OX

Thorsten Jolitz

June 18, 2013

Contents

1 ox.el — Generic Export Engine for Org Mode

Author: Nicolas Goaziou <n.goaziou at gmail dot com> Keywords: outlines, hypermedia, calendar, wp

2 Commentary:

This library implements a generic export engine for Org, built on its syntactical parser: Org Elements.

Besides that parser, the generic exporter is made of three distinct parts:

- The communication channel consists in a property list, which is created and updated during the process. Its use is to offer every piece of information, would it be about initial environment or contextual data, all in a single place. The exhaustive list of properties is given in "The Communication Channel" section of this file.
- The transcoder walks the parse tree, ignores or treat as plain text elements and objects according to export options, and eventually calls back-end specific functions to do the real transcoding, concatenating their return value along the way.
- The filter system is activated at the very beginning and the very end of the export process, and each time an element or an object has been converted. It is the entry point to fine-tune standard output from back-end transcoders. See "The Filter System" section for more information.

The core function is 'org-export-as'. It returns the transcoded buffer as a string.

An export back-end is defined with 'org-export-define-backend', which defines one mandatory information: his translation table. Its value is an alist whose keys are elements and objects types and values translator functions. See function's docstring for more information about translators.

Optionally, 'org-export-define-backend' can also support specific buffer keywords, OPTION keyword's items and filters. Also refer to function documentation for more information.

If the new back-end shares most properties with another one, 'org-export-define-derived-backend' can be used to simplify the process.

Any back-end can define its own variables. Among them, those customizable should belong to the 'org-export-BACKEND' group.

Tools for common tasks across back-ends are implemented in the following part of the file.

Then, a wrapper macro for asynchronous export, 'org-export-async-start', along with tools to display results. are given in the penultimate part.

Eventually, a dispatcher ('org-export-dispatch') for standard back-ends is provided in the last one.

3 Code:

```
(eval-when-compile (require 'cl))
(require 'org-element)
(require 'org-macro)
(require 'ob-exp)

(declare-function org-publish "ox-publish" (project &optional force async))
(declare-function org-publish-all "ox-publish" (&optional force async))
(declare-function
 org-publish-current-file "ox-publish" (&optional force async))
(declare-function org-publish-current-project "ox-publish"
                  (&optional force async))

(defvar org-publish-project-alist)
(defvar org-table-number-fraction)
(defvar org-table-number-regexp)
```

4 Internal Variables

Among internal variables, the most important is 'org-export-options-alist'. This variable define the global export options, shared between every exporter, and how they are acquired.

```
(defconst org-export-max-depth 19
  "Maximum nesting depth for headlines, counting from 0.")


(defconst org-export-special-keywords '("FILETAGS" "SETUPFILE" "OPTIONS")
  "List of in-buffer keywords that require special treatment.
These keywords are not directly associated to a property.  The
way they are handled must be hard-coded into
`org-export--get-inbuffer-options' function.")


(defconst org-export-default-inline-image-rule
  `(("file" .
     ,(format "\\.%s\\'"
              (regexp-opt
               '("png" "jpeg" "jpg" "gif" "tiff" "tif" "xbm"
                  "xpm" "pbm" "pgm" "ppm") t))))
  "Default rule for link matching an inline image.
This rule applies to links with no description.  By default, it
will be considered as an inline image if it targets a local file
whose extension is either \"png\", \"jpeg\", \"jpg\", \"gif\",
\"tiff\", \"tif\", \"xbm\", \"xpm\", \"pbm\", \"pgm\" or \"ppm\".
See ‘org-export-inline-image-p’ for more information about
rules.")

(defvar org-export-async-debug nil
  "Non-nil means asynchronous export process should leave data behind.

This data is found in the appropriate \"*Org Export Process*\"
buffer, and in files prefixed with \"org-export-process\" and
located in ‘temporary-file-directory’.

When non-nil, it will also set ‘debug-on-error’ to a non-nil
value in the external process.")

(defvar org-export-stack-contents nil
  "Record asynchronously generated export results and processes.
This is an alist: its CAR is the source of the
result (destination file or buffer for a finished process,
original buffer for a running one) and its CDR is a list
containing the back-end used, as a symbol, and either a process
or the time at which it finished.  It is used to build the menu
from ‘org-export-stack’.")

(defvar org-export-registered-backends nil
  "List of backends currently available in the exporter.

A backend is stored as a list where CAR is its name, as a symbol,
and CDR is a plist with the following properties:
‘:filters-alist’, ‘:menu-entry’, ‘:options-alist’ and
‘:translate-alist’.

This variable is set with ‘org-export-define-backend’ and
‘org-export-define-derived-backend’ functions.")

(defvar org-export-dispatch-last-action nil
  "Last command called from the dispatcher.
The value should be a list.  Its CAR is the action, as a symbol,
and its CDR is a list of export options.")

(defvar org-export-dispatch-last-position (make-marker)
  "The position where the last export command was created using the dispatcher.
This marker will be used with ‘C-u C-c C-e’ to make sure export repetition
uses the same subtree if the previous command was restricted to a subtree.")
```

5 User-configurable Variables

Configuration for the masses.

They should never be accessed directly, as their value is to be stored in a property list (cf. ‘org-export-options-alist’). Back-ends will read their value from there instead.

```
(defgroup org-export nil
  "Options for exporting Org mode files."
  :tag "Org Export"
  :group 'org)

(defgroup org-export-general nil
  "General options for export engine."
  :tag "Org Export General"
  :group 'org-export)

(defcustom org-export-with-archived-trees 'headline
  "Whether sub-trees with the ARCHIVE tag should be exported.

This can have three different values:
nil         Do not export, pretend this tree is not present.
t           Do export the entire tree.
‘headline’  Only export the headline, but skip the tree below it.

This option can also be set with the OPTIONS keyword,
e.g. \"arch:nil\"."
  :group 'org-export-general
  :type '(choice
          (const :tag "Not at all" nil)
          (const :tag "Headline only" headline)
          (const :tag "Entirely" t)))

(defcustom org-export-with-author t
  "Non-nil means insert author name into the exported file.
This option can also be set with the OPTIONS keyword,
e.g. \"author:nil\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-with-clocks nil
  "Non-nil means export CLOCK keywords.
This option can also be set with the OPTIONS keyword,
e.g. \"c:t\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-with-creator 'comment
  "Non-nil means the postamble should contain a creator sentence.

The sentence can be set in `org-export-creator-string' and
defaults to \"Generated by Org mode XX in Emacs XXX.\".

If the value is `comment' insert it as a comment."
  :group 'org-export-general
  :type '(choice
          (const :tag "No creator sentence" nil)
          (const :tag "Sentence as a comment" 'comment)
          (const :tag "Insert the sentence" t)))

(defcustom org-export-with-date t
  "Non-nil means insert date in the exported document.
This option can also be set with the OPTIONS keyword,
e.g. \"date:nil\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-date-timestamp-format nil
  "Time-stamp format string to use for DATE keyword.

The format string, when specified, only applies if date consists
```

```
in a single time-stamp.  Otherwise its value will be ignored.

See ‘format-time-string’ for details on how to build this
string."
  :group 'org-export-general
  :type '(choice
          (string :tag "Time-stamp format string")
          (const :tag "No format string" nil)))

(defcustom org-export-creator-string
  (format "Emacs %s (Org mode %s)"
          emacs-version
          (if (fboundp 'org-version) (org-version) "unknown version"))
  "Information about the creator of the document.
This option can also be set on with the CREATOR keyword."
  :group 'org-export-general
  :type '(string :tag "Creator string"))

(defcustom org-export-with-drawers '(not "LOGBOOK")
  "Non-nil means export contents of standard drawers.

When t, all drawers are exported.  This may also be a list of
drawer names to export.  If that list starts with ‘not’, only
drawers with such names will be ignored.

This variable doesn't apply to properties drawers.

This option can also be set with the OPTIONS keyword,
e.g. \"d:nil\"."
  :group 'org-export-general
  :version "24.4"
  :package-version '(Org . "8.0")
  :type '(choice
          (const :tag "All drawers" t)
          (const :tag "None" nil)
          (repeat :tag "Selected drawers"
                  (string :tag "Drawer name"))
          (list :tag "Ignored drawers"
                (const :format "" not)
                (repeat :tag "Specify names of drawers to ignore during export"
                        :inline t
                        (string :tag "Drawer name")))))

(defcustom org-export-with-email nil
  "Non-nil means insert author email into the exported file.
This option can also be set with the OPTIONS keyword,
e.g. \"email:t\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-with-emphasize t
  "Non-nil means interpret *word*, /word/, _word_ and +word+.

If the export target supports emphasizing text, the word will be
typeset in bold, italic, with an underline or strike-through,
respectively.

This option can also be set with the OPTIONS keyword,
e.g. \"*:nil\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-exclude-tags '("noexport")
  "Tags that exclude a tree from export.

All trees carrying any of these tags will be excluded from
export.  This is without condition, so even subtrees inside that
carry one of the `org-export-select-tags' will be removed.

This option can also be set with the EXCLUDE_TAGS keyword."
  :group 'org-export-general
  :type '(repeat (string :tag "Tag")))

(defcustom org-export-with-fixed-width t
  "Non-nil means lines starting with \":\" will be in fixed width font.

This can be used to have pre-formatted text, fragments of code
etc.  For example:
  : ;; Some Lisp examples
  : (while (defc cnt)
  :   (ding))
will be looking just like this in also HTML.  See also the QUOTE
keyword.  Not all export backends support this.

This option can also be set with the OPTIONS keyword,
e.g. \"::nil\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-with-footnotes t
  "Non-nil means Org footnotes should be exported.
This option can also be set with the OPTIONS keyword,
e.g. \"f:nil\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-with-latex t
  "Non-nil means process LaTeX environments and fragments.

This option can also be set with the OPTIONS line,
e.g. \"tex:verbatim\".  Allowed values are:

nil         Ignore math snippets.
`verbatim'  Keep everything in verbatim.
t           Allow export of math snippets."
  :group 'org-export-general
  :version "24.4"
  :package-version '(Org . "8.0")
  :type '(choice
          (const :tag "Do not process math in any way" nil)
          (const :tag "Interpret math snippets" t)
          (const :tag "Leave math verbatim" verbatim)))

(defcustom org-export-headline-levels 3
  "The last level which is still exported as a headline.

Inferior levels will usually produce itemize or enumerate lists
when exported, but back-end behaviour may differ.

This option can also be set with the OPTIONS keyword,
e.g. \"H:2\"."
  :group 'org-export-general
  :type 'integer)

(defcustom org-export-default-language "en"
  "The default language for export and clocktable translations, as a string.
This may have an association in
`org-clock-clocktable-language-setup'.  This option can also be
set with the LANGUAGE keyword."
  :group 'org-export-general
  :type '(string :tag "Language"))

(defcustom org-export-preserve-breaks nil
  "Non-nil means preserve all line breaks when exporting.
This option can also be set with the OPTIONS keyword,
e.g. \"\\n:t\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-with-entities t
  "Non-nil means interpret entities when exporting.

For example, HTML export converts \\alpha to &alpha; and \\AA to
&Aring;.

For a list of supported names, see the constant `org-entities'
and the user option `org-entities-user'.

This option can also be set with the OPTIONS keyword,
e.g. \"e:nil\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-with-inlinetasks t
  "Non-nil means inlinetasks should be exported.
This option can also be set with the OPTIONS keyword,
e.g. \"inline:nil\"."
  :group 'org-export-general
  :version "24.4"
  :package-version '(Org . "8.0")
  :type 'boolean)

(defcustom org-export-with-planning nil
  "Non-nil means include planning info in export.

Planning info is the line containing either SCHEDULED:,
DEADLINE:, CLOSED: time-stamps, or a combination of them.

This option can also be set with the OPTIONS keyword,
e.g. \"p:t\"."
  :group 'org-export-general
  :version "24.4"
  :package-version '(Org . "8.0")
  :type 'boolean)

(defcustom org-export-with-priority nil
  "Non-nil means include priority cookies in export.
This option can also be set with the OPTIONS keyword,
e.g. \"pri:t\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-with-section-numbers t
  "Non-nil means add section numbers to headlines when exporting.

When set to an integer n, numbering will only happen for
headlines whose relative level is higher or equal to n.

This option can also be set with the OPTIONS keyword,
e.g. \"num:t\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-select-tags '("export")
  "Tags that select a tree for export.

If any such tag is found in a buffer, all trees that do not carry
one of these tags will be ignored during export.  Inside trees
that are selected like this, you can still deselect a subtree by
tagging it with one of the `org-export-exclude-tags'.
```

```
This option can also be set with the SELECT_TAGS keyword."
  :group 'org-export-general
  :type '(repeat (string :tag "Tag")))

(defcustom org-export-with-smart-quotes nil
  "Non-nil means activate smart quotes during export.
This option can also be set with the OPTIONS keyword,
e.g., \"':t\".

When setting this to non-nil, you need to take care of
using the correct Babel package when exporting to LaTeX.
E.g., you can load Babel for french like this:

#+LATEX_HEADER: \\usepackage[french]{babel}"
  :group 'org-export-general
  :version "24.4"
  :package-version '(Org . "8.0")
  :type 'boolean)

(defcustom org-export-with-special-strings t
  "Non-nil means interpret \"\\-\", \"--\" and \"---\" for export.

When this option is turned on, these strings will be exported as:

   Org     HTML     LaTeX    UTF-8
  -----+----------+--------+-------
   \\-    &shy;      \\-
   --    &ndash;    --        –
   ---   &mdash;    ---       —
   ...   &hellip;   \\ldots     …

This option can also be set with the OPTIONS keyword,
e.g. \"-:nil\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-with-statistics-cookies t
  "Non-nil means include statistics cookies in export.
This option can also be set with the OPTIONS keyword,
e.g. \"stat:nil\""
  :group 'org-export-general
  :version "24.4"
  :package-version '(Org . "8.0")
  :type 'boolean)

(defcustom org-export-with-sub-superscripts t
  "Non-nil means interpret \"_\" and \"^\" for export.

When this option is turned on, you can use TeX-like syntax for
sub- and superscripts.  Several characters after \"_\" or \"^\"
will be considered as a single item - so grouping with {} is
normally not needed.  For example, the following things will be
parsed as single sub- or superscripts.

 10^24   or   10^tau     several digits will be considered 1 item.
 10^-12  or   10^-tau    a leading sign with digits or a word
 x^2-y^3                 will be read as x^2 - y^3, because items are
                         terminated by almost any nonword/nondigit char.
 x_{i^2} or   x^(2-i)    braces or parenthesis do grouping.

Still, ambiguity is possible - so when in doubt use {} to enclose
the sub/superscript.  If you set this variable to the symbol
`{}', the braces are *required* in order to trigger
interpretations as sub/superscript.  This can be helpful in
documents that need \"_\" frequently in plain text.

This option can also be set with the OPTIONS keyword,
e.g. \"^:nil\"."
  :group 'org-export-general
  :type '(choice
          (const :tag "Interpret them" t)
          (const :tag "Curly brackets only" {})
          (const :tag "Do not interpret them" nil)))

(defcustom org-export-with-toc t
  "Non-nil means create a table of contents in exported files.

The TOC contains headlines with levels up
to`org-export-headline-levels'.  When an integer, include levels
```

```
up to N in the toc, this may then be different from
‘org-export-headline-levels’, but it will not be allowed to be
larger than the number of headline levels.  When nil, no table of
contents is made.

This option can also be set with the OPTIONS keyword,
e.g. \"toc:nil\" or \"toc:3\"."
  :group 'org-export-general
  :type '(choice
          (const :tag "No Table of Contents" nil)
          (const :tag "Full Table of Contents" t)
          (integer :tag "TOC to level")))

(defcustom org-export-with-tables t
  "If non-nil, lines starting with \"|\" define a table.
For example:

  | Name        | Address  | Birthday  |
  |-------------+----------+-----------|
  | Arthur Dent | England  | 29.2.2100 |

This option can also be set with the OPTIONS keyword,
e.g. \"|:nil\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-with-tags t
  "If nil, do not export tags, just remove them from headlines.

If this is the symbol ‘not-in-toc’, tags will be removed from
table of contents entries, but still be shown in the headlines of
the document.

This option can also be set with the OPTIONS keyword,
e.g. \"tags:nil\"."
  :group 'org-export-general
  :type '(choice
          (const :tag "Off" nil)
          (const :tag "Not in TOC" not-in-toc)
          (const :tag "On" t)))
```

```
(defcustom org-export-with-tasks t
  "Non-nil means include TODO items for export.

This may have the following values:
t                    include tasks independent of state.
‘todo’               include only tasks that are not yet done.
‘done’               include only tasks that are already done.
nil                  ignore all tasks.
list of keywords     include tasks with these keywords.

This option can also be set with the OPTIONS keyword,
e.g. \"tasks:nil\"."
  :group 'org-export-general
  :type '(choice
          (const :tag "All tasks" t)
          (const :tag "No tasks" nil)
          (const :tag "Not-done tasks" todo)
          (const :tag "Only done tasks" done)
          (repeat :tag "Specific TODO keywords"
                  (string :tag "Keyword"))))

(defcustom org-export-time-stamp-file t
  "Non-nil means insert a time stamp into the exported file.
The time stamp shows when the file was created. This option can
also be set with the OPTIONS keyword, e.g. \"timestamp:nil\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-with-timestamps t
  "Non nil means allow timestamps in export.

It can be set to any of the following values:
  t          export all timestamps.
  ‘active’   export active timestamps only.
  ‘inactive’ export inactive timestamps only.
  nil        do not export timestamps

This only applies to timestamps isolated in a paragraph
containing only timestamps.  Other timestamps are always
```

```
exported.

This option can also be set with the OPTIONS keyword, e.g.
\"<:nil\"."
  :group 'org-export-general
  :type '(choice
          (const :tag "All timestamps" t)
          (const :tag "Only active timestamps" active)
          (const :tag "Only inactive timestamps" inactive)
          (const :tag "No timestamp" nil)))

(defcustom org-export-with-todo-keywords t
  "Non-nil means include TODO keywords in export.
When nil, remove all these keywords from the export.  This option
can also be set with the OPTIONS keyword, e.g.  \"todo:nil\"."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-allow-bind-keywords nil
  "Non-nil means BIND keywords can define local variable values.
This is a potential security risk, which is why the default value
is nil.  You can also allow them through local buffer variables."
  :group 'org-export-general
  :version "24.4"
  :package-version '(Org . "8.0")
  :type 'boolean)

(defcustom org-export-snippet-translation-alist nil
  "Alist between export snippets back-ends and exporter back-ends.

This variable allows to provide shortcuts for export snippets.

For example, with a value of '\(\(\"h\" . \"html\"\)\), the
HTML back-end will recognize the contents of \"@@h:<b>@@\" as
HTML code while every other back-end will ignore it."
  :group 'org-export-general
  :version "24.4"
  :package-version '(Org . "8.0")
  :type '(repeat
          (cons (string :tag "Shortcut")
                (string :tag "Back-end"))))

(defcustom org-export-coding-system nil
  "Coding system for the exported file."
  :group 'org-export-general
  :version "24.4"
  :package-version '(Org . "8.0")
  :type 'coding-system)

(defcustom org-export-copy-to-kill-ring 'if-interactive
  "Should we push exported content to the kill ring?"
  :group 'org-export-general
  :version "24.3"
  :type '(choice
          (const :tag "Always" t)
          (const :tag "When export is done interactively" if-interactive)
          (const :tag "Never" nil)))

(defcustom org-export-initial-scope 'buffer
  "The initial scope when exporting with `org-export-dispatch'.
This variable can be either set to `buffer' or `subtree'."
  :group 'org-export-general
  :type '(choice
          (const :tag "Export current buffer" buffer)
          (const :tag "Export current subtree" subtree)))

(defcustom org-export-show-temporary-export-buffer t
  "Non-nil means show buffer after exporting to temp buffer.
When Org exports to a file, the buffer visiting that file is ever
shown, but remains buried.  However, when exporting to
a temporary buffer, that buffer is popped up in a second window.
When this variable is nil, the buffer remains buried also in
these cases."
  :group 'org-export-general
  :type 'boolean)

(defcustom org-export-in-background nil
  "Non-nil means export and publishing commands will run in background.
Results from an asynchronous export are never displayed
automatically.  But you can retrieve them with \\[org-export-stack]."
```

```
  :group 'org-export-general
  :version "24.4"
  :package-version '(Org . "8.0")
  :type 'boolean)

(defcustom org-export-async-init-file user-init-file
  "File used to initialize external export process.
Value must be an absolute file name.  It defaults to user's
initialization file.  Though, a specific configuration makes the
process faster and the export more portable."
  :group 'org-export-general
  :version "24.4"
  :package-version '(Org . "8.0")
  :type '(file :must-match t))

(defcustom org-export-invisible-backends nil
  "List of back-ends that shouldn't appear in the dispatcher.

Any back-end belonging to this list or derived from a back-end
belonging to it will not appear in the dispatcher menu.

Indeed, Org may require some export back-ends without notice.  If
these modules are never to be used interactively, adding them
here will avoid cluttering the dispatcher menu."
  :group 'org-export-general
  :version "24.4"
  :package-version '(Org . "8.0")
  :type '(repeat (symbol :tag "Back-End")))

(defcustom org-export-dispatch-use-expert-ui nil
  "Non-nil means using a non-intrusive `org-export-dispatch'.
In that case, no help buffer is displayed.  Though, an indicator
for current export scope is added to the prompt (\"b\" when
output is restricted to body only, \"s\" when it is restricted to
the current subtree, \"v\" when only visible elements are
considered for export, \"f\" when publishing functions should be
passed the FORCE argument and \"a\" when the export should be
asynchronous).  Also, \[?] allows to switch back to standard
mode."
  :group 'org-export-general
  :version "24.4"
  :package-version '(Org . "8.0")
  :type 'boolean)
```

6 Defining Back-ends

'org-export-define-backend' is the standard way to define an export back-end. It allows to specify translators, filters, buffer options and a menu entry. If the new back-end shares translators with another back-end, 'org-export-define-derived-backend' may be used instead.

Internally, a back-end is stored as a list, of which CAR is the name of the back-end, as a symbol, and CDR a plist. Accessors to properties of a given back-end are: 'org-export-backend-filters', 'org-export-backend-menu', 'org-export-backend-options' and 'org-export-backend-translate-table'.

Eventually 'org-export-barf-if-invalid-backend' returns an error when a given back-end hasn't been registered yet.

```
(defun org-export-define-backend (backend translators &rest body)
  "Define a new back-end BACKEND.

TRANSLATORS is an alist between object or element types and
functions handling them.

These functions should return a string without any trailing
space, or nil.  They must accept three arguments: the object or
element itself, its contents or nil when it isn't recursive and
the property list used as a communication channel.

Contents, when not nil, are stripped from any global indentation
\(although the relative one is preserved).  They also always end
with a single newline character.

If, for a given type, no function is found, that element or
object type will simply be ignored, along with any blank line or
white space at its end.  The same will happen if the function
returns the nil value.  If that function returns the empty
string, the type will be ignored, but the blank lines or white
spaces will be kept.

In addition to element and object types, one function can be
```

```
associated to the ‘template’ (or ‘inner-template’) symbol and
another one to the ‘plain-text’ symbol.

The former returns the final transcoded string, and can be used
to add a preamble and a postamble to document’s body.  It must
accept two arguments: the transcoded string and the property list
containing export options.  A function associated to ‘template’
will not be applied if export has option \"body-only\".
A function associated to ‘inner-template’ is always applied.

The latter, when defined, is to be called on every text not
recognized as an element or an object.  It must accept two
arguments: the text string and the information channel.  It is an
appropriate place to protect special chars relative to the
back-end.

BODY can start with pre-defined keyword arguments.  The following
keywords are understood:

  :export-block

    String, or list of strings, representing block names that
    will not be parsed.  This is used to specify blocks that will
    contain raw code specific to the back-end.  These blocks
    still have to be handled by the relative ‘export-block’ type
    translator.

  :filters-alist

    Alist between filters and function, or list of functions,
    specific to the back-end.  See ‘org-export-filters-alist’ for
    a list of all allowed filters.  Filters defined here
    shouldn’t make a back-end test, as it may prevent back-ends
    derived from this one to behave properly.

  :menu-entry

    Menu entry for the export dispatcher.  It should be a list
    like:
```

```
  '(KEY DESCRIPTION-OR-ORDINAL ACTION-OR-MENU)

where :

  KEY is a free character selecting the back-end.

  DESCRIPTION-OR-ORDINAL is either a string or a number.

  If it is a string, is will be used to name the back-end in
  its menu entry.  If it is a number, the following menu will
  be displayed as a sub-menu of the back-end with the same
  KEY.  Also, the number will be used to determine in which
  order such sub-menus will appear (lowest first).

  ACTION-OR-MENU is either a function or an alist.

  If it is an action, it will be called with four
  arguments (booleans): ASYNC, SUBTREEP, VISIBLE-ONLY and
  BODY-ONLY.  See `org-export-as' for further explanations on
  some of them.

  If it is an alist, associations should follow the
  pattern:

    '(KEY DESCRIPTION ACTION)

  where KEY, DESCRIPTION and ACTION are described above.

Valid values include:

  '(?m \"My Special Back-end\" my-special-export-function)

  or

  '(?l \"Export to LaTeX\"
       \(?p \"As PDF file\" org-latex-export-to-pdf)
       \(?o \"As PDF file and open\"
           \(lambda (a s v b)
             \(if a (org-latex-export-to-pdf t s v b)
               \(org-open-file
                 \(org-latex-export-to-pdf nil s v b)))))))

      or the following, which will be added to the previous
      sub-menu,

      '(?l 1
          \((?B \"As TEX buffer (Beamer)\" org-beamer-export-as-latex)
           \(?P \"As PDF file (Beamer)\" org-beamer-export-to-pdf)))

  :options-alist

    Alist between back-end specific properties introduced in
    communication channel and how their value are acquired.  See
    `org-export-options-alist' for more information about
    structure of the values."

(defun org-export-define-derived-backend (child parent &rest body)
  "Create a new back-end as a variant of an existing one.

CHILD is the name of the derived back-end.  PARENT is the name of
the parent back-end.

BODY can start with pre-defined keyword arguments.  The following
keywords are understood:

  :export-block

    String, or list of strings, representing block names that
    will not be parsed.  This is used to specify blocks that will
    contain raw code specific to the back-end.  These blocks
    still have to be handled by the relative `export-block' type
    translator.

  :filters-alist

    Alist of filters that will overwrite or complete filters
    defined in PARENT back-end.  See `org-export-filters-alist'
    for a list of allowed filters.

  :menu-entry
```

```
    Menu entry for the export dispatcher.  See
    ‘org-export-define-backend’ for more information about the
    expected value.

  :options-alist

    Alist of back-end specific properties that will overwrite or
    complete those defined in PARENT back-end.  Refer to
    ‘org-export-options-alist’ for more information about
    structure of the values.

  :translate-alist

    Alist of element and object types and transcoders that will
    overwrite or complete transcode table from PARENT back-end.
    Refer to ‘org-export-define-backend’ for detailed information
    about transcoders.

As an example, here is how one could define \"my-latex\" back-end
as a variant of ‘latex’ back-end with a custom template function:

  \(org-export-define-derived-backend ’my-latex ’latex
     :translate-alist ’((template . my-latex-template-fun)))

The back-end could then be called with, for example:

  \(org-export-to-buffer ’my-latex \"*Test my-latex*\")"

(defun org-export-backend-parent (backend)
  "Return back-end from which BACKEND is derived, or nil."


(defun org-export-backend-filters (backend)
  "Return filters for BACKEND."


(defun org-export-backend-menu (backend)
  "Return menu entry for BACKEND."
```

```
(defun org-export-backend-options (backend)
  "Return export options for BACKEND."
```

```
(defun org-export-backend-translate-table (backend)
  "Return translate table for BACKEND."
```

```
(defun org-export-barf-if-invalid-backend (backend)
  "Signal an error if BACKEND isn't defined."
```

```
(defun org-export-derived-backend-p (backend &rest backends)
  "Non-nil if BACKEND is derived from one of BACKENDS."
```

7 The Communication Channel

During export process, every function has access to a number of properties. They are of two types:

1. Environment options are collected once at the very beginning of the process, out of the original buffer and configuration. Collecting them is handled by 'org-export-get-environment' function.

   Most environment options are defined through the 'org-export-options-alist' variable.

2. Tree properties are extracted directly from the parsed tree, just before export, by 'org-export-collect-tree-properties'.

Here is the full list of properties available during transcode process, with their category and their value type.

':author' Author's name.

> **category** option
>
> **type** string

':back-end' Current back-end used for transcoding.

category tree

type symbol

‘:creator’ String to write as creation information.

category option

type string

‘:date’ String to use as date.

category option

type string

‘:description’ Description text for the current data.

category option

type string

‘:email’ Author’s email.

category option

type string

‘:exclude-tags’ Tags for exclusion of subtrees from export process.

category option

type list of strings

‘:export-options’ List of export options available for current process.

category none

type list of symbols, among ‘subtree’, ‘body-only’ and ‘visible-only’.

‘:exported-data’ Hash table used for memoizing ‘org-export-data’.

category tree

type hash table

‘:filetags’ List of global tags for buffer. Used by ‘org-export-get-tags’ to get tags with inheritance.

category option

type list of strings

‘:footnote-definition-alist’ Alist between footnote labels and their definition, as parsed data. Only non-inlined footnotes are represented in this alist. Also, every definition isn’t guaranteed to be referenced in the parse tree. The purpose of this property is to preserve definitions from oblivion (i.e. when the parse tree comes from a part of the original buffer), it isn’t meant for direct use in a back-end. To retrieve a definition relative to a reference, use ‘org-export-get-footnote-definition’ instead.

category option

type alist (STRING . LIST)

‘:headline-levels’ Maximum level being exported as an headline. Comparison is done with the relative level of headlines in the parse tree, not necessarily with their actual level.

category option

type integer

‘:headline-offset’ Difference between relative and real level of headlines in the parse tree. For example, a value of -1 means a level 2 headline should be considered as level 1 (cf. ‘org-export-get-relative-level’).

category tree

type integer

‘:headline-numbering’ Alist between headlines and their numbering, as a list of numbers (cf. ‘org-export-get-headline-number’).

category tree

type alist (INTEGER . LIST)

‘:id-alist’ Alist between ID strings and destination file’s path, relative to current directory. It is used by ‘org-export-resolve-id-link’ to resolve ID links targeting an external file.

category option

type alist (STRING . STRING)

‘:ignore-list’ List of elements and objects that should be ignored during export.

category tree

type list of elements and objects

':input-file' Full path to input file, if any.

category option

type string or nil

':keywords' List of keywords attached to data.

category option

type string

':language' Default language used for translations.

category option

type string

':parse-tree' Whole parse tree, available at any time during transcoding.

category option

type list (as returned by 'org-element-parse-buffer')

':preserve-breaks' Non-nil means transcoding should preserve all line breaks.

category option

type symbol (nil, t)

':section-numbers' Non-nil means transcoding should add section numbers to headlines.

category option

type symbol (nil, t)

':select-tags' List of tags enforcing inclusion of sub-trees in transcoding. When such a tag is present, subtrees without it are de facto excluded from the process. See 'use-select-tags'.

category option

type list of strings

':time-stamp-file' Non-nil means transcoding should insert a time stamp in the output.

> **category** option
>
> **type** symbol (nil, t)

':translate-alist' Alist between element and object types and transcoding functions relative to the current back-end. Special keys 'inner-template', 'template' and 'plain-text' are also possible.

> **category** option
>
> **type** alist (SYMBOL . FUNCTION)

':with-archived-trees' Non-nil when archived subtrees should also be transcoded. If it is set to the 'headline' symbol, only the archived headline's name is retained.

> **category** option
>
> **type** symbol (nil, t, 'headline')

':with-author' Non-nil means author's name should be included in the output.

> **category** option
>
> **type** symbol (nil, t)

':with-clocks' Non-nil means clock keywords should be exported.

> **category** option
>
> **type** symbol (nil, t)

':with-creator' Non-nil means a creation sentence should be inserted at the end of the transcoded string. If the value is 'comment', it should be commented.

> **category** option
>
> **type** symbol ('comment', nil, t)

':with-date' Non-nil means output should contain a date.

> **category** option
>
> type :. symbol (nil, t)

':with-drawers' Non-nil means drawers should be exported. If its value is a list of names, only drawers with such names will be transcoded. If that list starts with 'not', drawer with these names will be skipped.

category option

type symbol (nil, t) or list of strings

':with-email' Non-nil means output should contain author's email.

category option

type symbol (nil, t)

':with-emphasize' Non-nil means emphasized text should be interpreted.

category option

type symbol (nil, t)

':with-fixed-width' Non-nil if transcoder should interpret strings starting with a colon as a fixed-with (verbatim) area.

category option

type symbol (nil, t)

':with-footnotes' Non-nil if transcoder should interpret footnotes.

category option

type symbol (nil, t)

':with-latex' Non-nil means 'latex-environment' elements and 'latex-fragment' objects should appear in export output. When this property is set to 'verbatim', they will be left as-is.

category option

type symbol ('verbatim', nil, t)

':with-planning' Non-nil means transcoding should include planning info.

category option

type symbol (nil, t)

':with-priority' Non-nil means transcoding should include priority cookies.

category option

type symbol (nil, t)

':with-smart-quotes' Non-nil means activate smart quotes in plain text.

category option

type symbol (nil, t)

':with-special-strings' Non-nil means transcoding should interpret special strings in plain text.

category option

type symbol (nil, t)

':with-sub-superscript' Non-nil means transcoding should interpret subscript and superscript. With a value of "{}", only interpret those using curly brackets.

category option

type symbol (nil, {}, t)

':with-tables' Non-nil means transcoding should interpret tables.

category option

type symbol (nil, t)

':with-tags' Non-nil means transcoding should keep tags in headlines. A 'not-in-toc' value will remove them from the table of contents, if any, nonetheless.

category option

type symbol (nil, t, 'not-in-toc')

':with-tasks' Non-nil means transcoding should include headlines with a TODO keyword. A 'todo' value will only include headlines with a todo type keyword while a 'done' value will do the contrary. If a list of strings is provided, only tasks with keywords belonging to that list will be kept.

category option

type symbol (t, todo, done, nil) or list of strings

':with-timestamps' Non-nil means transcoding should include time stamps. Special value 'active' (resp. 'inactive') ask to export only active (resp. inactive) timestamps. Otherwise, completely remove them.

category option

type symbol: (‘active’, ‘inactive’, t, nil)

‘:with-toc’ Non-nil means that a table of contents has to be added to the output. An integer value limits its depth.

category option

type symbol (nil, t or integer)

‘:with-todo-keywords’ Non-nil means transcoding should include TODO keywords.

category option

type symbol (nil, t)

7.1 Environment Options

Environment options encompass all parameters defined outside the scope of the parsed data. They come from five sources, in increasing precedence order:

- Global variables,
- Buffer’s attributes,
- Options keyword symbols,
- Buffer keywords,
- Subtree properties.

The central internal function with regards to environment options is ‘org-export-get-environment’. It updates global variables with "#+BIND:" keywords, then retrieve and prioritize properties from the different sources.

The internal functions doing the retrieval are: ‘org-export–get-global-options’, ‘org-export–get-buffer-attributes’, ‘org-export–parse-option-keyword’, ‘org-export–get-subtree-options’ and ‘org-export–get-inbuffer-options’

Also, ‘org-export–install-letbind-maybe’ takes care of the part relative to "#+BIND:" keywords.

```
(defun org-export-get-environment (&optional backend subtreep ext-plist)
  "Collect export options from the current buffer.
```

```
Optional argument BACKEND is a symbol specifying which back-end
specific options to read, if any.

When optional argument SUBTREEP is non-nil, assume the export is
done against the current sub-tree.

Third optional argument EXT-PLIST is a property list with
external parameters overriding Org default settings, but still
inferior to file-local settings."

(defun org-export--parse-option-keyword (options &optional backend)
  "Parse an OPTIONS line and return values as a plist.
Optional argument BACKEND is a symbol specifying which back-end
specific items to read, if any."
      plist))

(defun org-export--get-subtree-options (&optional backend)
  "Get export options in subtree at point.
Optional argument BACKEND is a symbol specifying back-end used
for export.  Return options as a plist."


(defun org-export--get-inbuffer-options (&optional backend)
  "Return current buffer export options, as a plist.

Optional argument BACKEND, when non-nil, is a symbol specifying
which back-end specific options should also be read in the
process.

Assume buffer is in Org mode.  Narrowing, if any, is ignored."


(defun org-export--get-buffer-attributes ()
  "Return properties related to buffer attributes, as a plist."


(defvar org-export--default-title nil)  ; Dynamically scoped.
(defun org-export-store-default-title ()
  "Return default title for current document, as a string.
Title is extracted from associated file name, if any, or buffer's
```

```
name."
```

```
(defun org-export--get-global-options (&optional backend)
  "Return global export options as a plist.
Optional argument BACKEND, if non-nil, is a symbol specifying
which back-end specific export options should also be read in the
process."
```

```
(defun org-export--list-bound-variables ()
  "Return variables bound from BIND keywords in current buffer.
Also look for BIND keywords in setup files.  The return value is
an alist where associations are (VARIABLE-NAME VALUE)."
```

7.2 Tree Properties

Tree properties are information extracted from parse tree. They are initialized at the beginning of the transcoding process by 'org-export-collect-tree-properties'.

Dedicated functions focus on computing the value of specific tree properties during initialization. Thus, 'org-export–populate-ignore-list' lists elements and objects that should be skipped during export, 'org-export–get-min-level' gets the minimal exportable level, used as a basis to compute relative level for headlines. Eventually 'org-export–collect-headline-numbering' builds an alist between headlines and their numbering.

```
(defun org-export-collect-tree-properties (data info)
  "Extract tree properties from parse tree.

DATA is the parse tree from which information is retrieved.  INFO
is a list holding export options.

Following tree properties are set or updated:

‘:exported-data’ Hash table used to memoize results from
                 ‘org-export-data’.

‘:footnote-definition-alist’ List of footnotes definitions in
```

```
                   original buffer and current parse tree.

‘:headline-offset’ Offset between true level of headlines and
                   local level.  An offset of -1 means a headline
                   of level 2 should be considered as a level
                   1 headline in the context.

‘:headline-numbering’ Alist of all headlines as key an the
                      associated numbering as value.

‘:ignore-list’     List of elements that should be ignored during
                   export.

Return updated plist."

(defun org-export--get-min-level (data options)
  "Return minimum exportable headline’s level in DATA.
DATA is parsed tree as returned by ‘org-element-parse-buffer’.
OPTIONS is a plist holding export options."


(defun org-export--collect-headline-numbering (data options)
  "Return numbering of all exportable headlines in a parse tree.

DATA is the parse tree.  OPTIONS is the plist holding export
options.

Return an alist whose key is a headline and value is its
associated numbering \(in the shape of a list of numbers\) or nil
for a footnotes section."


(defun org-export--populate-ignore-list (data options)
  "Return list of elements and objects to ignore during export.
DATA is the parse tree to traverse.  OPTIONS is the plist holding
export options."


(defun org-export--selected-trees (data info)
  "Return list of headlines and inlinetasks with a select tag in their tree.
```

```
DATA is parsed data as returned by ‘org-element-parse-buffer’.
INFO is a plist holding export options."
```

```
(defun org-export--skip-p (blob options selected)
  "Non-nil when element or object BLOB should be skipped during export.
OPTIONS is the plist holding export options.  SELECTED, when
non-nil, is a list of headlines or inlinetasks belonging to
a tree with a select tag."
```

8 The Transcoder

‘org-export-data’ reads a parse tree (obtained with, i.e. ‘org-element-parse-buffer’) and transcodes it into a specified back-end output. It takes care of filtering out elements or objects according to export options and organizing the output blank lines and white space are preserved. The function memoizes its results, so it is cheap to call it within translators.

It is possible to modify locally the back-end used by ‘org-export-data’ or even use a temporary back-end by using ‘org-export-data-with-translations’ and ‘org-export-data-with-backend’.

Internally, three functions handle the filtering of objects and elements during the export. In particular, ‘org-export-ignore-element’ marks an element or object so future parse tree traversals skip it, ‘org-export–interpret-p’ tells which elements or objects should be seen as real Org syntax and ‘org-export-expand’ transforms the others back into their original shape

‘org-export-transcoder’ is an accessor returning appropriate translator function for a given element or object.

```
(defun org-export-transcoder (blob info)
  "Return appropriate transcoder for BLOB.
INFO is a plist containing export directives."
```

```
(defun org-export-data (data info)
  "Convert DATA into current back-end format.

DATA is a parse tree, an element or an object or a secondary
string.  INFO is a plist holding export options.
```

```
Return transcoded string."


(defun org-export-data-with-translations (data translations info)
  "Convert DATA into another format using a given translation table.
DATA is an element, an object, a secondary string or a string.
TRANSLATIONS is an alist between element or object types and
a functions handling them.  See ‘org-export-define-backend’ for
more information.  INFO is a plist used as a communication
channel."


(defun org-export-data-with-backend (data backend info)
  "Convert DATA into BACKEND format.

DATA is an element, an object, a secondary string or a string.
BACKEND is a symbol.  INFO is a plist used as a communication
channel.

Unlike to ‘org-export-with-backend’, this function will
recursively convert DATA using BACKEND translation table."
    (org-export-data-with-translations
   data (org-export-backend-translate-table backend) info))

(defun org-export--interpret-p (blob info)
  "Non-nil if element or object BLOB should be interpreted during export.
If nil, BLOB will appear as raw Org syntax.  Check is done
according to export options INFO, stored as a plist."


(defun org-export-expand (blob contents &optional with-affiliated)
  "Expand a parsed element or object to its original state.

BLOB is either an element or an object.  CONTENTS is its
contents, as a string or nil.

When optional argument WITH-AFFILIATED is non-nil, add affiliated
keywords before output."
```

```
(defun org-export-ignore-element (element info)
  "Add ELEMENT to ‘:ignore-list’ in INFO.

Any element in ‘:ignore-list’ will be skipped when using
‘org-element-map’.  INFO is modified by side effects."
```

9 The Filter System

Filters allow end-users to tweak easily the transcoded output. They are the functional counterpart of hooks, as every filter in a set is applied to the return value of the previous one.

Every set is back-end agnostic. Although, a filter is always called, in addition to the string it applies to, with the back-end used as argument, so it's easy for the end-user to add back-end specific filters in the set. The communication channel, as a plist, is required as the third argument.

From the developer side, filters sets can be installed in the process with the help of ‘org-export-define-backend’, which internally stores filters as an alist. Each association has a key among the following symbols and a function or a list of functions as value.

- ‘:filter-options’ applies to the property list containing export options. Unlike to other filters, functions in this list accept two arguments instead of three: the property list containing export options and the back-end. Users can set its value through ‘org-export-filter-options-functions’ variable.

- ‘:filter-parse-tree’ applies directly to the complete parsed tree. Users can set it through ‘org-export-filter-parse-tree-functions’ variable.

- ‘:filter-final-output’ applies to the final transcoded string. Users can set it with ‘org-export-filter-final-output-functions’ variable

- ‘:filter-plain-text’ applies to any string not recognized as Org syntax. ‘org-export-filter-plain-text-functions’ allows users to configure it.

- ‘:filter-TYPE’ applies on the string returned after an element or object of type TYPE has been transcoded. A user can modify ‘org-export-filter-TYPE-functions’

All filters sets are applied with ‘org-export-filter-apply-functions’ function. Filters in a set are applied in a LIFO fashion. It allows developers to be sure that their filters will be applied first.

Filters properties are installed in communication channel with ‘org-export-install-filters’ function.

Eventually, two hooks (‘org-export-before-processing-hook’ and ‘org-export-before-parsing-hook’) are run at the beginning of the export process and just before parsing to allow for heavy structure modifications.

9.1 Hooks

```
(defvar org-export-before-processing-hook nil
  "Hook run at the beginning of the export process.

This is run before include keywords and macros are expanded and
Babel code blocks executed, on a copy of the original buffer
being exported.  Visibility and narrowing are preserved.  Point
is at the beginning of the buffer.

Every function in this hook will be called with one argument: the
back-end currently used, as a symbol.")
```

9.2 Special Filters

```
(defvar org-export-filter-options-functions nil
  "List of functions applied to the export options.
Each filter is called with two arguments: the export options, as
a plist, and the back-end, as a symbol.  It must return
a property list containing export options.")

(defvar org-export-filter-parse-tree-functions nil
  "List of functions applied to the parsed tree.
Each filter is called with three arguments: the parse tree, as
returned by ‘org-element-parse-buffer’, the back-end, as
a symbol, and the communication channel, as a plist.  It must
return the modified parse tree to transcode.")

(defvar org-export-filter-plain-text-functions nil
  "List of functions applied to plain text.
Each filter is called with three arguments: a string which
contains no Org syntax, the back-end, as a symbol, and the
communication channel, as a plist.  It must return a string or
nil.")

(defvar org-export-filter-final-output-functions nil
  "List of functions applied to the transcoded string.
Each filter is called with three arguments: the full transcoded
string, the back-end, as a symbol, and the communication channel,
as a plist.  It must return a string that will be used as the
final export output.")
```

9.3 Elements Filters

```
(defvar org-export-filter-babel-call-functions nil
  "List of functions applied to a transcoded babel-call.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-center-block-functions nil
  "List of functions applied to a transcoded center block.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-clock-functions nil
  "List of functions applied to a transcoded clock.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-comment-functions nil
  "List of functions applied to a transcoded comment.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-comment-block-functions nil
  "List of functions applied to a transcoded comment-block.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-diary-sexp-functions nil
  "List of functions applied to a transcoded diary-sexp.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-drawer-functions nil
  "List of functions applied to a transcoded drawer.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-dynamic-block-functions nil
  "List of functions applied to a transcoded dynamic-block.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-example-block-functions nil
  "List of functions applied to a transcoded example-block.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-export-block-functions nil
  "List of functions applied to a transcoded export-block.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-fixed-width-functions nil
  "List of functions applied to a transcoded fixed-width.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")
```

```
(defvar org-export-filter-footnote-definition-functions nil
  "List of functions applied to a transcoded footnote-definition.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-headline-functions nil
  "List of functions applied to a transcoded headline.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-horizontal-rule-functions nil
  "List of functions applied to a transcoded horizontal-rule.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-inlinetask-functions nil
  "List of functions applied to a transcoded inlinetask.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-item-functions nil
  "List of functions applied to a transcoded item.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-keyword-functions nil
  "List of functions applied to a transcoded keyword.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-latex-environment-functions nil
  "List of functions applied to a transcoded latex-environment.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
```

```
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-node-property-functions nil
  "List of functions applied to a transcoded node-property.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-paragraph-functions nil
  "List of functions applied to a transcoded paragraph.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-plain-list-functions nil
  "List of functions applied to a transcoded plain-list.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-planning-functions nil
  "List of functions applied to a transcoded planning.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-property-drawer-functions nil
  "List of functions applied to a transcoded property-drawer.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-quote-block-functions nil
  "List of functions applied to a transcoded quote block.
Each filter is called with three arguments: the transcoded quote
data, as a string, the back-end, as a symbol, and the
communication channel, as a plist.  It must return a string or
nil.")

(defvar org-export-filter-quote-section-functions nil
  "List of functions applied to a transcoded quote-section.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-section-functions nil
  "List of functions applied to a transcoded section.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-special-block-functions nil
  "List of functions applied to a transcoded special block.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-src-block-functions nil
  "List of functions applied to a transcoded src-block.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-table-functions nil
  "List of functions applied to a transcoded table.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-table-cell-functions nil
  "List of functions applied to a transcoded table-cell.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-table-row-functions nil
  "List of functions applied to a transcoded table-row.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")
```

```
(defvar org-export-filter-verse-block-functions nil
  "List of functions applied to a transcoded verse block.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")
```

9.4 Objects Filters

```
(defvar org-export-filter-bold-functions nil
  "List of functions applied to transcoded bold text.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-code-functions nil
  "List of functions applied to transcoded code text.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-entity-functions nil
  "List of functions applied to a transcoded entity.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-export-snippet-functions nil
  "List of functions applied to a transcoded export-snippet.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-footnote-reference-functions nil
  "List of functions applied to a transcoded footnote-reference.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-inline-babel-call-functions nil
  "List of functions applied to a transcoded inline-babel-call.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-inline-src-block-functions nil
  "List of functions applied to a transcoded inline-src-block.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-italic-functions nil
  "List of functions applied to transcoded italic text.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-latex-fragment-functions nil
  "List of functions applied to a transcoded latex-fragment.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-line-break-functions nil
  "List of functions applied to a transcoded line-break.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-link-functions nil
  "List of functions applied to a transcoded link.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-radio-target-functions nil
  "List of functions applied to a transcoded radio-target.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")
```

```
(defvar org-export-filter-statistics-cookie-functions nil
  "List of functions applied to a transcoded statistics-cookie.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-strike-through-functions nil
  "List of functions applied to transcoded strike-through text.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-subscript-functions nil
  "List of functions applied to a transcoded subscript.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-superscript-functions nil
  "List of functions applied to a transcoded superscript.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-target-functions nil
  "List of functions applied to a transcoded target.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-timestamp-functions nil
  "List of functions applied to a transcoded timestamp.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-underline-functions nil
  "List of functions applied to transcoded underline text.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")

(defvar org-export-filter-verbatim-functions nil
  "List of functions applied to transcoded verbatim text.
Each filter is called with three arguments: the transcoded data,
as a string, the back-end, as a symbol, and the communication
channel, as a plist.  It must return a string or nil.")
```

9.5 Filters Tools

Internal function 'org-export-install-filters' installs filters hard-coded in back-ends (developer filters) and filters from global variables (user filters) in the communication channel.

Internal function 'org-export-filter-apply-functions' takes care about applying each filter in order to a given data. It ignores filters returning a nil value but stops whenever a filter returns an empty string.

```
(defun org-export-filter-apply-functions (filters value info)
  "Call every function in FILTERS.

Functions are called with arguments VALUE, current export
back-end and INFO.  A function returning a nil value will be
skipped.  If it returns the empty string, the process ends and
VALUE is ignored.

Call is done in a LIFO fashion, to be sure that developer
specified filters, if any, are called first."


(defun org-export-install-filters (info)
  "Install filters properties in communication channel.
INFO is a plist containing the current communication channel.
Return the updated communication channel."
```

10 Core functions

This is the room for the main function, 'org-export-as', along with its derivatives, 'org-export-to-buffer', 'org-export-to-file' and 'org-export-string-as'.

They differ either by the way they output the resulting code (for the first two) or by the input type (for the latter). 'org-export–copy-to-kill-ring-p' determines if output of these function should be added to kill ring.

'org-export-output-file-name' is an auxiliary function meant to be used with 'org-export-to-file'. With a given extension, it tries to provide a canonical file name to write export output to.

Note that 'org-export-as' doesn't really parse the current buffer, but a copy of it (with the same buffer-local variables and visibility), where macros and include keywords are expanded and Babel blocks are executed, if appropriate. 'org-export-with-buffer-copy' macro prepares that copy.

File inclusion is taken care of by 'org-export-expand-include-keyword' and 'org-export–prepare-file-contents'. Structure wise, including a whole Org file in a buffer often makes little sense. For example, if the file contains a headline and the include keyword was within an item, the item should contain the headline. That's why file inclusion should be done before any structure can be associated to the file, that is before parsing.

'org-export-insert-default-template' is a command to insert a default template (or a back-end specific template) at point or in current subtree.

```
(defun org-export-copy-buffer ()
  "Return a copy of the current buffer.
The copy preserves Org buffer-local variables, visibility and
narrowing."


(defmacro org-export-with-buffer-copy (&rest body)
  "Apply BODY in a copy of the current buffer.
The copy preserves local variables, visibility and contents of
the original buffer.  Point is at the beginning of the buffer
when BODY is applied."

(defun org-export--generate-copy-script (buffer)
  "Generate a function duplicating BUFFER.

The copy will preserve local variables, visibility, contents and
narrowing of the original buffer.  If a region was active in
BUFFER, contents will be narrowed to that region instead.

The resulting function can be evaled at a later time, from
another buffer, effectively cloning the original buffer there.
```

```
The function assumes BUFFER's major mode is 'org-mode'."
```

###autoload

```
(defun org-export-as
  (backend &optional subtreep visible-only body-only ext-plist)
  "Transcode current Org buffer into BACKEND code.

If narrowing is active in the current buffer, only transcode its
narrowed part.

If a region is active, transcode that region.

When optional argument SUBTREEP is non-nil, transcode the
sub-tree at point, extracting information from the headline
properties first.

When optional argument VISIBLE-ONLY is non-nil, don't export
contents of hidden elements.

When optional argument BODY-ONLY is non-nil, only return body
code, without surrounding template.

Optional argument EXT-PLIST, when provided, is a property list
with external parameters overriding Org default settings, but
still inferior to file-local settings.

Return code as a string."
```

###autoload

```
(defun org-export-to-buffer
  (backend buffer &optional subtreep visible-only body-only ext-plist)
  "Call 'org-export-as' with output to a specified buffer.

BACKEND is the back-end used for transcoding, as a symbol.

BUFFER is the output buffer.  If it already exists, it will be
```

```
erased first, otherwise, it will be created.

Optional arguments SUBTREEP, VISIBLE-ONLY, BODY-ONLY and
EXT-PLIST are similar to those used in ‘org-export-as’, which
see.

Depending on ‘org-export-copy-to-kill-ring’, add buffer contents
to kill ring.  Return buffer."
```

###autoload

```
(defun org-export-to-file
  (backend file &optional subtreep visible-only body-only ext-plist)
  "Call ‘org-export-as’ with output to a specified file.

BACKEND is the back-end used for transcoding, as a symbol.  FILE
is the name of the output file, as a string.

Optional arguments SUBTREEP, VISIBLE-ONLY, BODY-ONLY and
EXT-PLIST are similar to those used in ‘org-export-as’, which
see.

Depending on ‘org-export-copy-to-kill-ring’, add file contents
to kill ring.  Return output file’s name."
```

###autoload

```
(defun org-export-string-as (string backend &optional body-only ext-plist)
  "Transcode STRING into BACKEND code.

When optional argument BODY-ONLY is non-nil, only return body
code, without preamble nor postamble.

Optional argument EXT-PLIST, when provided, is a property list
with external parameters overriding Org default settings, but
still inferior to file-local settings.

Return code as a string."
```

###autoload

```
(defun org-export-replace-region-by (backend)
  "Replace the active region by its export to BACKEND."
```

###autoload

```
(defun org-export-insert-default-template (&optional backend subtreep)
  "Insert all export keywords with default values at beginning of line.

BACKEND is a symbol representing the export back-end for which
specific export options should be added to the template, or
‘default’ for default template.  When it is nil, the user will be
prompted for a category.

If SUBTREEP is non-nil, export configuration will be set up
locally for the subtree through node properties."

(defun org-export-output-file-name (extension &optional subtreep pub-dir)
  "Return output file’s name according to buffer specifications.

EXTENSION is a string representing the output file extension,
with the leading dot.

With a non-nil optional argument SUBTREEP, try to determine
output file’s name by looking for \"EXPORT_FILE_NAME\" property
of subtree at point.

When optional argument PUB-DIR is set, use it as the publishing
directory.

When optional argument VISIBLE-ONLY is non-nil, don’t export
contents of hidden elements.

Return file name as a string."


(defun org-export-expand-include-keyword (&optional included dir)
  "Expand every include keyword in buffer.
```

```
Optional argument INCLUDED is a list of included file names along
with their line restriction, when appropriate.  It is used to
avoid infinite recursion.  Optional argument DIR is the current
working directory.  It is used to properly resolve relative
paths."

(defun org-export--prepare-file-contents (file &optional lines ind minlevel)
  "Prepare the contents of FILE for inclusion and return them as a string.

When optional argument LINES is a string specifying a range of
lines, include only those lines.

Optional argument IND, when non-nil, is an integer specifying the
global indentation of returned contents.  Since its purpose is to
allow an included file to stay in the same environment it was
created \(i.e. a list item), it doesn't apply past the first
headline encountered.

Optional argument MINLEVEL, when non-nil, is an integer
specifying the level that any top-level headline in the included
file should have."


(defun org-export-execute-babel-code ()
  "Execute every Babel code in the visible part of current buffer."
  ;; Get a pristine copy of current buffer so Babel references can be
  ;; properly resolved.


(defun org-export--copy-to-kill-ring-p ()
  "Return a non-nil value when output should be added to the kill ring.
See also ‘org-export-copy-to-kill-ring’."
```

11 Tools For Back-Ends

A whole set of tools is available to help build new exporters. Any function general enough to have its use across many back-ends should be added here.

11.1 For Affiliated Keywords

'org-export-read-attribute' reads a property from a given element as a plist. It can be used to normalize affiliated keywords' syntax.

Since captions can span over multiple lines and accept dual values, their internal representation is a bit tricky. Therefore, 'org-export-get-caption' transparently returns a given element's caption as a secondary string.

```
(defun org-export-read-attribute (attribute element &optional property)
  "Turn ATTRIBUTE property from ELEMENT into a plist.

When optional argument PROPERTY is non-nil, return the value of
that property within attributes.

This function assumes attributes are defined as \":keyword
value\" pairs.  It is appropriate for ‘:attr_html’ like
properties.

All values will become strings except the empty string and
\"nil\", which will become nil.  Also, values containing only
double quotes will be read as-is, which means that \"\" value
will become the empty string."
```

```
(defun org-export-get-caption (element &optional shortp)
  "Return caption from ELEMENT as a secondary string.

When optional argument SHORTP is non-nil, return short caption,
as a secondary string, instead.

Caption lines are separated by a white space."
```

11.2 For Derived Back-ends

'org-export-with-backend' is a function allowing to locally use another back-end to transcode some object or element. In a derived back-end, it may be used as a fall-back function once all specific cases have been treated.

```
(defun org-export-with-backend (back-end data &optional contents info)
  "Call a transcoder from BACK-END on DATA.
```

```
CONTENTS, when non-nil, is the transcoded contents of DATA
element, as a string.  INFO, when non-nil, is the communication
channel used for export, as a plist.."
```

11.3 For Export Snippets

Every export snippet is transmitted to the back-end. Though, the latter will only retain one type of export-snippet, ignoring others, based on the former's target back-end. The function 'org-export-snippet-backend' returns that back-end for a given export-snippet.

```
(defun org-export-snippet-backend (export-snippet)
  "Return EXPORT-SNIPPET targeted back-end as a symbol.
Translation, with ‘org-export-snippet-translation-alist’, is
applied."
```

11.4 For Footnotes

'org-export-collect-footnote-definitions' is a tool to list actually used footnotes definitions in the whole parse tree, or in a headline, in order to add footnote listings throughout the transcoded data.

'org-export-footnote-first-reference-p' is a predicate used by some back-ends, when they need to attach the footnote definition only to the first occurrence of the corresponding label.

'org-export-get-footnote-definition' and 'org-export-get-footnote-number' provide easier access to additional information relative to a footnote reference.

```
(defun org-export-collect-footnote-definitions (data info)
  "Return an alist between footnote numbers, labels and definitions.

DATA is the parse tree from which definitions are collected.
INFO is the plist used as a communication channel.

Definitions are sorted by order of references.  They either
appear as Org data or as a secondary string for inlined
footnotes.  Unreferenced definitions are ignored."


(defun org-export-footnote-first-reference-p (footnote-reference info)
  "Non-nil when a footnote reference is the first one for its label.

FOOTNOTE-REFERENCE is the footnote reference being considered.
INFO is the plist used as a communication channel."


(defun org-export-get-footnote-definition (footnote-reference info)
  "Return definition of FOOTNOTE-REFERENCE as parsed data.
INFO is the plist used as a communication channel.  If no such
definition can be found, return the \"DEFINITION NOT FOUND\"
string."


(defun org-export-get-footnote-number (footnote info)
  "Return number associated to a footnote.

FOOTNOTE is either a footnote reference or a footnote definition.
INFO is the plist used as a communication channel."
```

11.5 For Headlines

‘org-export-get-relative-level’ is a shortcut to get headline level, relatively to the lower headline level in the parsed tree.

‘org-export-get-headline-number’ returns the section number of an headline, while ‘org-export-number-to-roman’ allows to convert it to roman numbers.

‘org-export-low-level-p’, ‘org-export-first-sibling-p’ and ‘org-export-last-sibling-p’ are three useful predicates when it comes to fulfill the ‘:headline-levels’ property.

‘org-export-get-tags’, ‘org-export-get-category’ and ‘org-export-get-node-property’ extract useful information from an headline or a parent headline. They all handle inheritance.

‘org-export-get-alt-title’ tries to retrieve an alternative title, as a secondary string, suitable for table of contents. It falls back onto default title.

```
(defun org-export-get-relative-level (headline info)
  "Return HEADLINE relative level within current parsed tree.
INFO is a plist holding contextual information."
```

```
(defun org-export-low-level-p (headline info)
  "Non-nil when HEADLINE is considered as low level.

INFO is a plist used as a communication channel.

A low level headlines has a relative level greater than
‘:headline-levels’ property value.

Return value is the difference between HEADLINE relative level
and the last level being considered as high enough, or nil."


(defun org-export-get-headline-number (headline info)
  "Return HEADLINE numbering as a list of numbers.
INFO is a plist holding contextual information."


(defun org-export-numbered-headline-p (headline info)
  "Return a non-nil value if HEADLINE element should be numbered.
INFO is a plist used as a communication channel."


(defun org-export-number-to-roman (n)
  "Convert integer N into a roman numeral."


(defun org-export-get-tags (element info &optional tags inherited)
  "Return list of tags associated to ELEMENT.

ELEMENT has either an ‘headline’ or an ‘inlinetask’ type.  INFO
is a plist used as a communication channel.

Select tags (see ‘org-export-select-tags’) and exclude tags (see
‘org-export-exclude-tags’) are removed from the list.

When non-nil, optional argument TAGS should be a list of strings.
Any tag belonging to this list will also be removed.

When optional argument INHERITED is non-nil, tags can also be
```

```
inherited from parent headlines and FILETAGS keywords."


(defun org-export-get-node-property (property blob &optional inherited)
  "Return node PROPERTY value for BLOB.

PROPERTY is an upcase symbol (i.e. ‘:COOKIE_DATA’).  BLOB is an
element or object.

If optional argument INHERITED is non-nil, the value can be
inherited from a parent headline.

Return value is a string or nil."


(defun org-export-get-category (blob info)
  "Return category for element or object BLOB.

INFO is a plist used as a communication channel.

CATEGORY is automatically inherited from a parent headline, from
#+CATEGORY: keyword or created out of original file name.  If all
fail, the fall-back value is \"???\"."


(defun org-export-get-alt-title (headline info)
  "Return alternative title for HEADLINE, as a secondary string.
INFO is a plist used as a communication channel.  If no optional
title is defined, fall-back to the regular title."


(defun org-export-first-sibling-p (headline info)
  "Non-nil when HEADLINE is the first sibling in its sub-tree.
INFO is a plist used as a communication channel."


(defun org-export-last-sibling-p (headline info)
  "Non-nil when HEADLINE is the last sibling in its sub-tree.
INFO is a plist used as a communication channel."
```

11.6 For Keywords

‘org-export-get-date’ returns a date appropriate for the document to about to be exported. In particular, it takes care of ‘org-export-date-timestamp-format’.

```
(defun org-export-get-date (info &optional fmt)
  "Return date value for the current document.

INFO is a plist used as a communication channel.  FMT, when
non-nil, is a time format string that will be applied on the date
if it consists in a single timestamp object.  It defaults to
‘org-export-date-timestamp-format’ when nil.

A proper date can be a secondary string, a string or nil.  It is
meant to be translated with ‘org-export-data’ or alike."
```

11.7 For Links

‘org-export-solidify-link-text’ turns a string into a safer version for links, replacing most non-standard characters with hyphens.

‘org-export-get-coderef-format’ returns an appropriate format string for coderefs.

‘org-export-inline-image-p’ returns a non-nil value when the link provided should be considered as an inline image.

‘org-export-resolve-fuzzy-link’ searches destination of fuzzy links (i.e. links with "fuzzy" as type) within the parsed tree, and returns an appropriate unique identifier when found, or nil.

‘org-export-resolve-id-link’ returns the first headline with specified id or custom-id in parse tree, the path to the external file with the id or nil when neither was found.

‘org-export-resolve-coderef’ associates a reference to a line number in the element it belongs, or returns the reference itself when the element isn’t numbered.

```
(defun org-export-solidify-link-text (s)
  "Take link text S and make a safe target out of it."


(defun org-export-get-coderef-format (path desc)
  "Return format string for code reference link.
PATH is the link path.  DESC is its description."


(defun org-export-inline-image-p (link &optional rules)
  "Non-nil if LINK object points to an inline image.

Optional argument is a set of RULES defining inline images.  It
is an alist where associations have the following shape:

  \(TYPE . REGEXP)

Applying a rule means apply REGEXP against LINK's path when its
type is TYPE.  The function will return a non-nil value if any of
the provided rules is non-nil.  The default rule is
‘org-export-default-inline-image-rule’.

This only applies to links without a description."


(defun org-export-resolve-coderef (ref info)
  "Resolve a code reference REF.

INFO is a plist used as a communication channel.

Return associated line number in source code, or REF itself,
depending on src-block or example element's switches."

(defun org-export-resolve-fuzzy-link (link info)
  "Return LINK destination.

INFO is a plist holding contextual information.

Return value can be an object, an element, or nil:

- If LINK path matches a target object (i.e. <<path>>) return it.

- If LINK path exactly matches the name affiliated keyword
  \(i.e. #+NAME: path) of an element, return that element.
```

```
- If LINK path exactly matches any headline name, return that
  element.  If more than one headline share that name, priority
  will be given to the one with the closest common ancestor, if
  any, or the first one in the parse tree otherwise.

- Otherwise, return nil.

Assume LINK type is \"fuzzy\".  White spaces are not
significant."


(defun org-export-resolve-id-link (link info)
  "Return headline referenced as LINK destination.

INFO is a plist used as a communication channel.

Return value can be the headline element matched in current parse
tree, a file name or nil.  Assume LINK type is either \"id\" or
\"custom-id\"."


(defun org-export-resolve-radio-link (link info)
  "Return radio-target object referenced as LINK destination.

INFO is a plist used as a communication channel.

Return value can be a radio-target object or nil.  Assume LINK
has type \"radio\"."
```

11.8 For References

'org-export-get-ordinal' associates a sequence number to any object or element.

```
(defun org-export-get-ordinal (element info &optional types predicate)
  "Return ordinal number of an element or object.

ELEMENT is the element or object considered.  INFO is the plist
used as a communication channel.
```

```
Optional argument TYPES, when non-nil, is a list of element or
object types, as symbols, that should also be counted in.
Otherwise, only provided element's type is considered.

Optional argument PREDICATE is a function returning a non-nil
value if the current element or object should be counted in.  It
accepts two arguments: the element or object being considered and
the plist used as a communication channel.  This allows to count
only a certain type of objects (i.e. inline images).

Return value is a list of numbers if ELEMENT is a headline or an
item.  It is nil for keywords.  It represents the footnote number
for footnote definitions and footnote references.  If ELEMENT is
a target, return the same value as if ELEMENT was the closest
table, item or headline containing the target.  In any other
case, return the sequence number of ELEMENT among elements or
objects of the same type."
```

11.9 For Src-Blocks

‘org-export-get-loc’ counts number of code lines accumulated in src-block or example-block elements with a "+n" switch until a given element, excluded. Note: "-n" switches reset that count.

‘org-export-unravel-code’ extracts source code (along with a code references alist) from an ‘element-block’ or ‘src-block’ type element.

‘org-export-format-code’ applies a formatting function to each line of code, providing relative line number and code reference when appropriate. Since it doesn’t access the original element from which the source code is coming, it expects from the code calling it to know if lines should be numbered and if code references should appear.

Eventually, ‘org-export-format-code-default’ is a higher-level function (it makes use of the two previous functions) which handles line numbering and code references inclusion, and returns source code in a format suitable for plain text or verbatim output.

```
(defun org-export-get-loc (element info)
  "Return accumulated lines of code up to ELEMENT.

INFO is the plist used as a communication channel.
```

```
ELEMENT is excluded from count."


(defun org-export-unravel-code (element)
  "Clean source code and extract references out of it.

ELEMENT has either a ‘src-block’ an ‘example-block’ type.

Return a cons cell whose CAR is the source code, cleaned from any
reference and protective comma and CDR is an alist between
relative line number (integer) and name of code reference on that
line (string)."

(defun org-export-format-code (code fun &optional num-lines ref-alist)
  "Format CODE by applying FUN line-wise and return it.

CODE is a string representing the code to format.  FUN is
a function.  It must accept three arguments: a line of
code (string), the current line number (integer) or nil and the
reference associated to the current line (string) or nil.

Optional argument NUM-LINES can be an integer representing the
number of code lines accumulated until the current code.  Line
numbers passed to FUN will take it into account.  If it is nil,
FUN’s second argument will always be nil.  This number can be
obtained with ‘org-export-get-loc’ function.

Optional argument REF-ALIST can be an alist between relative line
number (i.e. ignoring NUM-LINES) and the name of the code
reference on it.  If it is nil, FUN’s third argument will always
be nil.  It can be obtained through the use of
‘org-export-unravel-code’ function."


(defun org-export-format-code-default (element info)
  "Return source code from ELEMENT, formatted in a standard way.

ELEMENT is either a ‘src-block’ or ‘example-block’ element.  INFO
is a plist used as a communication channel.
```

```
This function takes care of line numbering and code references
inclusion.  Line numbers, when applicable, appear at the
beginning of the line, separated from the code by two white
spaces.  Code references, on the other hand, appear flushed to
the right, separated by six white spaces from the widest line of
code."
  ;; Extract code and references.
```

11.10 For Tables

'org-export-table-has-special-column-p' and and 'org-export-table-row-is-special-p' are predicates used to look for meta-information about the table structure.

'org-table-has-header-p' tells when the rows before the first rule should be considered as table's header.

'org-export-table-cell-width', 'org-export-table-cell-alignment' and 'org-export-table-cell-borders' extract information from a table-cell element.

'org-export-table-dimensions' gives the number on rows and columns in the table, ignoring horizontal rules and special columns. 'org-export-table-cell-address', given a table-cell object, returns the absolute address of a cell. On the other hand, 'org-export-get-table-cell-at' does the contrary.

'org-export-table-cell-starts-colgroup-p', 'org-export-table-cell-ends-colgroup-p', 'org-export-table-row-starts-rowgroup-p', 'org-export-table-row-ends-rowgroup-p', 'org-export-table-row-starts-header-p' and 'org-export-table-row-ends-header-p' indicate position of current row or cell within the table.

```
(defun org-export-table-has-special-column-p (table)
  "Non-nil when TABLE has a special column.
All special columns will be ignored during export."

(defun org-export-table-has-header-p (table info)
  "Non-nil when TABLE has a header.

INFO is a plist used as a communication channel.

A table has a header when it contains at least two row groups."


(defun org-export-table-row-is-special-p (table-row info)
  "Non-nil if TABLE-ROW is considered special.

INFO is a plist used as the communication channel.

All special rows will be ignored during export."


(defun org-export-table-row-group (table-row info)
  "Return TABLE-ROW's group number, as an integer.

INFO is a plist used as the communication channel.

Return value is the group number, as an integer, or nil for
special rows and rows separators.  First group is also table's
header."


(defun org-export-table-cell-width (table-cell info)
  "Return TABLE-CELL contents width.

INFO is a plist used as the communication channel.

Return value is the width given by the last width cookie in the
same column as TABLE-CELL, or nil."


(defun org-export-table-cell-alignment (table-cell info)
  "Return TABLE-CELL contents alignment.

INFO is a plist used as the communication channel.

Return alignment as specified by the last alignment cookie in the
same column as TABLE-CELL.  If no such cookie is found, a default
alignment value will be deduced from fraction of numbers in the
column (see ‘org-table-number-fraction’ for more information).
Possible values are ‘left’, ‘right’ and ‘center’."


(defun org-export-table-cell-borders (table-cell info)
  "Return TABLE-CELL borders.
```

```
INFO is a plist used as a communication channel.

Return value is a list of symbols, or nil.  Possible values are:
‘top’, ‘bottom’, ‘above’, ‘below’, ‘left’ and ‘right’.  Note:
‘top’ (resp. ‘bottom’) only happen for a cell in the first
row (resp. last row) of the table, ignoring table rules, if any.

Returned borders ignore special rows."


(defun org-export-table-cell-starts-colgroup-p (table-cell info)
  "Non-nil when TABLE-CELL is at the beginning of a row group.
INFO is a plist used as a communication channel."

(defun org-export-table-cell-ends-colgroup-p (table-cell info)
  "Non-nil when TABLE-CELL is at the end of a row group.
INFO is a plist used as a communication channel."
  ;; A cell ends a column group either when it is at the end of a row
  ;; or when it has a right border.


(defun org-export-table-row-starts-rowgroup-p (table-row info)
  "Non-nil when TABLE-ROW is at the beginning of a column group.
INFO is a plist used as a communication channel."


(defun org-export-table-row-ends-rowgroup-p (table-row info)
  "Non-nil when TABLE-ROW is at the end of a column group.
INFO is a plist used as a communication channel."


(defun org-export-table-row-starts-header-p (table-row info)
  "Non-nil when TABLE-ROW is the first table header’s row.
INFO is a plist used as a communication channel."


(defun org-export-table-row-ends-header-p (table-row info)
  "Non-nil when TABLE-ROW is the last table header’s row.
INFO is a plist used as a communication channel."
```

```
(defun org-export-table-row-number (table-row info)
  "Return TABLE-ROW number.
INFO is a plist used as a communication channel.  Return value is
zero-based and ignores separators.  The function returns nil for
special colums and separators."


(defun org-export-table-dimensions (table info)
  "Return TABLE dimensions.

INFO is a plist used as a communication channel.

Return value is a CONS like (ROWS . COLUMNS) where
ROWS (resp. COLUMNS) is the number of exportable
rows (resp. columns)."


(defun org-export-table-cell-address (table-cell info)
  "Return address of a regular TABLE-CELL object.

TABLE-CELL is the cell considered.  INFO is a plist used as
a communication channel.

Address is a CONS cell (ROW . COLUMN), where ROW and COLUMN are
zero-based index.  Only exportable cells are considered.  The
function returns nil for other cells."


(defun org-export-get-table-cell-at (address table info)
  "Return regular table-cell object at ADDRESS in TABLE.

Address is a CONS cell (ROW . COLUMN), where ROW and COLUMN are
zero-based index.  TABLE is a table type element.  INFO is
a plist used as a communication channel.

If no table-cell, among exportable cells, is found at ADDRESS,
return nil."
```

11.11 For Tables Of Contents

'org-export-collect-headlines' builds a list of all exportable headline elements, maybe limited to a certain depth. One can then easily parse it and transcode it.

Building lists of tables, figures or listings is quite similar. Once the generic function 'org-export-collect-elements' is defined, 'org-export-collect-tables', 'org-export-collect-figures' and 'org-export-collect-listings' can be derived from it.

```
(defun org-export-collect-headlines (info &optional n)
  "Collect headlines in order to build a table of contents.

INFO is a plist used as a communication channel.

When optional argument N is an integer, it specifies the depth of
the table of contents.  Otherwise, it is set to the value of the
last headline level.  See 'org-export-headline-levels' for more
information.

Return a list of all exportable headlines as parsed elements.
Footnote sections, if any, will be ignored."

(defun org-export-collect-elements (type info &optional predicate)
  "Collect referenceable elements of a determined type.

TYPE can be a symbol or a list of symbols specifying element
types to search.  Only elements with a caption are collected.

INFO is a plist used as a communication channel.

When non-nil, optional argument PREDICATE is a function accepting
one argument, an element of type TYPE.  It returns a non-nil
value when that element should be collected.

Return a list of all elements found, in order of appearance."

(defun org-export-collect-tables (info)
  "Build a list of tables.
INFO is a plist used as a communication channel.
```

```
Return a list of table elements with a caption."


(defun org-export-collect-figures (info predicate)
  "Build a list of figures.

INFO is a plist used as a communication channel.  PREDICATE is
a function which accepts one argument: a paragraph element and
whose return value is non-nil when that element should be
collected.

A figure is a paragraph type element, with a caption, verifying
PREDICATE.  The latter has to be provided since a \"figure\" is
a vague concept that may depend on back-end.

Return a list of elements recognized as figures."


(defun org-export-collect-listings (info)
  "Build a list of src blocks.

INFO is a plist used as a communication channel.

Return a list of src-block elements with a caption."
```

11.12 Smart Quotes

The main function for the smart quotes sub-system is ‘org-export-activate-smart-quotes’, which replaces every quote in a given string from the parse tree with its "smart" counterpart.

Dictionary for smart quotes is stored in ‘org-export-smart-quotes-alist’.

Internally, regexps matching potential smart quotes (checks at string boundaries are also necessary) are defined in ‘org-export-smart-quotes-regexps’.

```
(defconst org-export-smart-quotes-alist
  '(("de"
     (opening-double-quote :utf-8 "„" :html "&bdquo;" :latex "\"‘"
                           :texinfo "@quotedblbase{}")
     (closing-double-quote :utf-8 "“" :html "&ldquo;" :latex "\"’"
                           :texinfo "@quotedblleft{}")
     (opening-single-quote :utf-8 "‚" :html "&sbquo;" :latex "\\glq{}"
                           :texinfo "@quotesinglbase{}")
     (closing-single-quote :utf-8 "‘" :html "&lsquo;" :latex "\\grq{}"
                           :texinfo "@quoteleft{}")
     (apostrophe :utf-8 "’" :html "&rsquo;"))
    ("en"
     (opening-double-quote :utf-8 "“" :html "&ldquo;" :latex "‘‘" :texinfo "‘‘")
     (closing-double-quote :utf-8 "”" :html "&rdquo;" :latex "’’" :texinfo "’’")
     (opening-single-quote :utf-8 "‘" :html "&lsquo;" :latex "‘" :texinfo "‘")
     (closing-single-quote :utf-8 "’" :html "&rsquo;" :latex "’" :texinfo "’")
     (apostrophe :utf-8 "’" :html "&rsquo;"))
    ("es"
     (opening-double-quote :utf-8 "«" :html "&laquo;" :latex "\\guillemotleft{}"
                           :texinfo "@guillemetleft{}")
     (closing-double-quote :utf-8 "»" :html "&raquo;" :latex "\\guillemotright{}"
                           :texinfo "@guillemetright{}")
     (opening-single-quote :utf-8 "“" :html "&ldquo;" :latex "‘‘" :texinfo "‘‘")
     (closing-single-quote :utf-8 "”" :html "&rdquo;" :latex "’’" :texinfo "’’")
     (apostrophe :utf-8 "’" :html "&rsquo;"))
    ("fr"
     (opening-double-quote :utf-8 "«" :html "&laquo;&nbsp;" :latex "\\og "
                           :texinfo "@guillemetleft{}@tie{}")
     (closing-double-quote :utf-8 "»" :html "&nbsp;&raquo;" :latex "\\fg{}"
                           :texinfo "@tie{}@guillemetright{}")
     (opening-single-quote :utf-8 "«" :html "&laquo;&nbsp;" :latex "\\og "
                           :texinfo "@guillemetleft{}@tie{}")
     (closing-single-quote :utf-8 "»" :html "&nbsp;&raquo;" :latex "\\fg{}"
                           :texinfo "@tie{}@guillemetright{}")
     (apostrophe :utf-8 "’" :html "&rsquo;")))
  "Smart quotes translations.

Alist whose CAR is a language string and CDR is an alist with
quote type as key and a plist associating various encodings to
their translation as value.

A quote type can be any symbol among ‘opening-double-quote’,
‘closing-double-quote’, ‘opening-single-quote’,
‘closing-single-quote’ and ‘apostrophe’.
```

```
Valid encodings include ‘:utf-8’, ‘:html’, ‘:latex’ and
‘:texinfo’.

If no translation is found, the quote character is left as-is.")

(defconst org-export-smart-quotes-regexps
  (list
   ;; Possible opening quote at beginning of string.
   "\\‘\\([\"’]\\)\\(\\w\\|\\s.\\|\\s_\\)"
   ;; Possible closing quote at beginning of string.
   "\\‘\\([\"’]\\)\\(\\s-\\|\\s)\\|\\s.\\)"
   ;; Possible apostrophe at beginning of string.
   "\\‘\\(’\\)\\S-"
   ;; Opening single and double quotes.
   "\\(?:\\s-\\|\\s(\\)\\([\"’]\\)\\(?:\\w\\|\\s.\\|\\s_\\)"
   ;; Closing single and double quotes.
   "\\(?:\\w\\|\\s.\\|\\s_\\)\\([\"’]\\)\\(?:\\s-\\|\\s)\\|\\s.\\)"
   ;; Apostrophe.
   "\\S-\\(’\\)\\S-"
   ;; Possible opening quote at end of string.
   "\\(?:\\s-\\|\\s(\\)\\([\"’]\\)\\’"
   ;; Possible closing quote at end of string.
   "\\(?:\\w\\|\\s.\\|\\s_\\)\\([\"’]\\)\\’"
   ;; Possible apostrophe at end of string.
   "\\S-\\(’\\)\\’")
  "List of regexps matching a quote or an apostrophe.
In every regexp, quote or apostrophe matched is put in group 1.")

(defun org-export-activate-smart-quotes (s encoding info &optional original)
  "Replace regular quotes with \"smart\" quotes in string S.

ENCODING is a symbol among ‘:html’, ‘:latex’, ‘:texinfo’ and
‘:utf-8’.  INFO is a plist used as a communication channel.

The function has to retrieve information about string
surroundings in parse tree.  It can only happen with an
unmodified string.  Thus, if S has already been through another
process, a non-nil ORIGINAL optional argument will provide that
original string.
```

```
Return the new string."
```

11.13 Topology

Here are various functions to retrieve information about the neighbourhood of a given element or object. Neighbours of interest are direct parent ('org-export-get-parent'), parent headline ('org-export-get-parent-headline'), first element containing an object, ('org-export-get-parent-element'), parent table ('org-export-get-parent-table'), previous element or object ('org-export-get-previous-element') and next element or object ('org-export-get-next-element').

'org-export-get-genealogy' returns the full genealogy of a given element or object, from closest parent to full parse tree.

```
(defsubst org-export-get-parent (blob)
  "Return BLOB parent or nil.
BLOB is the element or object considered."


(defun org-export-get-genealogy (blob)
  "Return full genealogy relative to a given element or object.

BLOB is the element or object being considered.

Ancestors are returned from closest to farthest, the last one
being the full parse tree."


(defun org-export-get-parent-headline (blob)
  "Return BLOB parent headline or nil.
BLOB is the element or object being considered."


(defun org-export-get-parent-element (object)
  "Return first element containing OBJECT or nil.
OBJECT is the object to consider."
```

```
(defun org-export-get-parent-table (object)
  "Return OBJECT parent table or nil.
OBJECT is either a ‘table-cell’ or ‘table-element’ type object."


(defun org-export-get-previous-element (blob info &optional n)
  "Return previous element or object.

BLOB is an element or object.  INFO is a plist used as
a communication channel.  Return previous exportable element or
object, a string, or nil.

When optional argument N is a positive integer, return a list
containing up to N siblings before BLOB, from farthest to
closest.  With any other non-nil value, return a list containing
all of them."


(defun org-export-get-next-element (blob info &optional n)
  "Return next element or object.

BLOB is an element or object.  INFO is a plist used as
a communication channel.  Return next exportable element or
object, a string, or nil.

When optional argument N is a positive integer, return a list
containing up to N siblings after BLOB, from closest to farthest.
With any other non-nil value, return a list containing all of
them."
```

11.14 Translation

‘org-export-translate’ translates a string according to language specified by LANGUAGE keyword or ‘org-export-language-setup’ variable and a specified charset. ‘org-export-dictionary’ contains the dictionary used for the translation.

```
(defconst org-export-dictionary
  '(("Author"
     ("ca" :default "Autor")
     ("cs" :default "Autor")
     ("da" :default "Ophavsmand")
     ("de" :default "Autor")
     ("eo" :html "A&#365;toro")
     ("es" :default "Autor")
     ("fi" :html "Tekij&auml;")
     ("fr" :default "Auteur")
     ("hu" :default "Szerz&otilde;")
     ("is" :html "H&ouml;fundur")
     ("it" :default "Autore")
     ("ja" :html "&#33879;&#32773;" :utf-8 "")
     ("nl" :default "Auteur")
     ("no" :default "Forfatter")
     ("nb" :default "Forfatter")
     ("nn" :default "Forfattar")
     ("pl" :default "Autor")
     ("ru" :html "&#1040;&#1074;&#1090;&#1086;&#1088;" :utf-8 "")
     ("sv" :html "F&ouml;rfattare")
     ("uk" :html "&#1040;&#1074;&#1090;&#1086;&#1088;" :utf-8 "")
     ("zh-CN" :html "&#20316;&#32773;" :utf-8 "")
     ("zh-TW" :html "&#20316;&#32773;" :utf-8 ""))
    ("Date"
     ("ca" :default "Data")
     ("cs" :default "Datum")
     ("da" :default "Dato")
     ("de" :default "Datum")
     ("eo" :default "Dato")
     ("es" :default "Fecha")
     ("fi" :html "P&auml;iv&auml;m&auml;&auml;r&auml;")
     ("hu" :html "D&aacute;tum")
     ("is" :default "Dagsetning")
     ("it" :default "Data")
     ("ja" :html "&#26085;&#20184;" :utf-8 "")
     ("nl" :default "Datum")
     ("no" :default "Dato")
     ("nb" :default "Dato")
     ("nn" :default "Dato")
     ("pl" :default "Data")
     ("ru" :html "&#1044;&#1072;&#1090;&#1072;" :utf-8 "")
     ("sv" :default "Datum")
     ("uk" :html "&#1044;&#1072;&#1090;&#1072;" :utf-8 "")
     ("zh-CN" :html "&#26085;&#26399;" :utf-8 "")
     ("zh-TW" :html "&#26085;&#26399;" :utf-8 ""))
    ("Equation"
     ("fr" :ascii "Equation" :default "Équation"))
    ("Figure")
    ("Footnotes"
     ("ca" :html "Peus de p&agrave;gina")
     ("cs" :default "Pozn\xe1mky pod carou")
     ("da" :default "Fodnoter")
     ("de" :html "Fu&szlig;noten")
     ("eo" :default "Piednotoj")
     ("es" :html "Pies de p&aacute;gina")
     ("fi" :default "Alaviitteet")
     ("fr" :default "Notes de bas de page")
     ("hu" :html "L&aacute;bjegyzet")
     ("is" :html "Aftanm&aacute;lsgreinar")
     ("it" :html "Note a pi&egrave; di pagina")
     ("ja" :html "&#33050;&#27880;" :utf-8 "")
     ("nl" :default "Voetnoten")
     ("no" :default "Fotnoter")
     ("nb" :default "Fotnoter")
     ("nn" :default "Fotnotar")
     ("pl" :default "Przypis")
     ("ru" :html "&#1057;&#1085;&#1086;&#1089;&#1082;&#1080;" :utf-8 "")
     ("sv" :default "Fotnoter")
     ("uk" :html "&#1055;&#1088;&#1080;&#1084;&#1110;&#1090;&#1082;&#1080;"
      :utf-8 "")
     ("zh-CN" :html "&#33050;&#27880;" :utf-8 "")
     ("zh-TW" :html "&#33139;&#35387;" :utf-8 ""))
    ("List of Listings"
     ("fr" :default "Liste des programmes"))
    ("List of Tables"
     ("fr" :default "Liste des tableaux"))
    ("Listing %d:"
     ("fr"
      :ascii "Programme %d :" :default "Programme nº %d :"
      :latin1 "Programme %d :"))
    ("Listing %d: %s"
     ("fr"
      :ascii "Programme %d : %s" :default "Programme nº %d : %s"
      :latin1 "Programme %d : %s"))
    ("See section %s"
     ("fr" :default "cf. section %s"))
    ("Table %d:"
     ("fr"
      :ascii "Tableau %d :" :default "Tableau nº %d :" :latin1 "Tableau %d :"))
    ("Table %d: %s"
     ("fr"
      :ascii "Tableau %d : %s" :default "Tableau nº %d : %s"
      :latin1 "Tableau %d : %s"))
    ("Table of Contents"
     ("ca" :html "&Iacute;ndex")
     ("cs" :default "Obsah")
     ("da" :default "Indhold")
     ("de" :default "Inhaltsverzeichnis")
     ("eo" :default "Enhavo")
     ("es" :html "&Iacute;ndice")
     ("fi" :html "Sis&auml;llysluettelo")
     ("fr" :ascii "Sommaire" :default "Table des matières")
     ("hu" :html "Tartalomjegyz&eacute;k")
     ("is" :default "Efnisyfirlit")
     ("it" :default "Indice")
     ("ja" :html "&#30446;&#27425;" :utf-8 "")
     ("nl" :default "Inhoudsopgave")
     ("no" :default "Innhold")
     ("nb" :default "Innhold")
     ("nn" :default "Innhald")
     ("pl" :html "Spis tre&#x015b;ci")
     ("ru" :html "&#1057;&#1086;&#1076;&#1077;&#1088;&#1078;&#1072;&#1085;&#1080;&#107
      :utf-8 "")
     ("sv" :html "Inneh&aring;ll")
     ("uk" :html "&#1047;&#1084;&#1110;&#1089;&#1090;" :utf-8 "")
     ("zh-CN" :html "&#30446;&#24405;" :utf-8 "")
     ("zh-TW" :html "&#30446;&#37636;" :utf-8 ""))
    ("Unknown reference"
     ("fr" :ascii "Destination inconnue" :default "Référence inconnue")))
  "Dictionary for export engine.
```

```
Alist whose CAR is the string to translate and CDR is an alist
whose CAR is the language string and CDR is a plist whose
properties are possible charsets and values translated terms.

It is used as a database for ‘org-export-translate’. Since this
function returns the string as-is if no translation was found,
the variable only needs to record values different from the
entry.")

(defun org-export-translate (s encoding info)
  "Translate string S according to language specification.

ENCODING is a symbol among ‘:ascii’, ‘:html’, ‘:latex’, ‘:latin1’
and ‘:utf-8’.  INFO is a plist used as a communication channel.

Translation depends on ‘:language’ property. Return the
translated string. If no translation is found, try to fall back
to ‘:default’ encoding. If it fails, return S."
  (let* ((lang (plist-get info :language))
         (translations (cdr (assoc lang
                                   (cdr (assoc s org-export-dictionary))))))
    (or (plist-get translations encoding)
        (plist-get translations :default)
        s)))
```

12 Asynchronous Export

‘org-export-async-start’ is the entry point for asynchronous export. It recreates current buffer (including visibility, narrowing and visited file) in an external Emacs process, and evaluates a command there. It then applies a function on the returned results in the current process.

Asynchronously generated results are never displayed directly. Instead, they are stored in ‘org-export-stack-contents’. They can then be retrieved by calling ‘org-export-stack’.

Export Stack is viewed through a dedicated major mode ‘org-export-stack-mode’ and tools: ‘org-export-stack-refresh’, ‘org-export-stack-delete’, ‘org-export-stack-view’ and ‘org-export-stack-clear’.

For back-ends, ‘org-export-add-to-stack’ add a new source to stack. It should used whenever ‘org-export-async-start’ is called.

```
(defmacro org-export-async-start  (fun &rest body)
  "Call function FUN on the results returned by BODY evaluation.

BODY evaluation happens in an asynchronous process, from a buffer
which is an exact copy of the current one.

Use ‘org-export-add-to-stack’ in FUN in order to register results
in the stack.  Examples for, respectively a temporary buffer and
a file are:

  \(org-export-async-start
      \(lambda (output)
        \(with-current-buffer (get-buffer-create \"*Org BACKEND Export*\")
        \(erase-buffer)
        \(insert output)
        \(goto-char (point-min))
        \(org-export-add-to-stack (current-buffer) ’backend)))
    ‘(org-export-as ’backend ,subtreep ,visible-only ,body-only ’,ext-plist))

and

  \(org-export-async-start
      \(lambda (f) (org-export-add-to-stack f ’backend))
    ‘(expand-file-name
      \(org-export-to-file
       ’backend ,outfile ,subtreep ,visible-only ,body-only ’,ext-plist)))"

(defun org-export-add-to-stack (source backend &optional process)
  "Add a new result to export stack if not present already.

SOURCE is a buffer or a file name containing export results.
BACKEND is a symbol representing export back-end used to generate
it.

Entries already pointing to SOURCE and unavailable entries are
removed beforehand.  Return the new stack."


(defun org-export-stack ()
  "Menu for asynchronous export results and running processes."
```

```
(defun org-export--stack-source-at-point ()
  "Return source from export results at point in stack."

(defun org-export-stack-clear ()
  "Remove all entries from export stack."
    (setq org-export-stack-contents nil))

(defun org-export-stack-refresh (&rest dummy)
  "Refresh the asynchronous export stack.
DUMMY is ignored.  Unavailable sources are removed from the list.
Return the new stack."


(defun org-export-stack-remove (&optional source)
  "Remove export results at point from stack.
If optional argument SOURCE is non-nil, remove it instead."

(defun org-export-stack-view (&optional in-emacs)
  "View export results at point in stack.
With an optional prefix argument IN-EMACS, force viewing files
within Emacs."

(defconst org-export-stack-mode-map
  (let ((km (make-sparse-keymap)))
    (define-key km " " 'next-line)
    (define-key km "n" 'next-line)
    (define-key km "\C-n" 'next-line)
    (define-key km [down] 'next-line)
    (define-key km "p" 'previous-line)
    (define-key km "\C-p" 'previous-line)
    (define-key km "\C-?" 'previous-line)
    (define-key km [up] 'previous-line)
    (define-key km "C" 'org-export-stack-clear)
    (define-key km "v" 'org-export-stack-view)
    (define-key km (kbd "RET") 'org-export-stack-view)
    (define-key km "d" 'org-export-stack-remove)
    km)
  "Keymap for Org Export Stack.")
```

```
(define-derived-mode org-export-stack-mode special-mode "Org-Stack"
  "Mode for displaying asynchronous export stack.

Type \\[org-export-stack] to visualize the asynchronous export
stack.

In an Org Export Stack buffer, use \\<org-export-stack-mode-map>\\[org-export-stack-vi
on current line, \\[org-export-stack-remove] to remove it from the stack and \\[org-exp
stack completely.

Removing entries in an Org Export Stack buffer doesn't affect
files or buffers, only the display.

\\{org-export-stack-mode-map}"
  (abbrev-mode 0)
  (auto-fill-mode 0)
  (setq buffer-read-only t
        buffer-undo-list t
        truncate-lines t
        header-line-format
        '(:eval
          (format "  %-12s | %6s | %s" "Back-End" "Age" "Source")))
  (org-add-hook 'post-command-hook 'org-export-stack-refresh nil t)
  (set (make-local-variable 'revert-buffer-function)
       'org-export-stack-refresh))
```

13 The Dispatcher

‘org-export-dispatch’ is the standard interactive way to start an export process. It uses ‘org-export–dispatch-ui’ as a subroutine for its interface, which, in turn, delegates response to key pressed to ‘org-export–dispatch-action’.

###autoload

```
(defun org-export-dispatch (&optional arg)
  "Export dispatcher for Org mode.

It provides an access to common export related tasks in a buffer.
Its interface comes in two flavours: standard and expert.

While both share the same set of bindings, only the former
```

```
displays the valid keys associations in a dedicated buffer.
Scrolling (resp. line-wise motion) in this buffer is done with
SPC and DEL (resp. C-n and C-p) keys.

Set variable ‘org-export-dispatch-use-expert-ui’ to switch to one
flavour or the other.

When ARG is \\[universal-argument], repeat the last export action, with the same set
of options used back then, on the current buffer.

When ARG is \\[universal-argument] \\[universal-argument], display the asynchronous ex

(defun org-export--dispatch-ui (options first-key expertp)
  "Handle interface for ‘org-export-dispatch’.

OPTIONS is a list containing current interactive options set for
export.  It can contain any of the following symbols:
‘body’    toggles a body-only export
‘subtree’ restricts export to current subtree
‘visible’ restricts export to visible part of buffer.
‘force’   force publishing files.
‘async’   use asynchronous export process

FIRST-KEY is the key pressed to select the first level menu.  It
is nil when this menu hasn’t been selected yet.

EXPERTP, when non-nil, triggers expert UI.  In that case, no help
buffer is provided, but indications about currently active
options are given in the prompt.  Moreover, \[?] allows to switch
back to standard interface."

(defun org-export--dispatch-action
  (prompt allowed-keys backends options first-key expertp)
  "Read a character from command input and act accordingly.

PROMPT is the displayed prompt, as a string.  ALLOWED-KEYS is
a list of characters available at a given step in the process.
BACKENDS is a list of menu entries.  OPTIONS, FIRST-KEY and
EXPERTP are the same as defined in ‘org-export--dispatch-ui’,
which see.
```

```
Toggle export options when required.  Otherwise, return value is
a list with action as CAR and a list of interactive export
options as CDR."

(provide 'ox)
```

Local variables: generated-autoload-file: "org-loaddefs.el" End:

14 ox.el ends here