotation-end` as being important. It displays the *output* in the Response buffer and analyses the contents further. Among other important messages characterised by these regular expressions (warnings, errors, or information), the prover can tell the interface whenever it processes or retracts a file.

To summarize, the settings for multiple file management that may be customized are as follows. To recognize file-processing, `proof-shell-process-file`. To recognize messages about file undoing, `proof-shell-retract-files-regexp` and `proof-shell-compute-new-files-list`. See Section 4.4 [Settings for matching urgent messages from proof process], page 21. To tell the prover about files handled with script management, use `proof-shell-inform-file-processed-cmd` and `proof-shell-inform-file-retracted-cmd`. See Section 4.1 [Proof shell commands], page 17. If your prover does not allow re-opening of closed files, set `proof-cannot-reopen-processed-files` to `t`. Finally, set the flag `proof-auto-multiple-files` for a automatic approximation to multiple file handling. See Chapter 3 [Proof Script Settings], page 9.

9 Configuring Editing Syntax

Emacs has some standard settings which configure the syntax of major modes. The main setting is the *syntax table*, which determines the syntax of programming elements such as strings, comments, and parentheses. To configure the syntax table, you can either write calls to `modify-syntax-entry` in your mode functions, or set the following variables to contain the tables for each mode. (The main mode to be concerned about is of course the proof script, where user editing takes place).

`proof-script-syntax-table-entries` [Variable]
List of syntax table entries for proof script mode.
A flat list of the form

```
(char syncode char syncode ...)
```

See doc of '`modify-syntax-entry`' for details of characters and syntax codes.

At present this is used only by the '`proof-easy-config`' macro.

`proof-shell-syntax-table-entries` [Variable]
List of syntax table entries for proof script mode.
A flat list of the form

```
(char syncode char syncode ...)
```

See doc of '`modify-syntax-entry`' for details of characters and syntax codes.

At present this is used only by the '`proof-easy-config`' macro.

Some additional useful settings are:

`comment-quote-nested` [Variable]
Non-nil if nested comments should be quoted. This should be locally set by each major mode if needed. The default setting is non-nil: modes which allow nested comments may set this to nil.

`outline-regexp` [Variable]
Regular expression to match the beginning of a heading. Any line whose beginning matches this regexp is considered to start a heading.

`outline-heading-end-regexp` [Variable]
Regular expression to match the beginning of a heading. Any line whose beginning matches this regexp is considered to start a heading.

10 Configuring Font Lock

Support for Font Lock in Proof General is described in the user manual (see the *Syntax highlighting* section). To configure Font Lock for a new proof assistant, you need to set the variable `font-lock-keywords` in each of the mode functions you want highlighting for. Proof General will automatically install these settings, and use font lock minor mode (for syntax highlighting as you type) in script buffers.

To understand its format, check the documentation of `font-lock-keywords` inside Emacs.

Instead of setting `font-lock-keywords` in each mode function, you can use the following four variables to make the settings in place. This is particularly useful if use the easy configuration mechanism for Proof General, see Section 1.2 [Demonstration instance and easy configuration], page 4.

`proof-script-font-lock-keywords` [Variable]
Value of '`font-lock-keywords`' used to fontify proof scripts.
This is currently used only by '`proof-easy-config`' mechanism, to set '`font-lock-keywords`' before calling `proof-config-done`. See also proof-{shell,resp,goals}-`font-lock-keywords`.

`proof-shell-font-lock-keywords` [Variable]
Value of '`font-lock-keywords`' used to fontify the proof shell.
This is currently used only by '`proof-easy-config`' mechanism, to set '`font-lock-keywords`' before calling '`proof-config-done`'. See also proof-{script,resp,goals}-`font-lock-keywords`.

`proof-goals-font-lock-keywords` [Variable]
Value of '`font-lock-keywords`' used to fontify the goals output.
NB: the goals output is not kept in font lock mode because the fontification may rely on annotations which are erased before displaying. This means internal functions of PG must be used to display to the goals buffer to ensure fontification is done! This is currently used only by '`proof-easy-config`' mechanism, to set '`font-lock-keywords`' before calling `proof-config-done`. See also proof-{script,shell,resp}-`font-lock-keywords`.

`proof-resp-font-lock-keywords` [Variable]
Value of '`font-lock-keywords`' used to fontify the response output.
NB: the goals output is not kept in font lock mode because the fontification may rely on annotations which are erased before displaying. This means internal functions of PG must be used to display to the goals buffer to ensure fontification is done! This is currently used only by `proof-easy-config` mechanism, to set `font-lock-keywords` before calling `proof-config-done`. See also proof-{script,shell,resp}-`font-lock-keywords`.

Proof General provides a special function, `proof-zap-commas`, for tweaking the font lock behaviour of provers which have declarations of the form `x,y,z:Ty`. This function removes highlighting on the commas, and can be added as the last element of `font-lock-keywords`. Further manipulation of font lock behaviour can be achieved via two hook functions which are run before and after fontifying the output buffers.

`proof-zap-commas` *limit* [Function]
Remove the face of all '`,`' from point to *limit*.
Meant to be used from '`font-lock-keywords`'.

`pg-before-fontify-output-hook` [Variable]
This hook is called before fontifying a region in an output buffer.
A function on this hook can alter the region of the buffer within the current restriction, and

must return the final value of (`point-max`). [This hook is presently only used by phox-symlock].

`pg-after-fontify-output-hook` [Variable]
This hook is called before fonfitying a region in an output buffer.
[This hook is presently only used by Isabelle].

11 Configuring X-Symbol

The X-Symbol package is described in the Proof General user manual. To configure X-Symbol for Proof General, you must understand a little bit of how X-Symbol works: read the documentation that is supplied with it.

The basic task is to set up a *token language* for your proof assistant. If your assistant is stored in the subdirectory *myprover*, the token language will be called *myprover* and be defined in a file '`x-symbol-`*`myprover`*`.el`' which is automatically loaded by X-Symbol. The name of the token language mode will be *`myprover`*`sym`.

Proof General will check that the file '`x-symbol-`*`myprover`*`.el`' exists and set up X-Symbol to load it. The token language file must define a number of standard settings, and X-Symbol will give warnings if any of them are missing.

Apart from the token language file, there are several settings for X-Symbol which you can set in the usual configuration file '*`myprover`*`.el`'. These settings are optional.

`proof-xsym-activate-command` [Variable]
: Command to activate token input/output for X-Symbol.
If non-nil, this command is sent to the proof assistant when X-Symbol support is activated.

`proof-xsym-deactivate-command` [Variable]
: Command to deactivate token input/output for X-Symbol.
If non-nil, this command is sent to the proof assistant when X-Symbol support is deactivated.

We expect tokens to be used uniformly, so that along with each script mode buffer, the response buffer and goals buffer also invoke X-Symbol to display special characters in the same token language. This happens automatically. If you want additional modes to use X-Symbol with the token language for your proof assistant, you can set `proof-xsym-extra-modes`.

`proof-xsym-extra-modes` [Variable]
: List of additional mode names to use X-Symbol with Proof General tokens.
These modes will have X-Symbol enabled for the proof assistant token language, in addition to the four modes for Proof General (script, shell, response, pbp).

Set this variable if you want additional modes to also display tokens (for example, editing documentation or source code files).

12 Writing More Lisp Code

You may want to add some extra features to your instance of Proof General which are not supported in the generic core. To do this, you can use the settings described above, plus a small number of fundamental functions in Proof General which you can consider as exported in the generic interface. Be careful using more functions than are mentioned here because the internals of Proof General may change between versions.

12.1 Default values for generic settings

Several generic settings are defined using `defpgcustom` in '`proof-config.el`'. This introduces settings of the form `<PA>-name` for each proof assistant *PA*.

To set the default value for these settings in prover-specific cases, you should use the special `defpgdefault` macro:

`defpgdefault` [Macro]

Set default for the proof assistant specific variable <PA>-*sym* to *value*.
This should be used in prover-specific code to alter the default values for prover specific settings.

Usage: (defpgdefault SYM *value*)

In your prover-specific code you can simply use the setting `<PA>-sym` directly, i.e., write `myprover-home-page`.

In the generic code, you can use a macro, writing `(proof-ass home-page)` to refer to the `<PA>-home-page` setting for the currently running instance of Proof General.

See Section 13.3 [Configuration variable mechanisms], page 46, for more details on this mechanism.

12.2 Adding prover-specific configurations

Apart from the generic settings, your prover instance will probably need some specific customizable settings.

Defining new prover-specific settings using customize is pretty easy. You should do it at least for your prover-specific user options.

The code in '`proof-site.el`' provides each prover with two customization groups automatically (based on the name of the assistant): `<PA>` for user options for prover *PA* and `<PA>-config` for configuration of prover *PA*. Typically `<PA>-config` holds settings which are constants but which may be nice to tweak.

The first group appears in the menu

```
ProofGeneral -> Advanced -> Customize -> <PA>
```

The second group appears in the menu:

```
ProofGeneral -> Internals -> <PA> config
```

A typical use of `defcustom` looks like this:

```
(defcustom myprover-search-page
  "http://findtheorem.myprover.org"
  "URL of search web page for myprover."
  :type 'string
  :group 'myprover-config)
```

This introduces a new customizable setting, which you might use to make a menu entry, for example. The default value is the string `"http://findtheorem.myprover.org"`.

12.3 Useful variables

In 'proof-site', some architecture flags are defined. These can be used to write conditional pieces of code for different Emacs and operating systems. They are referred to mainly in 'proof-compat' (which helps to keep the architecture and version dependent code in one place).

`proof-running-on-win32` [Variable]
Non-nil if Proof General is running on a windows variant system.

12.4 Useful functions and macros

The recommended functions you may invoke are these:

- Any of the interactive commands (i.e. anything you can invoke with `M-x`, including all key-bindings)
- Any of the internal functions and macros mentioned below

To insert text into the current (usually script) buffer, the function `proof-insert` is useful. There's also a handy macro `proof-defshortcut` for defining shortcut functions using it.

`proof-insert` *text* [Function]
Insert *text* into the current buffer.
text may include these special characters:

```
%p  - place the point here after input
```

Any other %-prefixed character inserts itself.

`proof-defshortcut` [Macro]
Define shortcut function FN to insert *string*, optional keydef KEY.
This is intended for defining proof assistant specific functions. *string* is inserted using 'proof-insert', which see. KEY is added onto `proof-assistant` map.

The function `proof-shell-invisible-command` is a useful utility for sending a single command to the process. You should use this to implement user-level or internal functions rather than attempting to directly manipulate the proof action list, or insert into the shell buffer.

`proof-shell-invisible-command` *cmd* **&optional** *wait* [Function]
Send *cmd* to the proof process.
The *cmd* is 'invisible' in the sense that it is not recorded in buffer. *cmd* may be a string or a string-yielding expression.

Automatically add `proof-terminal-char` if necessary, examining proof-shell-no-auto-terminate-commands.

By default, let the command be processed asynchronously. But if optional *wait* command is non-nil, wait for processing to finish before and after sending the command.

In case *cmd* is (or yields) nil, do nothing.

There are several handy macros to help you define functions which invoke `proof-shell-invisible-command`.

`proof-definvisible` [Macro]
Define function FN to send *string* to proof assistant, optional keydef KEY.
This is intended for defining proof assistant specific functions. *string* is sent using `proof-shell-invisible-command`, which see. *string* may be a string or a function which returns a string. KEY is added onto `proof-assistant` map.

`proof-define-assistant-command` [Macro]
Define FN (docstring DOC) to send *body* to prover, based on *cmdvar*.
body defaults to *cmdvar*, a variable.

`proof-define-assistant-command-witharg` [Macro]

Define command FN to prompt for string *cmdvar* to proof assistant.
cmdvar is a variable holding a function or string. Automatically has history.

`proof-format-filename` *string filename* [Function]

Format *string* by replacing quoted chars by escaped version of *filename*.

%e uses the canonicalized expanded version of filename (including directory, using `default-directory` – see '`expand-file-name`').

%r uses the unadjusted (possibly relative) version of *filename*.

%m ('module') uses the basename of the file, without directory or extension.

%s means the same as %e.

Using %e can avoid problems with dumb proof assistants who don't understand `~`, for example.

For all these cases, the escapes in '`proof-shell-filename-escapes`' are processed.

If *string* is in fact a function, instead invoke it on *filename* and return the resulting (string) value.

13 Internals of Proof General

This chapter sketches some of the internal functions and variables of Proof General, to help developers who wish to understand or modify the code.

Most of the documentation below is generated automatically from the comments in the code. Because Emacs lisp is interpreted and self-documenting, the best way to find your way around the source is inside Emacs once Proof General is loaded. Read the source files, and use functions such as `C-h v` and `C-h f`.

The code is split into files. The following sections document the important files, kept in the 'generic/' subdirectory.

13.1 Spans

Spans are an abstraction of XEmacs *extents* used to help bridge the gulf between GNU Emacs and XEmacs. In GNU Emacs, spans are implemented using *overlays*.

See the files 'span-extent.el' and 'span-overlay.el' for the implementation of the common interface in each case.

13.2 Proof General site configuration

The file 'proof-site.el' contains the initial configuration for Proof General for the site (or user) and the choice of provers.

The first part of the configuration is to set `proof-home-directory` to the directory that 'proof-site.el' is located in, or to the variable of the environment variable `PROOFGENERAL_HOME` if that is set.

`proof-home-directory` [Variable]
: Directory where Proof General is installed. Ends with slash.
Default value taken from environment variable 'PROOFGENERAL_HOME' if set, otherwise based on where the file 'proof-site.el' was loaded from. You can use customize to set this variable.

Further directory variables allow the files of Proof General to be split up and installed across a system if need be, rather than under the `proof-home-directory` root.

`proof-images-directory` [Variable]
: Where Proof General image files are installed. Ends with slash.

`proof-info-directory` [Variable]
: Where Proof General Info files are installed. Ends with slash.

After defining these settings, we define a *mode stub* for each proof assistant enabled. The mode stub will autoload Proof General for the right proof assistant when a file is visited with the corresponding extension. The proof assistants enabled are the ones listed in the `proof-assistants` setting.

`proof-assistants` [User Option]
: Choice of proof assistants to use with Proof General.
A list of symbols chosen from: `'isar 'coq 'phox 'lego 'plastic`. If nil, the default will be ALL available proof assistants.

Each proof assistant defines its own instance of Proof General, providing session control, script management, etc. Proof General will be started automatically for the assistants chosen here. To avoid accidently invoking a proof assistant you don't have, only select the proof assistants you (or your site) may need.

You can select which proof assistants you want by setting this variable before '`proof-site.el`' is loaded, or by setting the environment variable '`PROOFGENERAL_ASSISTANTS`' to the symbols you want, for example `"lego isa"`. Or you can edit the file '`proof-site.el`' itself.

Note: to change proof assistant, you must start a new Emacs session.

The default value is `nil`.

The file '`proof-site.el`' also defines a version variable.

`proof-general-version` [Variable]
Version string identifying Proof General release.

13.3 Configuration variable mechanisms

The file '`proof-config.el`' defines the configuration variables for Proof General, including instantiation parameters and user options. See previous chapters for details of its contents. Here we mention some conventions for declaring user options.

Global user options and instantiation parameters are declared using `defcustom` as usual. User options should have '`*`' as the first character of their docstrings (standard Emacs convention) and live in the customize group `proof-user-options`. See '`proof-config.el`' for the groups for instantiation parameters.

User options which are generic (having separate instances for each prover) and instantiation parameters (by definition generic) can be declared using the special macro `defpgcustom`. It is used in the same way as `defcustom`, except that the symbol declared will automatically be prefixed by the current proof assistant symbol.

`defpgcustom` [Macro]
Define a new customization variable <PA>-*sym* for the current proof assistant.
The function proof-assistant-<SYM> is also defined, which can be used in the generic portion of Proof General to set and retrieve the value for the current p.a. Arguments as for '`defcustom`', which see.

Usage: (defpgcustom SYM &rest *args*).

In specific instances of Proof General, the macro `defpgdefault` can be used to give a default value for a generic setting.

`defpgdefault` [Macro]
Set default for the proof assistant specific variable <PA>-*sym* to *value*.
This should be used in prover-specific code to alter the default values for prover specific settings.

Usage: (defpgdefault SYM *value*)

All new instantiation variables are best declared using the `defpgcustom` mechanism (old code may be converted gradually). Customizations which are liable to be different for different instances of Proof General are also best declared in this way. An example is the use of X Symbol, controlled by *PA*`-x-symbol-enable`, since it works poorly or not at all with some provers.

To access the generic settings, the following four functions and macros are useful.

`proof-ass` [Macro]
Return the value for SYM for the current prover.
This macro should only be invoked once a specific prover is engaged.

`proof-ass-sym` [Macro]
Return the symbol for SYM for the current prover. SYM not evaluated.
This macro should only be called once a specific prover is known.

`proof-ass-symv` [Macro]
Return the symbol for SYM for the current prover. SYM evaluated.
This macro should only be invoked once a specific prover is engaged.

If changing a user option setting amounts to more than just setting a variable (it may have some dynamic effect), we can set the `custom-set` property for the variable to the function `proof-set-value` which does an ordinary `set-default` to set the variable, and then calls a function with the same name as the variable, to do whatever is necessary according to the new value for the variable.

There are several settings which can be switched on or off by the user, which use this `proof-set-value` mechanism. They are controlled by boolean variables with names like `proof-`*foo*`-enable`, and appear at the start of the customize group `proof-user-options`. They should be edited by the user through the customization mechanism, and set in the code using `customize-set-variable`.

In `proof-utils.el` there is a handy macro, `proof-deftoggle`, which constructs an interactive function for toggling boolean customize settings. We can use this to make an interactive function `proof-`*foo*`-toggle` to put on a menu or bind to a key, for example.

This general scheme is followed as far as possible, to give uniform behaviour and appearance for boolean user options, as well as interfacing properly with the `customize` mechanism.

`proof-set-value` *sym value* [Function]
Set a customize variable using `set-default` and a function.
We first call '`set-default`' to set *sym* to *value*. Then if there is a function *sym* (i.e. with the same name as the variable *sym*), it is called to take some dynamic action for the new setting.

If there is no function *sym*, we try stripping `proof-assistant-symbol` and adding `"proof-"` instead to get a function name. This extends `proof-set-value` to work with generic individual settings.

The dynamic action call only happens when values **change**: as an approximation we test whether proof-config is fully-loaded yet.

`proof-deftoggle` [Macro]
Define a function VAR-toggle for toggling a boolean customize setting VAR.
The toggle function uses `customize-set-variable` to change the variable. *othername* gives an alternative name than the default <VAR>-toggle. The name of the defined function is returned.

13.4 Global variables

Global variables are defined in '`proof.el`'. The same file defines a few utility functions and some triggers to load in the other files.

`proof-script-buffer` [Variable]
The currently active scripting buffer or nil if none.

`proof-shell-buffer` [Variable]
Process buffer where the proof assistant is run.

`proof-response-buffer` [Variable]
The response buffer.

`proof-goals-buffer` [Variable]
The goals buffer.

`proof-buffer-type` [Variable]
Symbol for the type of this buffer: `'script`, `'shell`, `'goals`, or `'response`.

`proof-included-files-list` [Variable]
List of files currently included in proof process.
This list contains files in canonical truename format (see '`file-truename`').

Whenever a new file is being processed, it gets added to this list via the `proof-shell-process-file` configuration settings. When the prover retracts a file, this list is resynchronised via the `proof-shell-retract-files-regexp` and `proof-shell-compute-new-files-list` configuration settings.

Only files which have been **fully** processed should be included here. Proof General itself will automatically add the filenames of a script buffer which has been completely read when scripting is deactivated. It will automatically remove the filename of a script buffer which is completely unread when scripting is deactivated.

NB: Currently there is no generic provision for removing files which are only partly read-in due to an error, so ideally the proof assistant should only output a processed message when a file has been successfully read.

`proof-shell-proof-completed` [Variable]
Flag indicating that a completed proof has just been observed.
If non-nil, the value counts the commands from the last command of the proof (starting from 1).

`proof-shell-error-or-interrupt-seen` [Variable]
Flag indicating that an error or interrupt has just occurred.
Set to `'error` or `'interrupt` if one was observed from the proof assistant during the last group of commands.

13.5 Proof script mode

The file '`proof-script.el`' contains the main code for proof script mode, as well as definitions of menus, key-bindings, and user-level functions.

Proof scripts have two important variables for the locked and queue regions. These variables are local to each script buffer (although we only really need one queue span in total rather than one per buffer).

`proof-locked-span` [Variable]
The locked span of the buffer.
Each script buffer has its own locked span, which may be detached from the buffer. Proof General allows buffers in other modes also to be locked; these also have a non-nil value for this variable.

`proof-queue-span` [Variable]
The queue span of the buffer. May be detached if inactive or empty.
Each script buffer has its own queue span, although only the active scripting buffer may have an active queue span.

Various utility functions manipulate and examine the spans. An important one is `proof-init-segmentation`.

`proof-init-segmentation` [Function]
Initialise the queue and locked spans in a proof script buffer.
Allocate spans if need be. The spans are detached from the buffer, so the regions are made empty by this function. Also clear list of script portions.

For locking files loaded by a proof assistant, we use the next function.

`proof-complete-buffer-atomic` *buffer* [Function]
Make sure *buffer* is marked as completely processed, completing with a single step.

If buffer already contains a locked region, only the remainder of the buffer is closed off atomically.

This works for buffers which are not in proof scripting mode too, to allow other files loaded by proof assistants to be marked read-only.

Atomic locking is instigated by the next function, which uses the variables `proof-included-files-list` documented earlier (see Chapter 8 [Handling Multiple Files], page 33 and see Section 13.4 [Global variables], page 47).

`proof-register-possibly-new-processed-file` *file* **&optional** *informprover noquestions* [Function]
Register a possibly new *file* as having been processed by the prover.

If *informprover* is non-nil, the proof assistant will be told about this, to co-ordinate with its internal file-management. (Otherwise we assume that it is a message from the proof assistant which triggers this call). In this case, the user will be queried to save some buffers, unless *noquestions* is non-nil.

No action is taken if the file is already registered.

A warning message is issued if the register request came from the proof assistant and Emacs has a modified buffer visiting the file.

An important pair of functions activate and deactivate scripting for the current buffer. A change in the state of active scripting can trigger various actions, such as starting up the proof assistant, or altering `proof-included-files-list`.

`proof-activate-scripting` **&optional** *nosaves queuemode* [Command]
Ready prover and activate scripting for the current script buffer.

The current buffer is prepared for scripting. No changes are necessary if it is already in Scripting minor mode. Otherwise, it will become the new active scripting buffer, provided scripting can be switched off in the previous active scripting buffer with '`proof-deactivate-scripting`'.

Activating a new script buffer may be a good time to ask if the user wants to save some buffers; this is done if the user option '`proof-query-file-save-when-activating-scripting`' is set and provided the optional argument *nosaves* is non-nil.

The optional argument *queuemode* relaxes the test for a busy proof shell to allow one which has mode *queuemode*. In all other cases, a proof shell busy error is given.

Finally, the hooks '`proof-activate-scripting-hook`' are run. This can be a useful place to configure the proof assistant for scripting in a particular file, for example, loading the correct theory, or whatever. If the hooks issue commands to the proof assistant (via '`proof-shell-invisible-command`') which result in an error, the activation is considered to have failed and an error is given.

`proof-deactivate-scripting` **&optional** *forcedaction* [Command]
Deactivate scripting for the active scripting buffer.

Set '`proof-script-buffer`' to nil and turn off the modeline indicator. No action if there is no active scripting buffer.

We make sure that the active scripting buffer either has no locked region or a full locked region (everything in it has been processed). If this is not already the case, we question the

user whether to retract or assert, or automatically take the action indicated in the user option '`proof-auto-action-when-deactivating-scripting`.'

If the scripting buffer is (or has become) fully processed, and it is associated with a file, it is registered on '`proof-included-files-list`'. Conversely, if it is (or has become) empty, we make sure that it is **not** registered. This is to be certain that the included files list behaves as we might expect with respect to the active scripting buffer, in an attempt to harmonize mixed scripting and file reading in the prover.

This function either succeeds, fails because the user refused to process or retract a partly finished buffer, or gives an error message because retraction or processing failed. If this function succeeds, then '`proof-script-buffer`' is nil afterwards.

The optional argument *forcedaction* overrides the user option '`proof-auto-action-when-deactivating-scripting`' and prevents questioning the user. It is used to make a value for the '`kill-buffer-hook`' for scripting buffers, so that when a scripting buffer is killed it is always retracted.

The function `proof-segment-up-to` is the main one used for parsing the proof script buffer. There are several variants of this function available corresponding to different parsing strategies; the appropriate one is aliased to `proof-segment-up-to` according to which configuration variables have been set. If only `proof-script-command-end-regexp` or `proof-terminal-char` are set, then the default is `proof-segment-up-to-cmdend`. If `proof-script-command-start-regexp` is set, the choice is `proof-segment-up-to-cmdstart`.

`proof-segment-up-to-cmdend` *pos* **&optional** *next-command-end* [Function]

Parse the script buffer from end of locked to *pos*.
Return a list of (type, string, int) tuples.

Each tuple denotes the command and the position of its terminator, type is one of `'comment`, or `'cmd`. `'unclosed-comment` may be consed onto the start if the segment finishes with an unclosed comment.

If optional *next-command-end* is non-nil, we include the command which continues past *pos*, if any.

This version is used when '`proof-script-command-end-regexp`' is set.

`proof-segment-up-to-cmdstart` *pos* **&optional** *next-command-end* [Function]

Parse the script buffer from end of locked to *pos*.
Return a list of (type, string, int) tuples.

Each tuple denotes the command and the position of its terminator, type is one of `'comment`, or `'cmd`.

If optional *next-command-end* is non-nil, we include the command which continues past *pos*, if any. (NOT *implemented* IN *this version*).

This version is used when '`proof-script-command-start-regexp`' is set.

The function `proof-semis-to-vanillas` is used to convert a parsed region of the script into a series of commands to be sent to the proof assistant.

`proof-semis-to-vanillas` *semis* **&optional** *callback-fn* [Function]

Convert a sequence of terminator positions to a set of vanilla extents.
Proof terminator positions *semis* has the form returned by the function proof-segment-up-to.
Set the callback to *callback-fn* or `'proof-done-advancing` by default.

The function `proof-assert-until-point` is the main one used to process commands in the script buffer. It's actually used to implement the assert-until-point, electric terminator keypress, and find-next-terminator behaviours. In different cases we want different things, but usually the

information (i.e. are we inside a comment) isn't available until we've actually run `proof-segment-up-to (point)`, hence all the different options when we've done so.

`proof-assert-until-point` **&optional** *unclosed-comment-fun ignore-proof-process-p* [Function]
Process the region from the end of the locked-region until point.
Default action if inside a comment is just process as far as the start of the comment.

If you want something different, put it inside *unclosed-comment-fun*. If *ignore-proof-process-p* is set, no commands will be added to the queue and the buffer will not be activated for scripting.

`proof-assert-next-command` is a variant of this function.

`proof-assert-next-command` **&optional** *unclosed-comment-fun ignore-proof-process-p dont-move-forward for-new-command* [Command]
Process until the end of the next unprocessed command after point.
If inside a comment, just process until the start of the comment.

If you want something different, put it inside *unclosed-comment-fun*. If *ignore-proof-process-p* is set, no commands will be added to the queue. Afterwards, move forward to near the next command afterwards, unless *dont-move-forward* is non-nil. If *for-new-command* is non-nil, a space or newline will be inserted automatically.

The main command for retracting parts of a script is `proof-retract-until-point`.

`proof-retract-until-point` **&optional** *delete-region* [Function]
Set up the proof process for retracting until point.
In particular, set a flag for the filter process to call '`proof-done-retracting`' after the proof process has successfully reset its state. If *delete-region* is non-nil, delete the region in the proof script corresponding to the proof command sequence. If invoked outside a locked region, undo the last successfully processed command.

To clean up when scripting is stopped, a script buffer is killed, or the proof assistant exits, we use the functions `proof-restart-buffers` and `proof-script-remove-all-spans-and-deactivate`.

`proof-restart-buffers` *buffers* [Function]
Remove all extents in *buffers* and maybe reset '`proof-script-buffer`'.
No effect on a buffer which is nil or killed. If one of the buffers is the current scripting buffer, then '`proof-script-buffer`' will deactivated.

`proof-script-remove-all-spans-and-deactivate` [Function]
Remove all spans from scripting buffers via `proof-restart-buffers`.

13.6 Proof shell mode

The proof shell mode code is in the file '`proof-shell.el`'. Proof shell mode is defined to inherit from `comint-mode` using `define-derived-mode` near the end of the file. The bulk of the code in the file is concerned with sending code to and from the shell, and processing output for the associated buffers (goals and response).

Good process handling is a tricky issue. Proof General attempts to manage the process strictly, by maintaining a queue of commands to send to the process. Once a command has been processed, another one is popped off the queue and sent.

There are several important internal variables which control interaction with the process.

`proof-shell-busy` [Variable]
A lock indicating that the proof shell is processing.
When this is non-nil, `proof-shell-ready-prover` will give an error.

`proof-marker` [Variable]
Marker in proof shell buffer pointing to previous command input.

`proof-action-list` [Variable]
A list of

```
(span command action)
```

triples, which is a queue of things to do. See the functions '`proof-start-queue`' and '`proof-exec-loop`'.

`pg-subterm-anns-use-stack` [Variable]
Choice of syntax tree encoding for terms.
If nil, prover is expected to make no optimisations. If non-nil, the pretty printer of the prover only reports local changes. For *lego* 1.3.1 use nil, for Coq 6.2, use t.

The function `proof-shell-start` is used to initialise a shell buffer and the associated buffers.

`proof-shell-start` [Command]
Initialise a shell-like buffer for a proof assistant.
Also generates goal and response buffers. Does nothing if proof assistant is already running.

The function `proof-shell-kill-function` performs the converse function of shutting things down; it is used as a hook function for `kill-buffer-hook`. Then no harm occurs if the user kills the shell directly, or if it is done more cautiously via `proof-shell-exit`. The function `proof-shell-restart` allows a less drastic way of restarting scripting, other than killing and restarting the process.

`proof-shell-kill-function` [Function]
Function run when a proof-shell buffer is killed.
Try to shut down the proof process nicely and clear locked regions and state variables. Value for '`kill-buffer-hook`' in shell buffer, alled by '`proof-shell-bail-out`' if process exits.

`proof-shell-exit` [Command]
Query the user and exit the proof process.
This simply kills the '`proof-shell-buffer`' relying on the hook function '`proof-shell-kill-function`' to do the hard work.

`proof-shell-bail-out` *process event* [Function]
Value for the process sentinel for the proof assistant process.
If the proof assistant dies, run `proof-shell-kill-function` to cleanup and remove the associated buffers. The shell buffer is left around so the user may discover what killed the process.

`proof-shell-restart` [Command]
Clear script buffers and send '`proof-shell-restart-cmd`'.
All locked regions are cleared and the active scripting buffer deactivated.
If the proof shell is busy, an interrupt is sent with '`proof-interrupt-process`' and we wait until the process is ready.
The restart command should re-synchronize Proof General with the proof assistant, without actually exiting and restarting the proof assistant process.

It is up to the proof assistant how much context is cleared: for example, theories already loaded may be "cached" in some way, so that loading them the next time round only performs a re-linking operation, not full re-processing. (One way of caching is via object files, used by Lego and Coq).

13.6.1 Input to the shell

Input to the proof shell via the queue region is managed by the functions `proof-start-queue` and `proof-shell-exec-loop`.

`proof-start-queue` *start end alist* [Function]
Begin processing a queue of commands in *alist*.
If *start* is non-nil, *start* and *end* are buffer positions in the active scripting buffer for the queue region.

This function calls '`proof-append-alist`'.

`proof-append-alist` *alist* **&optional** *queuemode* [Function]
Chop off the vacuous prefix of the command queue *alist* and queue it.
For each '`proof-no-command`' item at the head of the list, invoke its callback and remove it from the list.

Append the result onto '`proof-action-list`', and if the proof shell isn't already busy, grab the lock with *queuemode* and start processing the queue.

If the proof shell is busy when this function is called, then *queuemode* must match the mode of the queue currently being processed.

`proof-shell-exec-loop` [Function]
Process the '`proof-action-list`'.

'`proof-action-list`' contains a list of (*span command action*) triples.

If this function is called with a non-empty `proof-action-list`, the head of the list is the previously executed command which succeeded. We execute (*action span*) on the first item, then (*action span*) on any following items which have '`proof-no-command`' as their cmd components. If a there is a next command after that, send it to the process. If the action list becomes empty, unlock the process and remove the queue region.

The return value is non-nil if the action list is now empty.

Input is actually inserted into the shell buffer and sent to the process by the low-level function `proof-shell-insert`.

`proof-shell-insert` *string action* [Function]
Insert *string* at the end of the proof shell, call '`comint-send-input`'.

First call '`proof-shell-insert-hook`'. The argument *action* may be examined by the hook to determine how to process the *string* variable. NB: the hook is not called for the empty (null) string.

Then strip *string* of carriage returns before inserting it and updating '`proof-marker`' to point to the end of the newly inserted text.

Do not use this function directly, or output will be lost. It is only used in '`proof-append-alist`' when we start processing a queue, and in '`proof-shell-exec-loop`', to process the next item.

When Proof General is processing a queue of commands, the lock is managed using a couple of utility functions. You should not need to use these directly.

`proof-grab-lock` **&optional** *queuemode* [Function]
Grab the proof shell lock, starting the proof assistant if need be.
Runs '`proof-state-change-hook`' to notify state change. Clears the '`proof-shell-error-or-interrupt-seen`' flag. If *queuemode* is supplied, set the lock to that value.

`proof-release-lock` **&optional** *err-or-int* [Function]
Release the proof shell lock, with error or interrupt flag *err-or-int*.
Clear '`proof-shell-busy`', and set '`proof-shell-error-or-interrupt-seen`' to err-or-int.

13.6.2 Output from the shell

Two main functions deal with output, `proof-shell-process-output` and `proof-shell-process-urgent-message`. In effect we consider the output to be two streams intermingled: the "urgent" messages which have "eager" annotations, as well as the ordinary ruminations from the prover.

The idea is to conceal as much irrelevant information from the user as possible; only the remaining output between prompts and after the last urgent message will be a candidate for the goal or response buffer. The internal variable `proof-shell-urgent-message-marker` tracks the last urgent message seen.

When output is grabbed from the prover process, the first action is to strip spurious carriage return characters from the end of lines, if `proof-shell-strip-crs-from-output` requires it. Then the output is stored into `proof-shell-last-output`, and its type is stored in `proof-shell-last-output-kind`. Output which is deferred or possibly discarded until the queue is empty is copied into `proof-shell-delayed-output`, with type `proof-shell-delayed-output-kind`. A record of the last prompt seen from the prover process is also kept, in `proof-shell-last-prompt`.

`proof-shell-strip-crs-from-output` [Variable]
If non-nil, remove carriage returns (^M) at the end of lines from output.
This is enabled for cygwin32 systems by default. You should turn it off if you don't need it (slight speed penalty).

`proof-shell-last-prompt` [Variable]
A raw record of the last prompt seen from the proof system.
This is the string matched by '`proof-shell-annotated-prompt-regexp`'.

`proof-shell-last-output` [Variable]
A record of the last string seen from the proof system.
This is raw string, for internal use only.

`proof-shell-last-output-kind` [Variable]
A symbol denoting the type of the last output string from the proof system.
Specifically:

```
'interrupt      An interrupt message
'error          An error message
'abort          A proof abort message
'loopback       A command sent from the PA to be inserted into the script
'response       A response message
'goals          A goals (proof state) display
'systemspecific Something specific to a particular system,
                 -- see 'proof-shell-process-output-system-specific'
```

The output corresponding to this will be in `proof-shell-last-output`.

See also ‘proof-shell-proof-completed’ for further information about the proof process output, when ends of proofs are spotted.

This variable can be used for instance specific functions which want to examine proof-shell-last-output.

proof-shell-delayed-output [Variable]
A copy of proof-shell-last-output held back for processing at end of queue.

proof-shell-delayed-output-kind [Variable]
A copy of proof-shell-last-output-lind held back for processing at end of queue.

proof-shell-process-output *cmd string* [Function]
Process shell output (resulting from *cmd*) by matching on *string*.
cmd is the first part of the ‘proof-action-list’ that lead to this output. The result of this function is a pair (*symbol newstring*).

Here is where we recognizes interrupts, abortions of proofs, errors, completions of proofs, and proof step hints (proof by pointing results). They are checked for in this order, using

‘proof-shell-interrupt-regexp’
‘proof-shell-error-regexp’
‘proof-shell-abort-goal-regexp’
‘proof-shell-proof-completed-regexp’
‘proof-shell-result-start’

All other output from the proof engine will be reported to the user in the response buffer by setting ‘proof-shell-delayed-output’ to a cons cell of ('insert . *text*) where *text* is the text string to be inserted.

Order of testing is: interrupt, abort, error, completion.

To extend this function, set ‘proof-shell-process-output-system-specific’.

The "aborted" case is intended for killing off an open proof during retraction. Typically it matches the message caused by a ‘proof-kill-goal-command’. It simply inserts the word "Aborted" into the response buffer. So it is expected to be the result of a retraction, rather than the indication that one should be made.

This function sets ‘proof-shell-last-output’ and ‘proof-shell-last-output-kind’, which see.

proof-shell-urgent-message-marker [Variable]
Marker in proof shell buffer pointing to end of last urgent message.

proof-shell-process-urgent-message *message* [Function]
Analyse urgent *message* for various cases.
Cases are: included file, retracted file, cleared response buffer, variable setting or dependency list. If none of these apply, display *message*.

message should be a string annotated with ‘proof-shell-eager-annotation-start’, ‘proof-shell-eager-annotation-end’.

The main processing point which triggers other actions is proof-shell-filter.

proof-shell-filter *str* [Function]
Filter for the proof assistant shell-process.
A function for ‘comint-output-filter-functions’.

Deal with output and issue new input from the queue.

Handle urgent messages first. As many as possible are processed, using the function '`proof-shell-process-urgent-messages`'.

Otherwise wait until an annotated prompt appears in the input. If '`proof-shell-wakeup-char`' is set, wait until we see that in the output chunk *str*. This optimizes the filter a little bit.

If a prompt is seen, run '`proof-shell-process-output`' on the output between the new prompt and the last input (position of '`proof-marker`') or the last urgent message (position of '`proof-shell-urgent-message-marker`'), whichever is later. For example, in this case:

```
PROMPT> input
output-1
urgent-message
output-2
PROMPT>
```

'`proof-marker`' is set after *input* by '`proof-shell-insert`' and '`proof-shell-urgent-message-marker`' is set after *urgent-message*. Only *output-2* will be processed. For this reason, error messages and interrupt messages should **not** be considered urgent messages.

Output is processed using the function '`proof-shell-filter-process-output`'.

The first time that a prompt is seen, '`proof-marker`' is initialised to the end of the prompt. This should correspond with initializing the process. The ordinary output before the first prompt is ignored (urgent messages, however, are always processed; hence their name).

`proof-shell-filter-process-output` *string* [Function]

Subroutine of '`proof-shell-filter`' to process output *string*.

Appropriate action is taken depending on what '`proof-shell-process-output`' returns: maybe handle an interrupt, an error, or deal with ordinary output which is a candidate for the goal or response buffer. Ordinary output is only displayed when the proof action list becomes empty, to avoid a confusing rapidly changing output.

After processing the current output, the last step undertaken by the filter is to send the next command from the queue.

13.7 Debugging

To debug Proof General, it may be helpful to set the configuration variable `proof-general-debug`.

`proof-general-debug` [User Option]

Non-nil to run Proof General in debug mode.

This changes some behaviour (e.g. markup stripping) and displays debugging messages in the response buffer. To avoid erasing messages shortly after they're printed, set '`proof-tidy-response`' to nil. This is only useful for PG developers.

The default value is `nil`.

For more information about debugging Emacs lisp, consult the Emacs Lisp Reference Manual. I recommend using the source-level debugger `edebug`.

Appendix A Plans and Ideas

This appendix contains some tentative plans and ideas for improving Proof General.

This appendix is no longer extended: instead we keep a list of Proof General projects on the web, and forthcoming plans and ideas in the '`TODO`' and '`todo`' files included in the ordinary and developers PG distributions, respectively. Once the items mentioned below are implemented, they will be removed from here.

Please send us contributions to our wish lists, or better still, an offer to implement something from them!

A.1 Proof by pointing and similar features

This is a note by David Aspinall about proof by pointing and similar features.

Proof General already supports proof by pointing, and experimental support is provided in LEGO. We would like to extend this support to other proof assistants. Unfortunately, proof by pointing requires rather heavy support from the proof assistant. There are two aspects to the support:

- term structure mark-up
- proof by pointing command generation

Term structure mark-up is useful in itself: it allows the user to explore the structure of a term using the mouse (the smallest subexpression that the mouse is over is highlighted), and easily copy subterms from the output to a proof script.

Command generation for proof by pointing is usually specific to a particular logic in use, if we hope to generate a good proof command unambiguously for any particular click. However, Proof General could easily be generalised to offer the user a context-sensitive choice of next commands to apply, which may be more useful in practice, and a worthy addition to Proof General.

Implementors of new proof assistants should be encouraged to consider supporting term-structure mark up from the start. Command generation should be something that the logic-implementor can specify in some way.

Of the supported provers, we can certainly hope for proof-by-pointing support from Coq, since the CtCoq proof-by-pointing code has been moved into the Coq kernel lately. I hope the Coq community can encourage somebody to do this.

A.2 Granularity of atomic command sequences

This is a proposal by Thomas Kleymann for generalising the way Proof General handles sequences of proof commands (see *Goal-save sequences* in the user manual), particularly to make retraction more flexible.

The blue region of a script buffer contains the initial segment of the proof script which has been processed successfully. It consists of atomic sequences of commands (ACS). Retraction is supported to the beginning of every ACS. By default, every command is an ACS. But the granularity of atomicity should be able to be adjusted.

This is essential when arbitrary retraction is not supported. Usually, after a theorem has been proved, one may only retract to the start of the goal. One needs to mark the proof of the theorem as an ACS. At present, support for goal-save sequences (see *Goal-save sequences* in the user manual), has been hard wired. No other ACS are currently supported. We propose the following to overcome this deficiency:

`proof-atomic-sequents-list`
: is a list of instructions for setting up ACSs. Each instruction is a list of the form (*end start* &optional *forget-command*). *end* is a regular expression to recognise

the last command in an ACS. *start* is a function. Its input is the last command of an ACS. Its output is a regular expression to recognise the first command of the ACS. It is evaluated once and, starting with the command matched by *end*, the output is successively matched against previously processed commands until a match occurs (or the beginning of the current buffer is reached). The region determined by (*start*,*end*) is locked as an ACS. Optionally, the ACS is annotated with the actual command to retract the ACS. This is computed by applying *forget-command* to the first and last command of the ACS.

For convenience one might also want to allow *start* to be the symbol '`t`' as a convenient short-hand for `'(lambda (str) ".")` which always matches.

A.3 Browser mode for script files and theories

This is a proposal by David Aspinall for a browser window.

A browser window should provide support for browsing script files and theories. We should be able to inspect data in varying levels of detail, perhaps using outlining mechanisms. For theories, it would be nice to query the running proof assistant. This may require support from the assistant in the form of output which has been specially marked-up with an SGML like syntax, for example.

A browser would be useful to:

- Provide impoverished proof assistants with a browser
- Extend the uniform interface of Proof General to theory browsing
- Interact closely with proof script writing

The last point is the most important. We should be able to integrate a search mechanism for proofs of similar theorems, theorems containing particular constants, etc.

Appendix B Demonstration Instantiations

This appendix contains the code for the two demonstration instantiations of Proof General, for Isabelle.

These instantiations make an almost-bare minimum of settings to get things working. To add embellishments, you should refer to the instantiations for other systems distributed with Proof General.

B.1 demoisa-easy.el

```
;; demoisa-easy.el Example Proof General instance for Isabelle
;;
;; Copyright (C) 1999 LFCS Edinburgh.
;;
;; Author: David Aspinall <David.Aspinall@ed.ac.uk>
;;
;; PG-adapting.texi,v 9.1 2008/07/12 14:00:46 da Exp
;;
;; This is an alternative version of demoisa.el which uses the
;; proof-easy-config macro to do the work of declaring derived modes,
;; etc.
;;
;; See demoisa.el and the Proof General manual for more documentation.
;;
;; To test this file you must rename it demoisa.el.
;;

(require 'proof-easy-config)            ; easy configure mechanism

(proof-easy-config
 'demoisa "Isabelle Demo"
 proof-prog-name  "isabelle"
 proof-terminal-char             ?\;
 proof-script-comment-start             "(*"
 proof-script-comment-end               "*)"
 proof-goal-command-regexp       "^Goal"
 proof-save-command-regexp       "^qed"
 proof-goal-with-hole-regexp     "qed_goal \"\\(\\(.*\\)\\)\""
 proof-save-with-hole-regexp     "qed \"\\(\\(.*\\)\\)\""
 proof-non-undoables-regexp      "undo\\|back"
 proof-goal-command              "Goal \"%s\";"
 proof-save-command              "qed \"%s\";"
 proof-kill-goal-command         "Goal \"PROP no_goal_set\";"
 proof-showproof-command         "pr()"
 proof-undo-n-times-cmd          "pg_repeat undo %s;"
 proof-auto-multiple-files       t
 proof-shell-cd-cmd              "cd \"%s\""
 proof-shell-prompt-pattern      "[ML-=#>]+>? "
 proof-shell-interrupt-regexp    "Interrupt"
 proof-shell-start-goals-regexp  "Level [0-9]"
 proof-shell-end-goals-regexp    "val it"
 proof-shell-quit-cmd            "quit();"
```

```
 proof-assistant-home-page
 "http://www.cl.cam.ac.uk/Research/HVG/Isabelle/"
 proof-shell-annotated-prompt-regexp
 "^\\(val it = () : unit\n\\)?ML>? "
 proof-shell-error-regexp
 "\\*\\*\\*\\|^.*Error:\\|^uncaught exception \\|^Exception- "
 proof-shell-init-cmd
 "fun pg_repeat f 0 = () | pg_repeat f n = (f(); pg_repeat f (n-1));"
 proof-shell-proof-completed-regexp "^No subgoals!"
 proof-shell-eager-annotation-start
 "^\\[opening \\|^###\\|^Reading")

(provide 'demoisa)
```

B.2 demoisa.el

```
;; demoisa.el Example Proof General instance for Isabelle
;;
;; Copyright (C) 1999 LFCS Edinburgh.
;;
;; Author: David Aspinall <David.Aspinall@ed.ac.uk>
;;
;; PG-adapting.texi,v 9.1 2008/07/12 14:00:46 da Exp
;;
;; ===================================================================
;;
;; See README in this directory for an introduction.
;;
;; Basic configuration is controlled by one line in `proof-site.el'.
;; It has this line in proof-assistant-table:
;;
;;     (demoisa "Isabelle Demo" "\\.ML$")
;;
;; From this it loads this file "demoisa/demoisa.el" whenever
;; a .ML file is visited, and sets the mode to `demoisa-mode'
;; (defined below).
;;
;; I've called this instance "Isabelle Demo Proof General" just to
;; avoid confusion with the real "Isabelle Proof General" in case the
;; demo gets loaded by accident.
;;
;; To make the line above take precedence over the real Isabelle mode
;; later in the table, set PROOFGENERAL_ASSISTANTS=demoisa in the
;; shell before starting Emacs  (or customize proof-assistants).
;;


(require 'proof) ; load generic parts


;; ======== User settings for Isabelle ========
;;
```

```
;; Defining variables using customize is pretty easy.
;; You should do it at least for your prover-specific user options.
;;
;; proof-site provides us with two customization groups
;; automatically:  (based on the name of the assistant)
;;
;; 'isabelledemo        -  User options for Isabelle Demo Proof General
;; 'isabelledemo-config -  Configuration of Isabelle Proof General
;;    (constants, but may be nice to tweak)
;;
;; The first group appears in the menu
;;   ProofGeneral -> Advanced -> Customize -> Isabelledemo
;; The second group appears in the menu:
;;   ProofGeneral -> Internals -> Isabelledemo config
;;

(defcustom isabelledemo-prog-name "isabelle"
  "*Name of program to run Isabelle."
  :type 'file
  :group 'isabelledemo)

(defcustom isabelledemo-web-page
  "http://www.cl.cam.ac.uk/Research/HVG/isabelle.html"
  "URL of web page for Isabelle."
  :type 'string
  :group 'isabelledemo-config)


;;
;; ======== Configuration of generic modes ========
;;

(defun demoisa-config ()
  "Configure Proof General scripting for Isabelle."
  (setq
   proof-terminal-char ?\; ; ends every command
   proof-script-comment-start "(*"
   proof-script-comment-end "*)"
   proof-goal-command-regexp    "^Goal"
   proof-save-command-regexp    "^qed"
   proof-goal-with-hole-regexp  "qed_goal \"\\(\\(.*\\)\\)\""
   proof-save-with-hole-regexp  "qed \"\\(\\(.*\\)\\)\""
   proof-non-undoables-regexp   "undo\\|back"
   proof-undo-n-times-cmd "pg_repeat undo %s;"
   proof-showproof-command "pr()"
   proof-goal-command "Goal \"%s\";"
   proof-save-command "qed \"%s\";"
   proof-kill-goal-command "Goal \"PROP no_goal_set\";"
   proof-assistant-home-page isabelledemo-web-page
   proof-auto-multiple-files    t))
```

```
(defun demoisa-shell-config ()
  "Configure Proof General shell for Isabelle."
  (setq
   proof-shell-annotated-prompt-regexp   "^\\(val it = () : unit\n\\)?ML>? "
   proof-shell-cd-cmd "cd \"%s\""
   proof-shell-prompt-pattern "[ML-=#>]+>? "
   proof-shell-interrupt-regexp         "Interrupt"
   proof-shell-error-regexp "\\*\\*\\*\\|^.*Error:\\|^uncaught exception \\|^Exception-
   proof-shell-start-goals-regexp "Level [0-9]"
   proof-shell-end-goals-regexp "val it"
   proof-shell-proof-completed-regexp   "^No subgoals!"
   proof-shell-eager-annotation-start   "^\\[opening \\|^###\\|^Reading"
   proof-shell-init-cmd  ; define a utility function, in a lib somewhere?
   "fun pg_repeat f 0 = ()
      | pg_repeat f n = (f(); pg_repeat f (n-1));"
   proof-shell-quit-cmd "quit();"))



;;
;; ======== Defining the derived modes ========
;;

;; The name of the script mode is always <proofsym>-script,
;; but the others can be whatever you like.
;;
;; The derived modes set the variables, then call the
;; <mode>-config-done function to complete configuration.

(define-derived-mode demoisa-mode proof-mode
    "Isabelle Demo script" nil
    (demoisa-config)
    (proof-config-done))

(define-derived-mode demoisa-shell-mode proof-shell-mode
   "Isabelle Demo shell" nil
   (demoisa-shell-config)
   (proof-shell-config-done))

(define-derived-mode demoisa-response-mode proof-response-mode
  "Isabelle Demo response" nil
  (proof-response-config-done))

(define-derived-mode demoisa-goals-mode proof-goals-mode
  "Isabelle Demo goals" nil
  (proof-goals-config-done))

;; The response buffer and goals buffer modes defined above are
;; trivial.  In fact, we don't need to define them at all -- they
;; would simply default to "proof-response-mode" and "pg-goals-mode".

;; A more sophisticated instantiation might set font-lock-keywords to
```

```
;; add highlighting, or some of the proof by pointing markup
;; configuration for the goals buffer.

(provide 'demoisa)
```

Function and Command Index

D

P

Variable and User Option Index

C

I

O

P

Concept Index

Table of Contents