Tools Application (TOOLS)

version 1.4

Kenneth Lundin

1997-05-02

Contents

Chapter 1

TOOLS User's Guide

The Erlang Tools Application (*TOOLS*) contains a number of stand-alone tools which are useful when developing Erlang programs. The current set of tools are:

coast A beta version of the new tool for coverage and call statistics analysis of Erlang programs.

dbg A text based trace facility. Makes it possible to trace function calls, messages etc. Can be used instead of, or as a complement to the *Trace Tool*.

eprof An Erlang profiler; measure how time is used in your Erlang programs.

emacs A Erlang editing mode for Emacs.

exref A cross reference tool. Can be used to check external references between Erlang programs.

instrument Utility functions for obtaining and analysing resource usage in an instrumented Erlang system.

make A make utility for Erlang programs (similar to UNIX make).

tags A module for creating TAGS file for the emacs tags functions.

1.1   The Erlang editing mode for Emacs

Introduction

If you want to get started immediately, the chapters "An Example for UNIX [page 28]" and "An Example for Windows [page 28]" gives you examples of the configurations you need to make to use the Erlang Editing mode for Emacs.

Emacs has been the text editor of choice for programmers in the UNIX community for many years. Thanks to a continuing development process, Emacs is the most powerful editor available. Today, Emacs runs under most operating systems including MS-Windows, OS/2, Macintosh, and several dialects of UNIX.

Emacs has editing support for all major programming languages and quite a lot of minor and unknown languages are supported as well.

Emacs is designed to be extendible. In the unlikely event that you would miss a feature in Emacs you can add it yourself, or you might find it in the large number of add-on packages that people all over the world have written.

This book is the documentation to the Emacs package `erlang.el`. It provides support for the programming language Erlang. The package provides an editing mode with lots of bells and whistles, compilation support, and it makes it possible for the user to start Erlang shells that run inside Emacs.

Emacs is written by the Free Software Foundation and is part of the GNU project. Emacs, including the source code and documentation, is released under the GNU General Public License.

Overview of this Book

This book can be divided into the following sections:

- *Introduction.*   This part introduces Emacs, the Erlang editing mode, and this book. In fact, this is the section you currently are reading.
- *The editing mode.*   Here the editing mode is described. The editing mode contains a whole series of features including indentation, syntax highlighting, electric commands, module name verification, comment support including paragraph filling, skeletons, tags support, and much more.
- *Erlang shells.*   How to start and use an Erlang shell that runs inside Emacs is described in this section.
- *Compilation support.*   This package is capable of starting compilations of Erlang module. Should compilation errors occur Emacs is capable of placing the cursor on the erroneous lines.
- *Customization.*   The Erlang editing mode, like most Emacs packages, supports extensive customization. In this chapter we demonstrate how you can bind your favorite functions to the hotkeys on the keyboard. It also introduces the concept of "hooks", a general method for the user to add code that will be executed when a specific situation occur, for example when an Erlang file is loaded into Emacs.

The terminology used in this book is the terminology used in the documentation to Emacs. The chapter "Notation [page 28]" contains a list of commonly used words and their meaning in the Emacs world.

The intended readers of this book are Emacs users. The book contains some examples on how to customize this package using the Emacs extension language Emacs Lisp. You can safely skip those sections.

Emacs

The first component needed to get this package up and running is, of course, an Emacs editor. You can use either the standard Emacs distribution from FSF or XEmacs, an alternative distribution. Both brands have their advantages and disadvantages.

Regardless of the brand, it is recommended to use a modern version. If an old version is used it is possible that some of the features provided by the editing mode can't be used.

The chapter "Emacs Distributions [page 25]" below contains a short summary on the differences between the Emacs brands, as well as instructions where to get the distributions and how to install them.

Installing the Erlang Support Packages

Once Emacs has been installed, it must be informed about the presence of the Erlang support packages.

If you don't know if the packages have been installed open, an Erlang source file. The mode line should contain the word "Erlang". You can check the version of the installed package by selecting the "version" entry in the Erlang menu in Emacs. Should no Erlang menu be present, or if the menu doesn't contain a "Version" item, you are using an old version.

The packages can either be installed for all users by the system administrator, or each individual user can install it in their own Emacs setup. The chapter "Installation of the Erlang Editing Mode [page 26]" contains a description on how to install the packages.

The Editing Mode

The Erlang editing for Emacs provides a number of features described in this and the following chapters. The editing mode can work with either Erlang source mode or Mnesia database rules. The Erlang editing mode for Emacs is in Emacs terminology a *Major mode* .

When Erlang mode is correctly installed, it is automatically activated when a file ending in `.erl` or `.hrl` is opened in Emacs. It is possible to activate Erlang mode for other buffers as well.

The editing mode provides a menu containing a selection of commands structured into logical subgroups. The menu is designed to help new users get an overview of the features provided by the Erlang packages while still giving full power to more advanced users.

Erlang mode has got a local key map that contains keyboard bindings for a number of commands. In the chapter "Custom Key Bindings [page 24]" below, we will demonstrate how the users can bind their favorite commands to the local Erlang key map.

It is possible for the users to perform advanced customizations by adding their own functions to the "hook" variables provided by this package. This will be described in the "Customization [page 6]" chapter below.

The Mode

- `M-x erlang-mode RET`
  This command activates the Erlang major mode for the current buffer. When this mode is active the mode line contain the word "Erlang".

The Version

- `M-x erlang-version RET`
  This command displays the version number of the Erlang editing mode. Remember to always supply the version number when asking questions about Erlang mode.
  Should this command not be present in your setup (after Erlang mode has been activated) you probably have a very old version of the Erlang editing mode.

Module Name Check

When a file is saved the name in the `-module().` line is checked against the file name. Should they mismatch Emacs can change the module specifier so that it matches the file name. By default, the user is asked before the change is performed.

- *Variable:* `erlang-check-module-name` (default `ask`)
  This variable controls the behavior of the module name check system. When it is t Emacs changes the module specifier without asking the user, when it is bound to the atom `ask` the user is asked. Should it be `nil` the module name check mechanism is deactivated.

Variables

There are several variables that control the behavior of the Erlang Editing mode.

- *Variable:* `erlang-mode-hook`
  Functions to run when the Erlang mode is activated. See chapter "Customization [page 6]" below for examples.
- *Variable:* `erlang-new-file-hook`
  Functions to run when a new file is created. See chapter "Customization [page 6]" below for examples.
- *Variable:* `erlang-mode-load-hook`
  Functions to run when the `erlang` package is loaded into Emacs. See chapter "Customization [page 6]" below for examples.

Indentation

The "Oxford Advanced Learners Dictionary of Current English" says the following about the word "indent":

> "start (a line of print or writing) farther from the margin than the others".

Possibly the most important feature of an editor designed for programmers is the ability to indent a line of code in accordance with the structure of the programming language.

The Erlang mode does, of course, provide this feature. The layout used is based on the common use of the language.

It is strongly recommend to use this feature and avoid to indent lines in a nonstandard way. Some motivations are:

- Code using the same layout is easy to read and maintain.
- The indentation features can be used to reindent large sections of a file. If some lines use nonstandard indentation they will be reindented.

- Since several features of Erlang mode is based on the standard layout they might not work correctly if a nonstandard layout is used. For example, the movement commands (described in chapter "Function and clause commands [page 12]" below) will not work unless the function headers start in the first column.

The Layout

The basic layout is that the clause headers start in the first column, and the bodies of clauses and complex expressions (e.g. "case" and "if") are indented more that the surrounding code. For example:

```
remove_bugs([]) ->
    [];
remove_bugs([X | Xs])
    case X of
        bug ->
            test(Xs);
        _ ->
            [X | test(Xs)]
    end.
```

- *Variable:* `erlang-indent-level`
  The depth of the indentation is controlled by the variable "erlang-indent-level", see section "Customization [page 6]" below.

Indentation of comments

Lines containing comment are indented differently depending on the number of %-characters used:

- Lines with one %-character is indented to the right of the code. The column is specified by the variable `comment-column`, by default column 48 is used.
- Lines with two %-characters will be indented to the same depth as code would have been in the same situation.
- Lines with three of more %-characters are indented to the left margin.

Example:

```
%%%
%%% Function: remove_bugs
%%%

remove_bugs([]) ->
    [];
remove_bugs([X | Xs])
    case X of
        bug ->                                  % Oh no, a bug!
                                                % Remove it.
            test(Xs);
        %% This element is not a bug, let's keep it.
        _ ->
            [X | test(Xs)]
    end.
```

Indentation commands

The following command are directly available for indentation.

- `TAB (erlang-indent-command)`
  Indent the current line of code.

- `M-C-\ (indent-region)`
  Indent all lines in the region.

- `M-l (indent-for-comment)`
  Insert a comment character to the right of the code on the line (if any). The comment character is placed in the column specified by the variable "comment-column", by default column 48 is used.

- `C-c C-q (erlang-indent-function)`
  Indent the current Erlang function.

- `M-x erlang-indent-clause RET`
  Indent the current Erlang clause.

- `M-x erlang-indent-current-buffer RET`
  Indent the entire buffer.

Customization

The most common customization of the indentation system is to bind the return key to `newline-and-indent`. Please see the chapter "Custom Key Bindings [page 24]" below for an example.

There are several Emacs variables that control the indentation system.

- *Variable:* `erlang-indent-level` (default 4)
  The amount of indentation for normal Erlang functions and complex expressions. Should, for example, the value of this variable be 2 the example above would be indented like:

```
remove_bugs([]) ->
  [];
remove_bugs([X | Xs])
  case X of
    bug ->
      test(Xs);
    _ ->
      [X | test(Xs)]
  end.
```

- *Variable:* `erlang-indent-guard` (default 2)
  The amount of indentation for Erlang guards.

- *Variable:* `erlang-argument-indent` (default 2)
  The amount of indentation for function calls that span several lines.
  Example:

```
foo() ->
    a_very_long_function_name(
      AVeryLongVariableName),
```

- *Variable:* `erlang-tab-always-indent` (default `t`)
  When non-`nil` the `TAB` command always indents the line (this is the default). When `nil`, the line will be indented only when the point is in the beginning of any text on the line, otherwise it will insert a tab character into the buffer.

General Commands

This chapter contains a group of commands that are not found in any other category. Unlike most other books we don't have a chapter named "Miscellaneous xxx" found at the end of most books. This chapter is placed near the beginning to reflect the importance and usefulness of the commands.

Filling comments

How many times have you edited a section of text in a comment only to wind up with a unevenly formatted paragraph? Or even worse, have you ever decided not to edit a comment just because the formatting would look bad?

When editing normal text in text mode you can let Emacs reformat the text by the `fill-paragraph` command. This command will not work for comments since it will treat the comment characters as words.

The Erlang editing mode provides a command that known about the Erlang comment structure and can be used to fill text paragraphs in comments.

- `M-q (erlang-fill-paragraph)`
  Fill the text in an Erlang comment. This command known about the Erlang comment characters. The column to perform the word wrap is defined by the variable `fill-column`.

Example:

For the sake of this example, let's assume that `fill-column` is set to column 30. Assume that we have an Erlang comment paragraph on the following form:

```
%% This is just a test to show
%% how the Erlang fill
%% paragraph command works.
```

Assume that you would add the words "very simple" before the word "test":

```
%% This is just a very simple test to show
%% how the Erlang fill
%% paragraph command works.
```

Clearly, the text is badly formatted. Instead of formatting this paragraph line by line, let's try `erlang-fill-paragraph` by pressing `M-q`. The result is:

```
%% This is just a very simple
%% test to show how the Erlang
%% fill paragraph command
%% works.
```

As you can see the paragraph is now evenly formatted.

Creating Comments

In Erlang it is possible to write comments to the right of the code. The indentation system described in the chapter "Indentation" above is able to indent lines containing only comments, and gives support for end-of-the-line comments.

- `M-; (indent-for-comment)`
  This command will create, or reindent, a comment to the right of the code. The variable `comment-column` controls the placement of the comment character.

Comment Region

The standard command `comment-region` can be used to comment out all lines in a region. To uncomment the lines in a region precede this command with `C-u`.

Syntax Highlighting

It is possible for Emacs to use colors when displaying a buffer. By "syntax highlighting", we mean that syntactic components, for example keywords and function names, will be colored.

The basic idea of syntax highlighting is to make the structure of a program clearer. For example, the highlighting will make it easier to spot simple bugs. Haven't you ever written a variable in lower-case only? With syntax highlighting a variable will colored while atoms will be shown with the normal text color.

The syntax highlighting can be activated from the Erlang menu. There are four different alternatives:

- Off: Normal black and white display.
- Level 1: Function headers, reserved words, comments, strings, quoted atoms, and character constants will be colored.
- Level 2: The above, attributes, Erlang bif:s, guards, and words in comments enclosed in single quotes will be colored.
- Level 3: The above, variables, records, and macros will be colored. (This level is also known as the Christmas tree level.)

The syntax highlighting is based on the standard Emacs package "font-lock". It is possible to use the font-lock commands and variables to enable syntax highlighting. The commands in question are:

- `M-x font-lock-mode RET`
  This command activates syntax highlighting for the current buffer.
- `M-x global-font-lock-mode RET`
  Activate syntax highlighting for all buffers.

The variable `font-lock-maximum-decoration` is used to specify the level of highlighting. If the variable is bound to an integer, that level is used; if it is bound to `t` the highest possible level is used. (It is possible to set different levels for different editing modes; please see the font-lock documentation for more information.)

It is possible to change the color used. It is even possible to use bold, underlined, and italic fonts in combination with colors. However, the method to do this differs between Emacs and XEmacs; and between different versions of Emacs. For Emacs 19.34, the variable `font-lock-face-attributes` controls the colors. For version 20 of Emacs and XEmacs, the faces can be defined in the interactive custom system.

Customization

Font-lock mode is activated in different ways in different versions of Emacs. For modern versions of GNU Emacs place the following lines in your `~/.emacs` file:

```
(setq font-lock-maximum-decoration t)
(global-font-lock-mode 1)
```

For modern versions of XEmacs the following code can be used:

```
(setq auto-font-lock-mode 1)
```

For older versions of Emacs and XEmacs, font-lock mode must be activated individually for each buffer. The following will add a function to the Erlang mode hook that activates font-lock mode for all Erlang buffers.

```
(defun my-erlang-font-lock-hook ()
  (font-lock-mode 1))
```

```
(add-hook 'erlang-mode-hook 'my-erlang-font-lock-hook)
```

Known Problems

Emacs has one problem with the syntactic structure of Erlang, namely the $ character. The normal Erlang use of the $ character is to denote the ASCII value of a character, for example:

```
ascii_value_of_a() -> $a.
```

In order to get the font-lock mechanism to work for the next example, the $ character must be marked as an "escape" character that changes the ordinary Emacs interpretation of the following double-quote character.

```
ascii_value_of_quote() -> $".
```

The problem is that Emacs will also treat the $ character as an "escape" character at the end of strings and quoted atoms. Practically, this means that Emacs will not detect the end of the following string:

```
the_id() -> "$id: $".
```

Fortunately, there are ways around this. From Erlang's point of view the following two strings are equal: `"test$"` and `"test\$"`. The `\`-character is also marked as an Emacs "escape" character, hence it will change the Emacs interpretation of the $-character.

This work-around can't always be used. For example, when the string is used by an external version control program. In this situation we can try to avoid placing the $-character at the end of the string, for example:

```
-vsn(" $Revision: 1.1 $ ").
```

Should this not be possible we can try to create an artificial end of the string by placing an extra quote sign in the file. We do this as a comment:

```
-vsn("$Revision: 1.1 $").    % "
```

The comment will be ignored by Erlang since it's a comment. From Emacs point of view the comment character is part of the string.

This problem is a generic problem for languages with similar syntax. For example, the major mode for Perl suffers from the same problem.

Electric Commands

An "electric" command is a character that in addition to just inserting the character performs some type of action. For example the ";" character is typed in a situation where is ends a function clause a new function header is generated.

Since some people find electric commands annoying they can be deactivated, see section "Unplugging the Electric Commands [page 11]" below.

The Commands

- `;` `(erlang-electric-semicolon)`
  Insert a semicolon. When ending a function or the body of a case clause, and the next few lines are empty, the special action will be performed. For functions, a new function header will be generated and the point will be placed between the parentheses. (See the command `erlang-clone-arguments`.) For other clauses the string " `->`" will be inserted and the point will be placed in from of the arrow.

- `,` `(erlang-electric-comma)`
  Insert a comma. If the point is at the end of the line and the next few lines are empty, a new indented line is created.

- `>` `(erlang-electric-arrow)`
  Insert a > character. If it is inserted at the end of a line after a - character so that an arrow "`->`" is being formed, a new indented line is created. This requires that the next few lines are empty.

- `RET` `(erlang-electric-newline)`
  The special action of this command is normally off by default. When bound to the return key the following line will be indented. Should the current line contain a comment the initial comment characters will be copied to the new line. For example, assume that the point is at the end of a line (denoted by "`<point>`" below).

      ```
      %% A comment<point>
      ```

  When pressing return (and `erlang-electric-newline` is active) the result will be:

      ```
      %% A comment
      %% <point>
      ```

  This command has a second feature. When issued directly after another electric command that created a new line this command does nothing. The motivation is that it's in the fingers of many programmers to hit the return key just when they have, for example, finished a function clause with the `;` character. Without this feature both the electric semicolon and this command would insert one line each which is probably not what the user wants.

Undo

All electric command will set an undo marker after the initial character has been inserted but before the special action has been performed. By executing the undo command (`C-x u`) the effect of the special action will be undone while leaving the character. Execute undo a second time to remove the character itself.

Variables

The electric commands are controlled by a number of variables.

- `erlang-electric-commands`
  This variable controls if an electric command is active or not. This variable should contain a list of electric commands to be active. To activate all electric commands bind this variable to the atom `t`.

- `erlang-electric-newline-inhibit`
  When non-`nil` when `erlang-electric-newline` should do nothing when preceded by a electric command that is member of the list `erlang-electric-newline-inhibit-list`.

- `erlang-electric-newline-inhibit-list`
  A list of electric commands. The command `erlang-electric-newline` will do nothing when preceded by a command in this list, and the variable `erlang-electric-newline-inhibit` is non-`nil`.

- `erlang-electric-X-criteria`
  There is one variable of this form for each electric command. The variable is used to decide if the special action of an electric command should be used. The variable contains a list of criteria functions that are called in the order they appear in the list.
  If a criteria function returns the atom `stop` the special action is not performed. If it returns a non-`nil` value the action is taken. If it returns `nil` the next function in the list is called. Should no function in the list return a non-`nil` value the special action will not be executed. Should the list contain the atom `t` the special action is performed (unless a previous function returned the atom `stop`).

- `erlang-next-lines-empty-threshold` (default 2)
  Should the function `erlang-next-lines-empty-p` be part of a criteria list of an electric command (currently semicolon, comma, and arrow), this variable controls the number of blank lines required.

Unplugging the Electric Commands

To disable all electric commands set the variable `erlang-electric-commands` to the empty list. In short, place the following line in your `~/.emacs` file:

```
(setq erlang-electric-commands '())
```

Customizing the Electric Commands

To activate all electric commands, including `erlang-electric-newline`, add the following line to your `~/.emacs` file:

```
(setq erlang-electric-commands t)
```

Function and Clause Commands

The Erlang editing mode has a set of commands that are aware of the Erlang functions and function clauses. The commands can be used to move the point (cursor) to the end of, or to the beginning of Erlang functions, or to jump between functions. The region can be placed around a function. Function headers can be cloned (copied).

Movement Commands

There is a set of commands that can be used to move the point to the beginning or the end of an Erlang clause or function. The commands are also designed for movement between Erlang functions and clauses.

- `C-a M-a (erlang-beginning-of-function)`
  Move the point to the beginning of the current or preceding Erlang function. With an argument skip backwards over this many Erlang functions. Should the argument be negative the point is moved to the beginning of a function below the current function.
  This function returns `t` if a function was found, `nil` otherwise.

- `M-C-a (erlang-beginning-of-clause)`
  As above but move point to the beginning of the current or preceding Erlang clause.
  This function returns `t` if a clause was found, `nil` otherwise.

- `C-a M-e (erlang-end-of-function)`
  Move to the end of the current or following Erlang function. With an argument to it that many times. Should the argument be negative move to the end of a function above the current functions.

- `M-C-e (erlang-end-of-clause)`
  As above but move point to the end of the current or following Erlang clause.

When one of the movement commands is executed and the point is already placed at the beginning or end of a function or clause, the point is moved to the previous/following function or clause.

When the point is above the first or below the last function in the buffer, and an `erlang-beginning-of-`, or `erlang-end-of-` command is issued, the point is moved to the beginning or to the end of the buffer, respectively.

Development Tips   The functions described above can be used both as user commands and called as functions in programs written in Emacs Lisp.

Example:

The sequence below will move the point to the beginning of the current function even if the point should already be positioned at the beginning of the function:

```
(end-of-line)
(erlang-beginning-of-function)
```

Example:

To repeat over all the function in a buffer the following code can be used. It will first move the point to the beginning of the buffer, then it will locate the first Erlang function. Should the buffer contain no functions at all the call to `erlang-beginning-of-function` will return `nil` and hence the loop will never be entered.

```
(goto-char (point-min))
(erlang-end-of-function 1)
(let ((found-func (erlang-beginning-of-function 1)))
   (while found-func
     ;; Do something with this function.
     ;; Go to the beginning of the next function.
     (setq found-func (erlang-beginning-of-function -1))))
```

Region Commands

- `C-c M-h (erlang-mark-function)`
  Put the region around the current Erlang function. The point is placed in the beginning and the mark at the end of the function.

- `M-C-h (erlang-mark-clause)`
  Put the region around the current Erlang clause. The point is placed in the beginning and the mark at the end of the function.

Function Header Commands

- `C-c C-j (erlang-generate-new-clause)`
  Create a new clause in the current Erlang function. The point is placed between the parentheses of the argument list.

- `C-c C-y (erlang-clone-arguments)`
  Copy the function arguments of the preceding Erlang clause. This command is useful when defining a new clause with almost the same argument as the preceding.

Limitations

Several clauses are considered to be part of the same Erlang function if they have the same name. It is possible that in the future the arity of the function also will be checked. To avoid to perform a full parse of the entire buffer the functions described in the chapter only look at lines where the function starts in the first column. This means that the commands does not work properly if the source code contain non-standardized indentation.

Skeletons

A skeleton is a piece of pre-written code that can be inserted into the buffer. Erlang mode comes with a set of predefined skeletons ranging from simple `if` expressions to stand-alone applications.

The skeletons can be accessed either from the Erlang menu of from commands named `tempo-template-erlang-X`.

The skeletons is defined using the standard Emacs package "tempo". It is possible to define new skeletons for your favorite erlang constructions.

Commands

- `C-c M-f (tempo-forward-mark)`
- `C-c M-b (tempo-backward-mark)` In a skeleton certain positions are marked. These two commands move the point between such positions.

Predefined Skeletons

- Simple skeletons: If, Case, Receive, Receive After, Receive Loop.
- Header elements: Module, Author. These commands inserts lines on the form `-module(xxx).` and `-author('my@home')..` They can be used directly, but are also used as part of the full headers described below:
- Full Headers: Small, Medium, and Large Headers These commands generate three variants of file headers.

The following skeletons will complete almost ready-to-run modules.

- Small Server
- application
- Supervisor
- Supervisor Bridge
- gen_server
- gen_event
- gen_fsm

Defining New Skeletons

It is possible to define new Erlang skeletons. The skeletons are defined using the standard package "tempo". The skeleton is described using the following variables:

- `erlang-skel-X` (Where X is the name of this skeleton.)
  Each skeleton is described by a variable. It contains a list of Tempo rules. See below for two examples of skeleton definitions. See the Tempo Reference Manual for a complete description of tempo rules.

- `erlang-skel`
  This variable describes all Erlang skeletons. It is used to define the skeletons and to add them to the Erlang menu. The variable is a list where is each entry is either the empty list, representing a vertical bar in the menu, or a list on the form:

      (Menu-name tempo-name erlang-skel-X)

  The Menu-name is name to use in the menu. A named function is created for each skeleton, it is `tempo-template-erlang-tempo-name`. Finally, `erlang-skel-X` is the name of the variable describing the skeleton.
  The best time to change this variable is right after the Erlang mode has been loaded but before it has been activated. See the "Example" section below.

Examples   Below is two example on skeletons and one example on how to add an entry to the `erlang-skel` variable. Please see the Tempo reference manual for details about the format.

Example 1:

The "If" skeleton is defined by the following variable (slightly rearranged for pedagogical reasons):

```
(defvar erlang-skel-if
  '((erlang-skel-skip-blank)   ;;   1
    o                          ;;   2
    >                          ;;   3
    "if"                       ;;   4
    n>                         ;;   5
    p                          ;;   6
    " ->"                      ;;   7
    n>                         ;;   8
    p                          ;;   9
    "ok"                       ;;  10
    n>                         ;;  11
    "end"                      ;;  12
    p))                        ;;  13
```

Each line describes an action to perform:

- 1: This is a normal function call. Here we skip over any space characters after the point. (If we don't they will end up after the skeleton.)
- 2: This means "Open Line", i.e. split the current line at the point, but leave the point on the end of the first line.
- 3: Indent Line. This indents the current line.
- 4: Here we insert the string `if` into the buffer
- 5, 8, 11: Newline and indent.
- 6, 9, 13: Mark these positions as special. The point will be placed at the position of the first `p`. The point can later be moved to the other by the `tempo-forward-mark` and `tempo-backward-mark` described above.
- 7, 10, 12: These insert the strings " ->", "ok", and "end", respectively.

Example 2:

This example contains very few entries. Basically, what it does is to include other skeletons in the correct place.

```
(defvar erlang-skel-small-header
  '(o                                        ;;  1
    (erlang-skel-include erlang-skel-module  ;;  2
                         erlang-skel-author)
    n                                        ;;  3
    (erlang-skel-include erlang-skel-compile ;;  4
                         erlang-skel-export  ;;  5
                         erlang-skel-vc)))   ;;  6
```

The lines performs the following actions:

- 1: "Open Line" (see example 1 above).
- 2: Insert the skeletons `erlang-skel-module` and `erlang-skel-compile` into the buffer.
- 3: Insert one empty line.
- 4: Insert three more skeletons.

Example 3:

Here we assume that we have defined a new skeleton named `erlang-skel-example`. The best time to add this skeleton to the variable `erlang-skel` is when Erlang mode has been loaded but before it has been activated. We define a function that adds two entries to `erlang-skel`, the first is `()` that represent a divisor in the menu, the second is the entry for the `Example` skeleton. We then add the function to the `erlang-load-hook`, a hook that is called when Erlang mode is loaded into Emacs.

```
(defun my-erlang-skel-hook ()
  (setq erlang-skel
        (append erlang-skel
                '(()
                  ("Example" "example" erlang-skel-example)))))

(add-hook 'erlang-load-hook 'my-erlang-skel-hook)
```

Manual Pages

The UNIX version of Erlang tools contain a set of manual pages that can be accessed by the standard UNIX command "man". The Erlang mode place a list of all available manual pages in the "Erlang" menu.

Unfortunately this feature is not available in the Windows version of the Erlang editing mode since the Erlang tools are not delivered with the manual pages.

The Menu

In the Erlang menu a list of all Erlang manual pages can be found. The menu item "Man Pages". The sub-menu to this menu item contains a list of categories, normally "Man - Commands" and "Man - Modules". Under these is a menu containing the names of the man pages. Should this menu be to large it is split alphabetically into a number of sub-menus.

The menu item "Man - Function" is capable of finding the man page of a named Erlang function. This commands understands the `module:function` notation. This command defaults to the name under the point. Should the name not contain a module name the list of imported modules is searched.

Customization

The following variables control the manual page feature.

- `erlang-man-dirs`
  This variable is a list representing the sub-menu to the "Man Pages" menu item in the Erlang menu. Each element is a list with three elements. The first is the name of the menu, e.g. "Man - Modules" or "Man - Local Stuff". The second is the name of a directory. The third is a flag that control the interpretation of the directory. When `nil` the directory is treated as an absolute path, when non-`nil` it is taken as relative to the directory named in the variable `erlang-root-dir`.

- `erlang-man-max-menu-size`
  The maximum number of menu items in a manual page menu. If the number of manual pages would be more than this variable the menu will be split alphabetically into chunks each not larger than the value of this variable.

Tags

Tags is a standard Emacs package used to record information about source files in large development projects. In addition to listing the files of a project, a tags file normally contains information about all functions and variables that are defined. By far, the most useful command of the tags system is it's ability to find the definition of functions in any file in the project. However the Tags system is not limited to this feature, for example, it is possible to do a text search in all files in a project, or to perform a project-wide search and replace.

Creating a TAGS file

In order to use the Tags system a file named TAGS must be created. The file can be seen as a database over all functions, records, and macros in all files in the project. The TAGS file can be created using to different methods for Erlang. The first is the standard Emacs utility "etags", the second is by using the Erlang module tags.

The etags utility

The etags is a program that is part of the Emacs distribution. It is normally executed from a command line, like a unix shell or a DOS box.

The etags program of fairly modern versions of Emacs and XEmacs has native support for Erlang. To check if your version does include this support, issue the command etags --help at a the command line prompt. At the end of the help text there is a list of supported languages. Unless Erlang is a member of this list I suggest that you should upgrade to a newer version of Emacs.

As seen in the help text – unless you haven't upgraded your Emacs yet (well, what are you waiting around here for? Off you go and upgrade!) – etags associate the file extensions .erl and .hrl with Erlang.

Basically, the etags utility is runed using the following form:

```
etags file1.erl file2.erl
```

This will create a file named TAGS in the current directory.

The etags utility can also read a list of files from it's standard input by supplying a single dash in place of the file names. This feature is useful when a project consists of a large number of files. The standard UNIX command find can be used to generate the list of files, e.g:

```
file . -name "*.[he]rl" -print | etags -
```

The above line will create a TAGS file covering all the Erlang source files in the current directory, and in the subdirectories below.

Please see the GNU Emacs Manual and the etags man page for more info.

The code implementing the Erlang support for the etags program has been donated to the Free Software Foundation by the company Anders Lindgren Development.

The tags Erlang module

One of the tools in the Erlang distribution is a module named `tags`. This tool can create a `TAGS` file from Erlang source files.

The following are examples of useful functions in this module. Please see the reference manual on tags for details.

- `tags:file('foo.erl').`
  Create a `TAGS` file for the file "foo.erl".

- `tags:subdir('src/project/', [{outfile, 'project.TAGS'}]).`
  Create a tags file containing all Erlang source files in the directory `"src/project/"`. The option `outfile` specify the name of the created `TAGS` file.

- `tags:root([{outdir, 'bar'}]).`
  Create a `TAGS` file of all the Erlang files in the Erlang distribution. The `TAGS` file will be placed in the the directory `bar`.

Additional Erlang support

The standard Tags system has only support for simple names. The naming convention `module:function` used by Erlang is not supported.

The Erlang mode supplies an alternative set of Tags functions that is aware of the format `module:function`. When selecting a the default search string for the commands the name under the point is first selected. Should the name not contain a module name the `-import` list at the beginning of the buffer is scanned.

Limitations   Currently, the additional Erlang module name support is not compatible with the `etags.el` package that is part of XEmacs.

Useful Tags Commands

- `M-.   (erlang-find-tag)`
  Find a function definition. The default value is the function name under the point. Should the function name lack the module specifier the `-import` list is searched for an appropriate candidate.

- ` C-u M-.   (erlang-find-tag` with an argument)
  The `find-tag` commands place the point on the first occurrence of a function that match the tag. This command move the point to the next match.

- `C-x 4 .   (erlang-find-tag-other-window)`
  As above, but the new file will be shown in another window in the same frame.

- `C-x 5 .   (erlang-find-tag-other-frame)`
  As `erlang-find-tag` but the new file will be shown in a new frame.

- `M-TAB (erlang-complete-tag)`
  This command is used to fill in the end of a partially written function name. For example, assume that the point is at the end of the string `a_long`, and the Tags file contain the function `a_long_function_name`. By executing this command the string `a_long` will be expanded into `a_long_function_name`.

- ` M-x tags-search RET`
  This command will search through all the files in a project for a string. (Actually, it search for a pattern described by a regular expression.)

- M-, (tags-loop-continue)
  Move the point to the next search match.

IMenu

IMenu is a standard package of GNU Emacs. With IMenu it is possible to get a menu in the menu bar containing all the functions in the buffer. Erlang mode provides support for Erlang source files.

Starting IMenu

- M-x imenu-add-to-menubar RET
  This command will create the IMenu menu containing all the functions in the current buffer. The command will ask you for a suitable name for the menu.

Customization

See chapter "Customization [page 6]" below for a general description on how to customize the Erlang mode.

To automatically create the IMenu menu for all Erlang buffers, place the lines below into the appropriate init file (e.g. ~/.emacs). The function `my-erlang-imenu-hook` will be called each time an Erlang source file is read. It will call the `imenu-add-to-menubar` function. The menu will be named "Functions".

```
(add-hook 'erlang-mode-hook 'my-erlang-imenu-hook)

(defun my-erlang-imenu-hook ()
  (if (and window-system (fboundp 'imenu-add-to-menubar))
      (imenu-add-to-menubar "Functions")))
```

Running Erlang from Emacs

One of the strengths of Emacs is its ability to start slave processes. Since Emacs is extendible it is possible let Emacs be a part of a large application. For example, Emacs could be used as the user interface for Erlang applications.

The Erlang editing mode provides two simple, yet very useful, applications. The first is to start an Erlang shell and use an Emacs buffer for input and output. The second is a compile commands that makes it possible to compile Erlang modules and to locate the lines containing the compilation errors.

The actual communication between Emacs and Erlang can be performed by different low-level techniques. The Erlang editing mode provides a technique called "inferior" processes. The add on package Erl'em supplies a technically much more advanced communication technique known as an Erl'em link. All the commands that are provided by the editing mode can use either technique. However, more advanced packages will probably need features only provided by the Erl'em package.

Inferior Erlang

The editing mode is capable of starting a so called "inferior" Erlang process. This is a normal subprocess that use one Emacs buffer for input and output. The effect is that a command line shell, or an Erlang shell, can be displayed inside Emacs.

The big drawback with an inferior process is that the communication between Emacs and the process is limited to commands issued at the command line. Since this is the interface that is used by the user it is difficult, to say the least, to write an Emacs application that communicate with the inferior process. For example, the `erlang-compile` command described in the section "Compilation" below really stretch the capabilities of the inferior Erlang process. In fact, should the user have issued a command that would take some time to complete it is impossible for Emacs to perform the `erlang-compile` command.

The Erl'em Link

The Erl'em package established a low-level communication channel between Emacs and an Erlang node. This communication channel can be used by Emacs to issue several independent Erlang commands, to start Erlang processes and to open several Erlang IO streams. It is also possible for Erlang to call Emacs functions.

In short the Erl'em package is designed to be the base of complex application that is partially implemented in Emacs and partially in Erlang.

It is the hope of the author that the Erl'em link in the future will be used as the base for porting the user interface of the Erlang debugger to Emacs. If this could be possible, Emacs could be used as an Integrated Debugger Environment (IDE) for Erlang.

The structure of the Erl'em link and its programming interface is described in the text "Erl'em Developers Manual".

Erlang Shell

It is possible to start an Erlang shell inside Emacs. The shell will use an Emacs buffer for input and output. Normal Emacs commands can be used to edit the command line and to recall lines from the command line history.

The output will never be erased from the buffer so you will never risk letting important output fall over the top edge of the display.

As discussed in the previous chapter there are two low-level methods for Emacs to communicate with Erlang. The first is by starting an inferior process, the second is by using an Erl'em link. When using inferior processes each new shell will start a new Erlang node. Should the Erl'em link be used it is possible to start several shells on the same node, a feature not normally available.

The shell

In this section we describe how to start a shell. In the next we cover how to use it once it has been started.

- `M-x erlang-shell RET`
  Start a new Erlang shell. When an inferior process is used a new Erlang node is started for each shell. Should the Erl'em link package be installed several shells can be started on the same Erlang node.
  A word of warning. The Erlang function `halt().` will kill the current Erlang node, including all shells running on it.

- `M-x erlang-shell-display RET`
  Display one Erlang shell. If there are no Erlang shells active a new will be started.

Command line history

The look and feel on an Erlang shell inside Emacs should be the same as in a normal Erlang shell. There is just one major difference, the cursor keys will actually move the cursor around just like in any normal Emacs buffer. The command line history can be accessed by the following commands:

- `C-up` or `M-p` (`comint-previous-input`)
  Move to the previous line in the input history.

- `C-down` or `M-n` (`comint-next-input`)
  Move to the next line in the input history.

If the Erlang shell buffer would be killed the command line history is saved to a file. The command line history is automatically retrieved when a new Erlang shell is started.

The Erlang Shell Mode

The buffers that are used to run Erlang shells use the major mode `erlang-shell-mode`. This major mode is based on the standard mode `comint-mode`.

- `erlang-shell-mode`
  Enter Erlang shell mode. To operate correctly the buffer should be in Comint mode when this command is called.

Variables

In this section we present the variables that control the behavior of the Erlang shell. See also the next section "Inferior Erlang Variables".

- *Variable:* `erlang-shell-mode-hook` (default `()`)
  Function to run when this mode is activated. See chapter "Customization [page 6]" below for examples.

- *Variable:* `erlang-input-ring-file-name` (default "~/.erlang_history")
  The file name used to save the command line history.

- *Variable:* `erlang-shell-function` (default `inferior-erlang`)
  This variable contain the low-level function to call to start an Erlang shell. This variable will be changed by the Erl'em installation.

- *Variable:* `erlang-shell-display-function` (default `inferior-erlang-run-or-select`)
  This variable contain the low-level function to call when the `erlang-shell-display` is issued. This variable will be changed by the Erl'em installation.

Inferior Erlang Variables

The variables described in this chapter are only used when inferior Erlang processes are used. They do not affect the behavior of the shell when using an Erl'em link.

- *Variable:* `inferior-erlang-display-buffer-any-frame` (default `nil`)
  When this variable is `nil` the command `erlang-shell-display` will display the inferior process in the current frame. When `t`, it will do nothing when it already is visible in another frame. When it is bound to the atom `raise` the frame displaying the buffer will be raised.
- *Variable:* `inferior-erlang-shell-type` (default `newshell`)
  There are two different variants of the Erlang shell, named the old and the new shell. The old is a simple variant that does not provide command line editing facilities. The new, on the other hand, provide full edition features. Apart from this major difference, they differ on some subtle points. Since Emacs itself takes care of the command line edition features you can switch between the two shell types if your shell behaves strange.
  To use the new or the old shell bind this variable to `newshell` or `oldshell`, respectively.
- *Variable:* `inferior-erlang-machine` (default `"erl"`)
  The command name of the Erlang system.
- *Variable:* `inferior-erlang-machine-options` (default `()`)
  A list of strings containing command line options that is used when starting an inferior Erlang.
- *Variable:* `inferior-erlang-buffer-name` (default `"*erlang*"`)
  The base name of the Erlang shell buffer. Should several Erlang shell buffers be used they will be named `*erlang*<2>`, `*erlang*<3>` etc.

Compilation

The classic edit-compile-bugfix cycle for Erlang is to edit the source file in an editor, save it to a file and switch to an Erlang shell. In the shell the compilation command is given. Should the compilation fail you have to bring out the editor and locate the correct line.

With the Erlang editing mode the entire edit-compile-bugfix cycle can be performed without leaving Emacs. Emacs can order Erlang to compile a file and it can parse the error messages to automatically place the point on the erroneous lines.

Commands

- `C-c C-k` (`erlang-compile`)
  This command compiles the file in the current buffer.
  The action performed by this command depend on the low-level communication method used. Should an inferior Erlang process be used Emacs tries to issue a compile command at the Erlang shell prompt. The compilation output will be sent to the shell buffer. This command will fail if it's not possible to issue a command at the Erlang shell prompt.
  Should an Erl'em link be used the compile command sent to Erlang will be independent of any active shell. The output will be sent to a dedicated buffer.

- `C-x ‘` (erlang-next-error)
  This command will place the point on the line where the first error was found. Each successive use of this command will move the point to the next error. The buffer displaying the compilation errors will be updated so that the current error will be visible.
  You can reparse the compiler output from the beginning by preceding this command by `C-u`.

- `erlang-compile-display`
  Show the output generated by the compile command.

Variables

- *Variable:* `erlang-compile-use-outdir` (default `t`)
  In some versions of Erlang the `outdir` options contains a bug. Should the directory not be present in the current Erlang load path the object file will not be loaded.
  Should this variable be set to `nil` the `erlang-compile` command will use a workaround by change current directory, compile the file, and change back.

- *Variable:* `erlang-compile-function` (default `inferior-erlang-compile`)
  The low-level function to use to compile an Erlang module.

- *Variable:* `erlang-compile-display-function` (default `inferior-erlang-run-or-select`)
  The low-level function to call when the result of a compilation should be shown.

- *Variable:* `erlang-next-error-function` (default `inferior-erlang-next-error`)
  The low-level function to use when `erlang-next-error` is used.

Customization

One of the strengths of Emacs is that users can fairly easy customize the behavior of almost every detail. The Erlang editing mode is not an exception to this rule.

Normally, Emacs is customized through the user and system init files, `~/.emacs` and `site-start.el`, respectively. The content of the files are expressions written in the Emacs extension language Emacs Lisp. The semantics of Lisp is fairly similar Erlang's. However, the syntax is very different. Fortunately, most customizations require only very minor knowledge of the language.

Emacs Lisp

In this section we show the basic constructions of Emacs Lisp needed to perform customizations.

In addition to placing the expressions in the init file, they can be evaluated while Emacs is started. One method is to use the `M-:` (On older versions of Emacs this is bound to `ESC ESC`) function that evaluates Emacs Lisp expressions in the minibuffer. Another method is to write the expressions in the `*scratch*` buffer, place the point at the end of the line and press `C-j`.

Below is a series of example that we use to demonstrate simple Emacs Lisp constructions.

- *Example 1:*
  In this example we set the variable `foo` to the value 10 added to the value of the variable `a`. As we can see by this example, Emacs Lisp use prefix form for all function calls, including simple functions like +.

  ```
  (setq foo (+ 10 a))
  ```

- *Example 2:*
  In this example we first define a function `bar` that sums the value of it's four parameters. Then we evaluated an expression that first calls `bar` then calls the standard Emacs function `message`.

```
(defun bar (a b c d)
  (+ a b c d))

(message "The sum becomes %d" (bar 1 2 3 4))
```

- *Example 3:*
  Among the Emacs Lisp data types we have atoms. However, in the following expressions we assign the variable `foo` the value of the variable `bar`.

```
(setq foo bar)
```

  To assign the variable `foo` the atom `bar` we must quote the atom with a '-character. Note the syntax, we should precede the expression (in this case `bar`) with the quote, not surround it.

```
(setq foo 'bar)
```

Hooks

A hook variable is a variable that contain a list of functions to run. In Emacs there is a large number of hook variables, each is runed at a special situation. By adding functions to hooks the user make Emacs automatically perform anything (well, almost).

To add a function to a hook you must use the function `add-hook`. To remove it use `remove-hook`.

See chapter "The Editing Mode" above for a list of hooks defined by the Erlang editing mode.

- *Example:*
  In this example we add `tempo-template-erlang-large-header` to the hook `erlang-new-file-hook`. The effect is that whenever a new Erlang file is created a file header is immediately inserted.

```
    (add-hook 'erlang-new-file-hook 'tempo-template-erlang-large-header)
```

- *Example:*
  Here we define a new function that sets a few variables when it is called. We then add the function to the hook `erlang-mode-hook` that gets called every time erlang mode is activated.

```
(defun my-erlang-mode-hook ()
  (setq erlang-electric-commands t))

(add-hook 'erlang-mode-hook 'my-erlang-mode-hook)
```

Custom Key Bindings

It is possible to bind keys to your favorite commands. Emacs use a number of key-maps: the global key-map defines the default value of keys, local maps are used by the individual major modes, minor modes can have their own key map etc.

The commands `global-set-key` and `local-set-key` defines keys in the global and in the current local key-map, respectively.

If we would like to redefine a key in the Erlang editing mode we can do that by activating Erlang mode and call `local-set-key`. To automate this we must define a function that calls `local-set-key`. This function can then be added to the Erlang mode hook so that the correct local key map is active when the key is defined.

Example:

Here we bind `C-c C-c` to the command `erlang-compile`, the function key `f1` to `erlang-shell`, and `M-f1` to `erlang-shell-display` . The calls to `local-set-key` will not be performed when the init file is loaded, they will be called first when the functions in the hook `erlang-mode-hook` is called, i.e. when Erlang mode is started.

```
(defun my-erlang-keymap-hook ()
  (local-set-key (read-kbd-macro "C-c C-c") 'erlang-compile)
  (local-set-key (read-kbd-macro "<f1>")    'erlang-shell)
  (local-set-key (read-kbd-macro "M-<f1>")  'erlang-shell-display))
(add-hook 'erlang-mode-hook 'my-erlang-keymap-hook)
```

The function `read-kbd-macro` used in the above example converts a string of readable keystrokes into Emacs internal representation.

Example:

In Erlang mode the tags commands understand the Erlang module naming convention. However, the normal tags commands does not. This line will bind `M-.` in the global map to `erlang-find-tag`.

```
(global-set-key (read-kbd-macro "M-." 'erlang-find-tag))
```

Emacs Distributions

Today there are two major Emacs development streams. The first is GNU Emacs from Free Software Foundation and the second is XEmacs. Both have advantages and disadvantages, you have to decide for yourself which Emacs you prefer.

GNU Emacs

This is the standard distribution from The Free Software Foundation, an organization lead by the original author of Emacs, Richard M. Stallman.

The source code for the latest version of Emacs can be fetched from `http://www.gnu.org`. A binary distribution for Window NT and 95 can be found at `http://www.cs.washington.edu/homes/voelker/ntemacs.html`.

XEmacs

This is an alternative version of Emacs. Historically XEmacs is based on Lucid Emacs that in turn was based on an early version of Emacs 19. The big advantage of XEmacs is that it can handle graphics much better. One difference is a list of icons that contains a number of commonly used commands. Another is the ability to display graphical images in the buffer.

The major drawback is that when a new feature turns up in GNU Emacs, it will often take quite a long time before it will be incorporated into XEmacs.

The latest distribution can be fetched from `http://www.xemacs.org`.

Installing Emacs

The source distributions usually comes in a tared and gzipped format. Unpack this with the following command:

```
tar zxvf <file>.tar.gz
```

If your tar command don't know how to handle the "z" (unpack) option you can unpack it separately:

```
gunzip <file>.tar.gz
tar xvf <file>.tar
```

The program `gunzip` is part of the `gzip` package that can be found on the `http://www.gnu.org` site.

Next, read the file named `INSTALL`. The build process is normally performed in three steps: in the first the build system performs a number of tests on your system, the next step is to actually build the Emacs executable, finally Emacs is installed.

Installation of the Erlang Editing Mode

In the OTP installation, the Erlang editing mode is already installed. All that is needed is that the system administrator or the individual user configures their Emacs Init files to use it.

If we assume that OTP has been installed in *OTP_ROOT*, the editing mode can be found in *OTP_ROOT*/`misc/emacs`.

The `erlang.el` file found in the installation directory is already compiled. If it needs to be recompiled, the following command line should create a new `erlang.elc` file:

```
emacs -batch -q -no-site-file -f batch-byte-compile erlang.el
```

Editing the right Emacs Init file

System administrators edit `site-start.el`, individuals edit their `.emacs` files.

On UNIX systems, individuals should edit/create the file `.emacs` in their home directories.

On Windows NT/95, individuals should also edit/create their `.emacs` file, but the location of the file depends on the configuration of the system.

- If the *HOME* environment variable is set, Emacs will look for the `.emacs` file in the directory indicated by the *HOME* variable.
- If *HOME* is not set, Emacs will look for the `.emacs` file in `C:\`.

Extending the load path

The directory with the editing mode, *OTP_ROOT*/`misc/emacs`, must be in the load path for Emacs.

Add the following line to the selected initialization file (replace `OTP_ROOT` with the name of the installation directory for OTP, keep the quote characters):

```
(setq load-path (cons "OTP_ROOT/misc/emacs" load-path))
```

Note: When running under Windows, use `/` or `\\` as separator in pathnames in the Emacs configuration files. Using a single `\` in strings does not work, as it is interpreted by Emacs as an escape character.

Specifying the OTP installation directory

Some functions in the Erlang editing mode require that the OTP installation directory is known. The following is an example where we assume that they are installed in the directory `OTP_ROOT`, change this to reflect the location on your system.

```
(setq erlang-root-dir "OTP_ROOT")
```

Extending the execution path

To use inferior Erlang Shells, you need to do the following configuration. If your *PATH* environment variable already includes the location of the `erl` or `erl.exe` executable this configuration is not necessary.

You can either extend the *PATH* environment variable with the location of the `erl`/`erl.exe` executable. Please refer to instructions for setting environment variables on your particular platform for details.

You can also extend the execution path for Emacs as described below. If the executable is located in `OTP_ROOT`/`bin` then you add the following line to you Emacs Init file:

```
(setq exec-path (cons "OTP_ROOT/bin" exec-path))
```

Final setup

Finally, add the following line to the init file:

```
(require 'erlang-start)
```

This will inform Emacs that the Erlang editing mode is available. It will associate the file extensions `.erl` and `.hrl` with Erlang mode. Also it will make sure that files with the extension `.jam` will be ignored when using file name completion.

An Example for UNIX

Below is a complete example of what should be added to a user's `.emacs` provided that OTP is installed in the directory `/usr/local/otp`:

```
(setq load-path (cons  "/usr/local/otp/misc/emacs"
                        load-path))
(setq erlang-root-dir "/usr/local/otp")
(setq exec-path (cons "/usr/local/otp/bin" exec-path))
(require 'erlang-start)
```

Any additional user configurations can be added after this. See for instance section "Customization [page 9]" for some useful customizations.

An Example for Windows

Below is a complete example of what should be added to a user's `.emacs` provided that OTP is installed in the directory `C:\Program Files\erl-4.7`:

```
(setq load-path (cons  "C:/Program Files/erl-4.7/misc/emacs"
                        load-path))
(setq erlang-root-dir "C:/Program Files/erl-4.7")
(setq exec-path (cons "C:/Program Files/erl-4.7/bin" exec-path))
(require 'erlang-start)
```

Any additional user configurations can be added after this. See for instance section "Customization [page 9]" for some useful customizations.

Check the Installation

Restart the Emacs and load or create an Erlang file (with the `.erl` extension). If the installation was performed correctly the mode line should contain the word "Erlang". Select the "Version" menu item in the "Erlang" menu, check that the version number matches the version in found in the files in `OTP_ROOT/misc/emacs`.

Notation

In this book we use the same terminology used in the Emacs documentation. This chapter contain a short glossary of words and their meaning in the Emacs world.

- *Buffer* A buffer is used by Emacs to handle text. When editing a file the content is loaded into a buffer. However buffers can contain other things as well. For example, a buffer can contain a list of files in a directory, it can contain generated help texts, or it is possible to start processes that use a buffer in Emacs for input and output. A buffer need not be visible, but if it is, it is shown in a window.

- *Emacs Lisp* Emacs is written in two languages. The Emacs core is written in C. The major part, as well as most add-on packages, are written in Emacs Lisp. This is also the language used by the init files.

- *Frame* This is what most other systems refer to as a *window* . Emacs use frame since the word window was used for another feature long before window systems were invented.

- *init file* Files read by Emacs at startup. The user startup file is named `~/.emacs`. The init files are used to customize Emacs, for example to add new packages like `erlang`. The language used in the startup files is Emacs Lisp.
- *Major mode* A major mode provides support for edit text of a particular sort. For example, the Erlang editing mode is a major mode. Each buffer have exactly one major mode active.
- *Minor mode* A minor mode provides some additional support. Each buffer can have several minor modes active at the same time. One example is `font-lock-mode` that activates syntax highlighting, another is `follow-mode` that make two side-by-side windows act like one tall window.
- *Mode line* The line at the bottom of each Emacs window that contain information about the buffer. E.g. the name of the buffer, the line number, and the name of the the current major mode.
- `nil` The value used in Emacs Lisp to represent false. True can be represented by any non-`nil` value, but it is preferred to use `t`.
- *Point* The point can be seen as the position of the cursor. More precisely, the point is the position between two characters while the cursor is drawn over the character following the point.
- `t` The value `t` is used by flags in Emacs Lisp to represent true. See also `nil`.
- *Window* An area where text is visible in Emacs. A *frame* (which is a window in non-Emacs terminology) can contain one or more windows. New windows can be created by splitting windows either vertically or horizontally.

Keys

- `C-` The control key.
- `M-` The meta key. Normally this is the left ALT key. Alternatively the escape key can be used (with the difference that the escape key should be pressed and released while the ALT key work just like the control key.)
- `M-C-` Press both meta and control at the same time. (Or press the escape key, release it, and then press the control key.)
- `RET` The return key.

All commands in Emacs have names. A named command can be executed by pressing `M-x`, typing the name of the command, and hitting `RET` .

Further reading

In this chapter I present some references to material on Emacs. They are divided into the two categories "Usage" and "Development". The first is for normal Emacs users who would like to know how to get more power out of their editor. The second is for people who would like to develop their own applications in Emacs Lisp.

Personally, I would recommend the series of books from the Free Software Foundation, they are written by the people that wrote Emacs and they form a coherent series useful for everyone from beginners to experienced Emacs Lisp developers.

Usage

- Richard M. Stallman. GNU Emacs Manual. Free Software Foundation, 1995.
  This is the Bible on using Emacs. It is written by the principle author of Emacs. An on-line version of this manual is part of the standard Emacs distribution, see the "Help->Browse Manuals" menu.

- "comp.emacs", News Group on Usenet.
  General Emacs group, everything is discussed here from beginners to complex development issues.

- "comp.emacs.xemacs", News Group on Usenet.
  This group cover XEmacs only.

- "gnu.emacs.help", News Group on Usenet.
  This group is like "comp.emacs" except that the topic only should cover GNU Emacs, not XEmacs or any other Emacs derivate.

- "gnu.emacs.sources", News Group on Usenet.
  In this group a lot of interesting Emacs packages are posted. In fact only source code is permitted, questions should be redirected to one of the other Emacs groups.

- "gnu.emacs.bugs", News Group on Usenet.
  If you have found a bug in Emacs you should post it here. Do not post bug reports on packages that are nor part of the standard Emacs distribution, they should be sent to the maintainer of the package.

Development

- Robert J. Chassell. Programming in Emacs Lisp: an Introduction. Free Software Foundation, 1995.
  This a good introduction to Lisp in general and Emacs Lisp in particular. Just like the other books form FSF, this book is free and can be downloaded from `http://www.gnu.org` .

- Bil Lewis et.al. The GNU Emacs Lisp Reference Manual. Free Software Foundation, 1995.
  This is the main source of information for any serious Emacs developer. This manual covers every aspect of Emacs Lisp. This manual, like Emacs itself, is free. The manuscript can be downloaded from `http://www.gnu.org` and can either be converted into printable form or be converted into a hypertext on-line manual.

- Bob Glickstein. Writing GNU Emacs Extensions. O'Reilly, 1997.
  This is a good tutorial on how to write Emacs packages.

- Anders Lindgren. Erl'em Developers Manual. Ericsson, 1998.
  This text covers the architecture of the Erl'em communication link and the application programmers interface to it.
  The tempo package is presented in this manual. The latest version can be found at `http://www.lysator.liu.se` .

Reporting Bugs

Please send bug reports to the following email address:

```
support@erlang.ericsson.se
```

Please state as accurate as possible:

- Version number of the Erlang editing mode (see the menu), Emacs, Erlang, and of any other relevant software.

- What the expected result was.

- What you did, preferably in a repeatable step-by-step form.

- A description of the unexpected result.

- Relevant pieces of Erlang code causing the problem.

- Personal Emacs customizations, if any.

Should the Emacs generate an error, please set the emacs variable `debug-on-error` to `t`. Repeat the error and enclose the debug information in your bug-report.

To set the variable you can use the following command:

```
M-x set-variable RET debug-on-error RET t RET
```

1.2   The Profiler (eprof)

The Profiler (eprof)

The profiler `eprof` tool is used to profile a system in order to find out how much processing time various processes occupy.

The modules to be profiled must be compiled with the trace flag. The following functions are used to start and stop the `eprof` server, select modules to be profiled, and display the profiling results.

- `start() -> {ok, Pid} | {error, {already_started, Pid}}` starts the `eprof` server
- `stop() -> stopped` stops the `eprof` server
- `profile(Rootset, Mod, fun, Args)` profiles a process
- `profile(Rootset) -> profiling | error` starts profiling a process
- `stop_profiling() -> profiling_stopped | profiling_already_stopped` stops profiling
- `analyse() -> ok` displays the profiling results
- `total_analyse() -> ok` prints the profiling results
- `log(File) -> OK` activates logging of `eprof` printouts.

> **Note:**
> The trace flag slows the system slightly. The part of the system which is profiled runs at approximately 20% of its original speed.

1.3 The Cross-Reference Tool

The Cross-Reference Tool (exref)

`exref` is an incremental cross reference tool which builds a cross reference graph for selected modules. Information such as module dependencies and usage graphs can be derived from the cross reference graph produced by `exref`.

A function vertex is represented as: .

```
{{Mod, Fun, Arity}, {Type, File, Line}}
```

In this representation, the `Type` argument equals:

```
local | exported | {exported, compiler} | {local, compiler} | {Void(), record}
```

A call edge is represented as:

```
{EdgeId, {Mod1, Fun1, Arity1}, {Modd2, Fun2, Arity2}, Line}
```

The following functions are available for configuring and using the cross reference tool:

- `start() -> {ok, Pid} | {error, {already_started, Pid}}` starts the `exref` server. The server must be started before any other functions in module `exref` can be used.
- `stop() -> stopped` stops the `exref` server.
- `module(Module) -> true` loads the specified module, or modules, into the cross reference graph.
- `module(Module, Options) -> true` loads the module `Module` into the cross reference graph. `Module` may also be a list of modules. Type can be any of the following:
  1. `module = atom() | [atom()]`
  2. `Options = [Option]`, where
  3. `Option = search | verbose | auto | warnings | recursive`, where

  - `search` searches for source file in code path and replaces the path `X/ebin` with `X/src`.
  - `verbose` creates an output of module names during loading.
  - `auto` includes all referenced modules in the graph, with the exception of modules listed in the excludes(Modules) [page 34] function shown below.
  - `warnings` emits warnings about the application and the spawning of variables. The reason for this is that `apply` calls lead to an incomplete graph for variable modules or functions. The `apply` call is inserted into the graph instead of the actual call. The same applies to `spawn`.
  - `recursive` recursively includes all files in a directory.
- `directory(Directory)` loads all modules in a directory into the cross reference graph.
- `directory(Directory, Module)` loads specific modules from a directory other than the current directory into the cross reference graph.
- `directory(Directory, Module, Options)` loads specific modules from a directory other than the current directory into the cross reference graph.

- `delete_module(Module)` deletes a module from the cross reference path. `Module` can be a list of modules.

- `excludes(Modules)` excludes a module, or list of modules, from the cross reference path.

- `includes(Dirs)` specifies where to search for Erlang `include` files.

- `defs(Defs)` adds definitions to be used when processing source code. This function appends the definitions specified with `Defs` to the definition list used by `erl_pp`.

- `analyse(Type [,Arg]) -> Result` analyses the cross reference graph and returns an Erlang term of a format which depends on the `Type` specified. The result from this analysis can be pretty printed with the `pretty/1` function listed below. Some of `Type` can have optional arguments. Refer to the Reference Manual, the section `tools`, module `exref` for details.

- `pretty(AnalyseResult) -> ok` prints a verbose textual representation of the analysis result produced by the `analyse/2` function shown above. The result produced from a user defined analysis cannot be used as input to this function.

1.4   TOOLS Release Notes

These release notes concern the following tools:

- `coast`
- `cover`
- `dbg`
- Emacs support for Erlang (GNU Emacs only)
- `eprof`
- `exref`
- `instrument`
- `make`
- `tags`

Tools 1.4.1

Improvements and New Features

Coast -

Cover -

Dbg -

Emacs Support for Erlang -

Eprof -

Exref -

Instrument -

Make -

Tags -

Fixed Bugs and Malfunctions

Coast

- Recursive function calls containing other function calls as arguments are now recognised as being recursive.
  Own Id: OTP-1981
- There can no longer be a race condition if two (or more) processes, running the same code, tries to update the ETS tables used.
  Own Id: OTP-2625

Cover The `cover` module itself hasn't been changed. However, some malfunctions have been removed in the new coverage tool, `coast`, please see above for more details.

Dbg -

Emacs Support for Erlang -

Eprof -

Exref -

Instrument -

Make -

Tags -

Incompatibilities With Tools v1.4

Coast

- The `run` function call will, if the call to the Coast-compiled module fails, return an exit signal instead of the tuple {'EXIT', Reason}.
- The internal representation of data has been changed, meaning that all previously Coast-compiled modules have to be recompiled.

Cover -

Dbg -

Emacs Support for Erlang -

Eprof -

Exref -

Instrument -

Make -

Tags -

Known Bugs and Problems

Coast

Cover

- Two or more processes, running the same code, may in rare situations give rise to race conditions when updating the ETS table used.
  Own Id: OTP-2625
- `cover` counts the number of executed statements, instead of the number of function calls. (This problem is solved by using `coast` instead!)
  Own Id: OTP-1247
- `cover` doesn't store the accumulated data persistently.
  Own Id: OTP-1318

Dbg -

Emacs Support for Erlang -

Eprof -

Exref -

Instrument -

Make -

Tags -

Tools 1.4

Please note there is a new document module creating Emacs TAGS files, `tags`.

Improvements and New Features

Coast -

Cover -

Dbg -

Emacs Support for Erlang

- Check auto generated Release Notes.

Tags Added a reference manual.

Eprof -

Exref -

Make -

Fixed Bugs and Malfunctions

Coast -

Cover The `cover` module itself hasn't been changed. However, some malfunctions have been removed in the new coverage tool, `coast`, please see above for more details, and also the section `Known Bugs` below.

Dbg -

Emacs Support for Erlang -

Eprof -

Exref

- `exref:delete_module/1` now works properly.
  Own Id: OTP-1115
- `exref` no longer have problems with guards (such as `record` and `atom`) in list comprehension (LC) expressions.
  Own Id: OTP-1291
- `exref` no longer mixes up modules
  Own Id: OTP-1220
- `exref` no longer changes the working directory.
- `exref` now handles internal functions correctly under certain conditions.
- When using the `auto` option, exref no longer enters an infinite loop.

Make   -

Incompatibilities With Tools v1.2.1

There should be no noticeable incompatibilities between Tools v1.2.2 and the previous versions. However, for the sake of completeness, some minor issues that may change the behaviour for users taking advantage of undocumented and unsupported behaviour are listed below.

Coast   -

Cover   -

Dbg   -

Emacs Support for Erlang   -

Eprof   -

Exref

- `exref:module` no longer accepts modules specified as strings, so the implementation now agrees with the documentation.

Make   -

Known Bugs and Problems

Coast

- A recursive function call containing another function call as argument is currently not recognised as being recursive, unless the function call used as argument is made to a module not having been coast-compiled.
  Own Id: OTP-1981

Cover

- `cover` counts the number of executed statements, instead of the number of function calls. (This problem is solved by using `coast` instead!)
  Own Id: OTP-1247
- `cover` doesn't store the accumulated data persistently.
  Own Id: OTP-1318

Dbg   -

Emacs Support for Erlang

- Occurences of `$` confuses indentation mechanism.
  Own Id: OTP-1203, OTP-1674, OTP-1675

Eprof   -

Exref   -

Make   -

Tools v1.2.1

Improvements and new features

-

Incompatibilities with Tools 1.2

-

Fixed bugs and malfunctions

make • The function `make:all` now works withoout a `Emakefile`. Own Id: OTP-1340

exref • `exref` now know about the language construct `query`. Own Id: OTP-1368

-

Known bugs and problems

exref • The function `exref:delete_module/1` does not work properly, so subesquent calls to `exref:analyse` will give incorrect results. Own Id: OTP-1115

Tools version 1.2

Improvements and new features

-

Incompatibilities with Tools 1.1

-

Fixed bugs and malfunctions

cover • Deadlock situation with COVER compile solved.
Own Id: OTP-1072, Aux Id: HA34612.

  • Cover does now work with include files.
OwnId: OTP-1074, Aux Id: HA34302

  • Changed cover to used `filename:find_src/1` instead of the obsolete `file_util` module.

Known bugs and problems

exref • Not cleaning up when module removed.

Tools version 1.1

Improvements and new features

exref • -

cover • -

Incompatibilities with Tools in OTP P1H

Fixed bugs and malfunctions

exref • -

cover • Cover did not use `Options` in `file/2` in a correct way (OTP-1074).

• Cover tried to load the "tools" application dynamically which could cause deadlock in the startup if cover-compiled modules are involved at this stage. The attempt to load the application is now removed (must be specified in the start script).

Known bugs and problems

Tools Application (TOOLS)

TOOLS Reference Manual

Short Summaries

- Erlang Module **coast** [page 48] – Coverage and Statistics Analysis Tool
- Erlang Module **dbg** [page 56] – The Text Based Trace Facility
- Erlang Module **eprof** [page 58] – Time Profiling Tool
- Erlang Module **exref** [page 60] – Cross Reference Tool
- Erlang Module **instrument** [page 63] – Analysis and Utility Functions for Instrumentation
- Erlang Module **make** [page 66] – Functions Similar to UNIX Type Make Program.
- Erlang Module **tags** [page 68] – Generate Emacs TAGS file from Erlang source files

coast

The following functions are exported:

- `compile(Module) -> Result`
  [page 48] Compiles a module for coverage analysis.
- `compile(Module, Options) -> Result`
  [page 48] Compiles a module for coverage analysis.
- `compile_all() -> Result`
  [page 49] Prepares all Erlang source code files in a directory for coverage and call statistics analysis.
- `compile_all(Dir) -> Result`
  [page 49] Prepares all Erlang source code files in a directory for coverage and call statistics analysis.
- `compile_all(Dir, Options) -> Result`
  [page 49] Prepares all Erlang source code files in a directory for coverage and call statistics analysis.
- `run(Module, Function, ArgumentList) -> Result`
  [page 49] Executes a function in a coast-compiled module.
- `mod_calls(Modules) -> Result`
  [page 50] Lists the number of times `Modules` have been called.
- `func_calls(Modules) -> Result`
  [page 50] Lists the number of times the functions in `Modules` have been called.

- clause_calls(Modules) -> Result
  [page 51] Lists the number of times the function clauses in Modules have been called.

- mod_coverage(Modules) -> Result
  [page 52] Lists the number of covered and uncovered lines of code in Modules.

- func_coverage(Modules) -> Result
  [page 52] Lists, for each function in Modules, the number of covered and uncovered lines of code.

- clause_coverage(Modules) -> Result
  [page 53] Lists, for each function clause in Modules, the number of covered and uncovered lines of code.

- analyse_to_file(Modules) -> Result
  [page 54] Prints to file detailed coverage analysis concerning Modules.

- known_modules() -> Result
  [page 54] Lists the modules that, as known by the coast program, may be subject to analysis.

- source_files(Modules) -> Result
  [page 54] Lists the source files that the coast-compiled modules Modules originates from.

- clear(Modules) -> ok
  [page 55] Removes data, stored from previous executions, concerning Modules.

- clear_all() -> ok
  [page 55] Removes all data stored from previous executions.

- quit() -> ok
  [page 55] Stops the server controlling the collected coverage and call statistics data.

dbg

The following functions are exported:

- p(Item) -> ok | false
  [page 56] Traces messages to and from Item.

- p(Item, Flags) -> ok | false
  [page 56] Traces Item according to Flags.

- c(Mod, Fun, Args)
  [page 57] Evaluates apply(M,F,Args) with all trace flags set.

- c(Mod, Fun, Args, Flags)
  [page 57] Evaluates apply(M,F,Args) with Flags trace flags set.

- i() -> ok
  [page 57] Displays information about all traced processes.

- stop() -> stopped
  [page 57] Stops the dbg server and the tracing of all processes.

- filter({Module, Fun})
  [page 57] Redirects all trace messages to Module:Fun/1.

eprof

The following functions are exported:

- start() -> {ok, Pid} | {error, {already_started, Pid}}
  [page 58] Starts the eprof server

- stop() -> stopped
  [page 58] Stops the eprof server.

- profile(Rootset, Mod, Fun, Args)
  [page 58] Profiles a process

- profile(Rootset) -> profiling | error
  [page 58] Starts profiling of processes.

- stop_profiling() -> profiling_stopped | profiling_already_stopped
  [page 58] Stops profiling.

- analyse() -> ok
  [page 59] Displays profiling results.

- total_analyse() -> ok
  [page 59] Displays the results of profiling.

- log(File) -> ok
  [page 59] Activates logging of eprof printouts.

exref

The following functions are exported:

- start()-> {ok, Pid} | {error, {already_started, Pid}}
  [page 60] Starts the exref server

- stop() -> stopped
  [page 60] Stops the exref server

- module(Module) -> true
  [page 60] Loads module(s) into the cross reference graph

- module(Module, Options) -> true
  [page 60] Loads module(s) into the cross reference graph

- directory(Directory)
  [page 61] Loads all modules in a directory into the cross reference graph.

- directory(Directory, Options)
  [page 61] Loads all modules in a directory into the cross reference graph.

- directory_module(Directory, Module)
  [page 61] Loads specific modules from a directory other than the current directory into the cross reference graph.

- directory_module(Directory, Module, Options)
  [page 61] Loads specific modules from a directory other than the current directory into the cross reference graph.

- delete_module(Module)
  [page 61] Deletes module(s) from the cross reference graph.

- excludes(Modules)
  [page 61] Specifies modules which should not be loaded into the cross reference graph.
- includes(Dirs)
  [page 61] Specifies directories where Erlang include files should be searched for.
- defs(Defs)
  [page 61] Adds definitions to be used when processing source code.
- analyse(Type [,Arg]) -> Result
  [page 61] Performs various analysis based on the cross reference graph
- pretty(AnalyseResult) -> ok
  [page 62] Pretty prints the AnalyseResult

instrument

The following functions are exported:

- holes(AllocList) -> ok
  [page 64] Prints out the sizes of unused memory blocks
- mem_limits(AllocList) -> {Low, High}
  [page 64] Returns lowest and highest memory address used
- memory_data() -> AllocList
  [page 64] Returns current memory allocation list
- read_memory_data(File) -> {ok, AllocList} | {error, Reason}
  [page 64] Reads memory allocation list
- sort(AllocList) -> AllocList
  [page 64] Sorts a memory allocation list
- store_memory_data(File) -> ok
  [page 64] Stores the current memory allocation list on a file
- sum_blocks(AllocList) -> int()
  [page 65] Returns the total amount of memory used
- type_string(Type) -> string()
  [page 65] Translates a memory block type number to a string

make

The following functions are exported:

- all() -> up_to_date | error
  [page 66] Compiles all files in a directory
- all(Options) -> up_to_date | error
  [page 66] Compiles all files in a directory (with Options)
- files(FileList) -> up_to_date
  [page 66] Compiles the files in FileList
- files(FileList, Options) -> up_to_date | error
  [page 66] Compiles the files in FileList using Options

tags

The following functions are exported:

- `file(File [, Options])`
  [page 68] Create a `TAGS` file for the file `File`.

- `files(FileList [, Options])`
  [page 68] Create a TAGS file for the files in the list `FileList`.

- `dir(Dir [, Options])`
  [page 68] Create a TAGS file for all files in directory `Dir`.

- `dirs(DirList [, Options])`
  [page 68] Create a TAGS file for all files in any directory in `DirList`.

- `subdir(Dir [, Options])`
  [page 68] Descend recursively down the directory `Dir` and create a `TAGS` file based on all files found.

- `subdirs(DirList [, Options])`
  [page 68] Descend recursively down all the directories in `DirList` and create a `TAGS` file based on all files found.

- `root([Options])`
  [page 68] Create a `TAGS` file covering all files in the Erlang distribution.

coast (Module)

The module `coast` provides a set of functions for coverage and call statistics analysis of Erlang programs.

Coverage analysis consists of monitoring executing programs, observing if each line of code is executed, and, if so, the number of times.

Call statistics analysis consists of monitoring executing programs, observing the number of times certain modules, and the functions in them, are called. This analysis may be done in various levels of detail.

Before any analysis can take place, the module(s) must be compiled in a special way. Execution may then take place - in this phase executional data is gathered: in an internal database, a counter for each statement in the module(s) is incremented every time that particular statement is executed. In the final phase we analyse the collected data, presenting it in various ways. (The observant reader may here correctly conclude that a module has to be executed, at least partially, before any useful coverage and call statistics analysis can take place.)

Exports

```
compile(Module) -> Result
compile(Module, Options) -> Result
```

Types:

- Module = ModuleName | [ModuleName]
- ModuleName = atom() | string()
- Options = [CompilerOptions]
- CompilerOptions = {outdir, OutDir} | {i, IncludeDir} | {d, Def} | OtherOptions
- OutDir = atom() | string()
- IncludeDir = atom() | string()
- Result = {ok, CompiledModules} | {error, Reason}
- CompiledModules = CompiledModule | [CompiledModule]
- CompiledModule = atom() | string()

Compiles a module for coverage and call statistics analysis. Currently `compile` does not search for modules - if Module not resides in the durrent working directory, the complete path has to be specified. The file extension `.erl` may be omitted.

`compile/2` makes it possible to pass several options to the compiler. Some of these options are the tuples `{i, IncludeDir}`, `{outdir, OutDir}`, and `{d, Def}`; for a complete list, please see the manual page(s) for `compile:file/2`.

The return value `Result` is one of the following:

{ok, CompiledModules} Compilation and loading succeeded, meaning that CompiledModule now is prepared for coverage and call statistics analysis.

{error,Reason} The compilation failed, due to the reason specified in Reason.

The function creates the subdirectories COAST and COAST/tmp_code in either the current directory or the directory specified using the {outdir, OutDir} option. In COAST/tmp_code two files, <File>.COAST.pretty.erl and <File>.COAST.erl, will be placed. <File>.COAST.pretty.erl is a transformed version (among other things, containing no comments) of the original file, <File>.erl. <File>.COAST.erl contains the code in <File>.COAST.pretty.erl, modified with the counter code necessary to gather coverage and call statistics data during execution.

In either the current working directory or in the directory specified using the {outdir, OutDir} option, the file <File>.jam will be placed. This file is the the compiled version of <File>.COAST.erl.

Note: <File>.COAST.pretty.erl and <File>.COAST.erl shall never ever be renamed or moved, or the coverage and call statistics analysis will fail!

Example:

```
1> coast:compile(test).
{ok,test}
2> coast:compile("../can", [{outdir, "../ebin"}]).
{ok,"../can"}
```

compile_all() -> Result
compile_all(Dir) -> Result
compile_all(Dir, Options) -> Result

Types:

- Dir = atom() | string()
- Options = [CompilerOptions]
- CompilerOptions = {outdir, OutDir} | {i, IncludeDir} | {d, Def} | OtherOptions
- OutDir = atom() | string()
- IncludeDir = atom() | string()
- Result = {ok, CompiledModules} | {error, Reason}
- CompiledModules = [CompiledModule]
- CompiledModule = atom() | string()

compile_all/0 evaluates compile/1 for all .erl files found in the current working directory Dir.

compile_all/1 evaluates compile/1 for all .erl files found in the directory Dir.

compile_all/2 evaluates compile/2 for all .erl files found in the directory Dir.

Example:

```
3> coast:compile_all().
{ok,["can","test"]}
```

run(Module, Function, ArgumentList) -> Result

Types:

- Module = atom()

- Function = atom()
- ArgumentList = [Args]
- Result = term()

`run/3` applies (the presumably coast-compiled) `Function` in `Module` on `ArgumentList`. The function in question must have been exported from `Module`. The length of the `ArgumentList` is the arity of the function.

A function in a coast-compiled module cannot be started from the shell directly, it has to be started either using the function `run/3` or from another process than the shell.

Example:

```
4> coast:run(can,start,[10]).
<0.91.0>
```

`mod_calls(Modules) -> Result`

Types:

- Modules = ModuleName | [ModuleName]
- ModuleName = atom()
- Result = {module_calls, ModuleResults} | {error, Reason}
- ModuleResults = [ModuleResult]
- ModuleResult = {ModuleName, TotalCalls, ExternalCalls, InternalNonRecursiveCalls}
- TotalCalls = ExternalCalls = InternalNonRecursiveCalls = integer()
- Reason = {not_coast_compiled, ModuleName}

`mod_calls/1` lists the number of times `Modules` have been called. The listing is presented module by module, with the following data:

`TotalCalls` The total number of times the module (i.e., the functions in the module) has been called. This number is the sum of both internal calls and calls made from other modules.

`ExternalCalls` The number of times the module (i.e., the functions in the module) has been called from other modules.

`InternalNonRecursiveCalls` The number of times the module has been called non-recursively by itself. Here a recursive module call is defined as when the module (i.e., a function in the module) calls itself (i.e., the same or another function in the module). Using this definition, it follows that a module can never call itself non-recursively, meaning that `InternalNonRecursiveCalls` always equals to 0 (zero). (The reason for nevertheless presenting it is to produce results having the same format from the `mod_calls/1`, `func_calls/1` and `clause_calls/1` functions.)

Example:

```
5> coast:mod_calls(can).
{module_calls,[{can,37,3,0}]}
6> coast:mod_calls([can,test]).
{module_calls,[{can,37,3,0},{test,0,0,0}]}
```

`func_calls(Modules) -> Result`

Types:

- Modules = ModuleName | [ModuleName]
- ModuleName = atom()
- Result = {function_calls, FunctionResults} | {error, Reason}
- FunctionResults = [FunctionResult]
- FunctionResult = {Function, TotalCalls, ExternalCalls, InternalNonRecursiveCalls}
- Function = {ModuleName, FunctionName, Arity}
- FunctionName = atom()
- Arity = integer()
- TotalCalls = ExternalCalls = InternalNonRecursiveCalls = integer()
- Reason = {not_coast_compiled, ModuleName}

`func_calls/1` lists the number of times the functions in `Modules` have been called. The listing is presented in order, module by module and function by function, with the following data:

`TotalCalls` The total number of times the function in question has been called. This number is the sum of both internal calls (i.e., calls made from the same module) and calls made from other modules.

`ExternalCalls` The number of times the function in question has been called from other modules.

`InternalNonRecursiveCalls` The number of times the function in question has been called non-recursively from the same module (i.e., by other functions in the same module).

Example:

```
7> coast:func_calls(can).
{function_calls,[{{can,create_rects,2},1,0,1},
                 {{can,create_rects,3},11,0,1},
                 {{can,event_loop,2},20,0,1},
                 {{can,f,1},1,0,1},
                 {{can,mk_canvas,1},1,1,0},
                 {{can,prov,1},2,1,1},
                 {{can,prov2,1},0,0,0},
                 {{can,start,1},1,1,0}]}
```

`clause_calls(Modules) -> Result`

Types:
- Modules = ModuleName | [ModuleName]
- ModuleName = atom()
- Result = {clause_calls, ClauseResults} | {error, Reason}
- ClauseResults = [ClauseResult]
- ClauseResult = {Clause, TotalCalls, ExternalCalls, InternalNonRecursiveCalls}
- Clause = {ModuleName, FunctionName, Arity, ClauseNumber}
- FunctionName = atom()
- Arity = ClauseNumber = integer()
- TotalCalls = ExternalCalls = InternalNonRecursiveCalls = integer()
- Reason = {not_coast_compiled, ModuleName}

clause_calls/1 lists the number of times the function clauses in `Modules` have been called. The listing is presented in order, module by module, function by function and clause by clause. To distinguish between clauses in a function, they are numbered sequentially, the first clause encountered getting number 1 (one). For each clause the following data is presented:

`TotalCalls` The total number of times the function clause in question has been called. This number is the sum of both internal calls (i.e., calls made from the same module) and calls made from other modules.

`ExternalCalls` The number of times the function clause in question has been called from other modules.

`InternalNonRecursiveCalls` The number of times the function clause in question has been called non-recursively from the same module (i.e., by other functions in the same module). (Please note that a call from another clause `in the same function` also is a recursive call!)

Example:

```
8> coast:clause_calls(can).
{clause_calls,[{{can,create_rects,2,1},1,0,1},
               {{can,create_rects,3,1},10,0,1},
               {{can,create_rects,3,2},1,0,0},
               {{can,event_loop,2,1},20,0,1},
               {{can,f,1,1},1,0,1},
               {{can,mk_canvas,1,1},1,1,0},
               {{can,prov,1,1},1,0,1},
               {{can,prov,1,2},1,1,0},
               {{can,prov2,1,1},0,0,0},
               {{can,prov2,1,2},0,0,0},
               {{can,start,1,1},1,1,0}]}
```

mod_coverage(Modules) -> Result

Types:

- Modules = ModuleName | [ModuleName]
- ModuleName = atom()
- Result = {module_coverage, ModuleResults} | {error, Reason}
- ModuleResults = [ModuleResult]
- ModuleResult = {ModuleName, Covered, Uncovered}
- Covered = Uncovered = integer()
- Reason = {not_coast_compiled, ModuleName}

mod_coverage/1 lists the number of covered and uncovered lines of code in `Modules`. The listing is presented module by module.

Example:

```
9> coast:mod_coverage(can).
{module_coverage,[{can,22,4}]}
```

func_coverage(Modules) -> Result

Types:

- Modules = ModuleName | [ModuleName]

- ModuleName = atom()
- Result = {function_coverage, FunctionResults} | {error, Reason}
- FunctionResults = [FunctionResult]
- FunctionResult = {Function, Covered, Uncovered}
- Function = {ModuleName, FunctionName, Arity}
- FunctionName = atom()
- Arity = integer()
- Covered = Uncovered = integer()
- Reason = {not_coast_compiled, ModuleName}

func_coverage/1 lists, for each function in Modules, the number of covered and uncovered lines of code.

Example:

```
10> coast:func_coverage(can).
{function_coverage,[{{can,create_rects,2},1,0},
                    {{can,create_rects,3},5,0},
                    {{can,event_loop,2},5,2},
                    {{can,f,1},1,0},
                    {{can,mk_canvas,1},6,0},
                    {{can,prov,1},3,0},
                    {{can,prov2,1},0,2},
                    {{can,start,1},1,0}]}
```

clause_coverage(Modules) -> Result

Types:

- Modules = ModuleName | [ModuleName]
- ModuleName = atom()
- Result = {clause_coverage, ClauseResults} | {error, Reason}
- ClauseResults = [ClauseResult]
- ClauseResult = {Clause, Covered, Uncovered}
- Clause = {ModuleName, FunctionName, Arity, ClauseNumber}
- FunctionName = atom()
- Arity = integer()
- ClauseNumber = integer()
- Covered = Uncovered = integer()
- Reason = {not_coast_compiled, ModuleName}

clause_coverage/1 lists, for each function clause in Modules, the number of covered and uncovered lines of code. To distinguish between clauses in a function, they are numbered sequentially, the first clause encountered getting number 1 (one).

Example:

```
11> coast:clause_coverage(can).
{clause_coverage,[{{can,create_rects,2,1},1,0},
                  {{can,create_rects,3,1},4,0},
                  {{can,create_rects,3,2},1,0},
                  {{can,event_loop,2,1},5,2},
                  {{can,f,1,1},1,0},
                  {{can,mk_canvas,1,1},6,0},
                  {{can,prov,1,1},1,0},
```

```
                              {{can,prov,1,2},2,0},
                              {{can,prov2,1,1},0,1},
                              {{can,prov2,1,2},0,1},
                              {{can,start,1,1},1,0}]}
```

analyse_to_file(Modules) -> Result

> Types:
>
> - Modules = ModuleName | [ModuleName]
> - ModuleName = atom()
> - Result = {ok, Files} | {error, Reason}
> - Files = [FileName]
> - FileName = string()
> - Reason = {not_coast_compiled, ModuleName} | OtherReason
>
> analyse_to_file/1 performs a detailed coverage analysis, showing the number of times each line in Modules has been called so far. The result is stored in FileNames (one file for each module).
>
> Example:
>
> ```
> 12> coast:analyse_to_file([can,test]).
> {ok,["/clearcase/otp/tools/devtools/tools/ebin/COAST/can.COAST.out",
>     "/clearcase/otp/tools/devtools/tools/ebin/COAST/test.COAST.out"]}
> ```

known_modules() -> Result

> Types:
>
> - Result = [ModuleName]
> - ModuleName = atom()
>
> known_modules/0 lists the modules that the coast program is aware of, i.e., the coast-compiled modules that so far, during this session working with coast, have been coast-compiled or subject to execution. The absence of a module in the list probably means it has never been coast-compiled. The presence of an unexpected module in the list probably means that an old coast-compiled module has been executed.
>
> Example:
>
> ```
> 13> coast:known_modules().
> [can,test]
> ```

source_files(Modules) -> Result

> Types:
>
> - Modules = ModuleName | [ModuleName]
> - ModuleName = atom()
> - Result = [ModuleResult]
> - ModuleResult = FileName | {error, Reason}
> - FileName = string()
> - Reason = {no_such_module, ModuleName} | {not_coast_compiled, ModuleName} | OtherReason

source_files/1 lists the source files that the coast-compiled modules specified in Modules originates from.

Result is a list containing (for each module in Modules) either the corresponding source file found, or an error.

This function is useful if one wants to make sure that the correct module actually is the one being subject to coverage and call statistics analysis.

Example:

```
14> coast:source_files(coast:known_modules()).
["/clearcase/otp/tools/devtools/tools/ebin/can.erl",
 "/clearcase/otp/tools/devtools/tools/ebin/test.erl"]
15> c(test).
{ok,test}
16> coast:source_files([can,xxx,test]).
["/clearcase/otp/tools/devtools/tools/ebin/can.erl",
 {error,{no_such_module,xxx}},
 {error,{not_coast_compiled,test}}]
```

clear(Modules) -> ok

Types:

- Modules = ModuleName | [ModuleName]
- ModuleName = atom()

clear/1 discards all coverage and call statistics data, concerning one or more modules, that has been stored (in the internal database) up to the present. (Trying to analyse any of the modules cleared will then yield the same result as when they still not have been subject to any execution.)

clear_all() -> ok

clear_all/0 discards all coverage and call statistics data that has been stored (in the internal database) up to the present.

quit() -> ok

quit/0 stops the server controlling the collected coverage and call statistics data.

Note

This module will in due time replace the cover module.

dbg (Module)

This module implements a text based interface to the `trace/3` BIF. It makes it possible to trace functions, processes, and messages on text based terminals. It can be used instead of, or as complement to, the `pman` module.

Exports

`p(Item) -> ok | false`

>   Equivalent to `p(Item, [m])`.

`p(Item, Flags) -> ok | false`

>   Traces `Item` in accordance with the value specified by `Flags`. `Item` can vary.
>
>   - If it is a Pid, the corresponding process is traced.
>   - If it is an atom, the process with the corresponding registered name is traced.
>   - If it is an integer, the process <`Item.1`> is traced.
>   - If it is a tuple {`X, Y, Z`}, the process <`X.Y.Z`> is traced.
>
>   `Flags` can be a single atom, or a list of flags. The available flags are:
>
>   s (`send`) Traces the messages the process sends.
>
>   r (`receive`) Traces the messages the process receives.
>
>   m (`messages`) Traces the messages the process receives and sends.
>
>   c (`call`) Traces function calls for the process. `Note:` This only works for modules which have been trace compiled (see c(3)).
>
>   b (`bifs`) Traces calls to BIFs.
>
>   p (`proc`) Traces process related events to the process.
>
>   sos (`set on spawn`) Lets all processes created by the traced process inherit the trace flags of the traced process.
>
>   sol (`set on link`) Lets another process, `P2`, inherit the trace flags of the traced process whenever the traced process links to `P2`.
>
>   sofs (`set on first spawn`) This is the same as `sos`, but only for the first process spawned by the traced process.
>
>   sofl (`set on first link`) This is the same as `sol`, but only for the first call to `link/1` by the traced process.
>
>   all Sets all flags.

`clear` Clears all flags.

`c(Mod, Fun, Args)`

>Equivalent to `c(Mod, Fun, Args, all)`.

`c(Mod, Fun, Args, Flags)`

>Evaluates the expression `apply(Mod, Fun, Args)` with the trace flags in `Flags` set. This is a convenient way to trace processes from the Erlang shell.

`i() -> ok`

>Displays information about all traced processes.

`stop() -> stopped`

>Stops the `dbg` server and clears all trace flags for all processes.

`filter({Module, Fun})`

>Displays all traced messages to the `dbg` server, using the function {`Module, Fun`}. The default function merely displays the messages on the screen by means of `io:format/2`. Can be used to write the traced output to a file, or to send the output to a remote node.

eprof (Module)

This module is used to profile a system in order to find out how time is used. It is usually very difficult to measure time in a large system. The `eprof` tool was designed to overcome this.

The modules that will be included in the profiling must be compiled with the trace flag. The trace flag itself makes the system slightly slower, but the code in the part of the system which is profiled only runs at approximately 20 % of its original speed.

Exports

`start() -> {ok, Pid} | {error, {already_started, Pid}}`

`stop() -> stopped`

`profile(Rootset, Mod, Fun, Args)`

> This function evaluates the expression `spawn(Mod, Fun, Args)` and profiles the process which evaluates it. The profiling is done for one function with a set of arguments in a certain root set of processes. All processes which are created by that function are profiled, together with its root set and all processes which are created by processes in that root set.
>
> The profiling is terminated when the given function returns a value. The application must check that the function is truly synchronized and that no work continues after the function has returned a value.
>
> For example, if a portion of code with a large amount of file I/O is to be profiled, the Erlang module `file` should be compiled with the trace flag and `file` should then be included in the root set.
>
> The root set is a list of Pids or atoms. If atoms, they are assumed to be registered processes.

`profile(Rootset) -> profiling | error`

> Sometimes, it is not possible to start profiling with the aid of a single function.
>
> For example, if some external stimuli enters the Erlang system through a port, and the handling of this stimuli is to be profiled until a response goes out through a port, it may be appropriate to change the source code and insert an explicit call to this function. The `profile(Rootset)` function starts the profiling for processes included in `Rootset`.

`stop_profiling() -> profiling_stopped | profiling_already_stopped`

This function stops the collection of statistics performed by the `eprof` process. The `eprof` process then holds data which can be analysed at a later stage.

`analyse() -> ok`

When the profiling has ended - profiling using `profile/4`, or `profile/1` together with `stop_profiling/0` - the `eprof` process can print the data collected during the run. The printed profiling statistics show the activity for each process.

`total_analyse() -> ok`

With this function, the total results of profiling is printed irrespective which process each function has used.

`log(File) -> ok`

This function ensures that a copy of all printouts are sent to both `File` and the screen.

Notes

The actual supervision of execution times is in itself a CPU intensive activity. A message is sent to the `eprof` process for every function call that is made by the profiled code. This can only be done for trace compiled code and timings will be incorrect if parts of the code to be profiled uses non-trace compiled code. It is therefore strongly recommended that all code, including all library code, is compiled with the trace flag.

SEE ALSO

compile(3)

exref (Module)

The `exref` tool is an incremental cross reference server which builds a cross reference graph for all modules loaded into it. Information which can be derived from the cross reference graph includes use graphs and module dependencies. The call graph is represented as a directed graph (see digraph(3)). A function vertex is represented as:

    {{Mod, Fun, Arity}, {Type, File, Line}}

In this code:

    Type = local | exported | {exported, compiler} |
           {local, compiler} | {void(), record}

A call edge is represented as:

    {EdgeId, {Mod1, Fun1, Arity1}, {Mod2, Fun2, Arity2}, Line}.

Exports

start()-> {ok, Pid} | {error, {already_started, Pid}}

> Starts the `exref` server. The `exref` server must be started before any other functions in module `exref` can be used.

stop() -> stopped

> Stops the `exref` server.

module(Module) -> true

> This is a short form for calling `module(Module,[search,verbose])` (see below).

module(Module, Options) -> true

> Types:
> - Module = atom() | [atom()]
> - Options = [Option]
> - Option = search | verbose | auto | warnings | recursive | no_libs
>
> Loads the module `Module` into the cross reference graph. `Module` can also be a list of modules. `Options` is a list with the following possible options:

search Searches for source file in code path and replaces the path `X/ebin` with the path `X/src`.

verbose Creates an output of module names during loading.

auto Automatically loads all referenced modules into the cross reference graph, with the exception of modules specified with the `excludes/1` function. See also the `no_libs` option.

recursive Recursively includes all files in a directory. See also the `no_libs` option.

warnings Emits warnings about the application and the spawning of variables. The reason for this is that `apply`, with variable modules or functions, leads to an incomplete call graph. The `apply` call will be inserted into the call graph instead of the actual call. The same applies to spawn.

no_libs Used together with the options `auto` and `recursive`, this options prevents modules in the standard libraries from being loaded into the cross reference graph.

`directory(Directory)`
`directory(Directory, Options)`

Loads all modules found in the directory `Directory` into the cross reference graph. Options are the same as for `module/2`. The function `directory/1` is equivalent to `directory(Directory,[verbose])`.

`directory_module(Directory, Module)`
`directory_module(Directory, Module, Options)`

Loads the module `Module` located in the directory `Directory`. `Module` can also be a list of modules. Options are the same as for `module/2`. The function `directory_module/2` is equivalent to `directory_module(Directory,Module,[verbose])`.

`delete_module(Module)`

Deletes the module `Module` from the cross reference graph. `Module` can also be a list of modules.

`excludes(Modules)`

Appends the modules of the `Modules` list to the list of modules which are excluded from the cross reference graph.

`includes(Dirs)`

Appends the directories of the `Dirs` list to the include search path for Erlang include files (see epp(3)).

`defs(Defs)`

Appends the definitions in the `Defs` list to the definition list used by `epp` (see epp(3)).

`analyse(Type [,Arg]) -> Result`

Performs various analyses of the cross reference graph and returns an Erlang term with a format that depends on the `Type` of analyse. Some analyse types can have an optional argument `Arg`. The result can be formatted to a textual printout with `pretty/1`. The available `Type` and `Arg` combinations are:

call Emits the calls from the functions, for all functions in the graph.

call, Module Emits the calls for all functions in the module Module,

call, Function Emits the calls from the function Function, which has the format {Mod, Fun, Arity}.

use Emits the use of functions, for all functions in the graph.

use, Module Emits the use of functions, for all functions of the module Module.

use, Function Emits the use of the function Function, which has the format {Mod, Fun, Arity}.

module_call Emits the module dependency graph. For example, if module M1 has calls to M2, this analysis emits M1: M2 ...

module_use Emits a module graph which is the reverse of the module dependency graph. For example, if module M1 is called by modules M2 and M3, the analysis emits M1: M2 M3.

exports_not_called Reports all exported functions which are not used. This means that all entry points to a program can be found, also exported functions that should be local.

locals_not_called Reports all local functions which are used. These functions can be removed without the program being affected.

undefined_functions Reports all function calls which are calls to functions outside the cross reference graph. The library functions and Erlang BIFs are never considered undefined.

recursive_modules Reports modules that are (partially) recursively defined, which means that they contain function calls outside the module which in turn call the functions in that module.

user_defined, {Mod, Fun} Calls user-defined analysis. The reason for user-defined analysis being attached in this way is that the call graph cannot easily be copied to other processes. It should be performed within the exref server process.
The function definition must be as follows for user supplied analysis:

```
my_analysis(G) ->
    io:format("MY ANALYSIS ... ~n", Args),
    ...
```

G is the cross reference graph as described above. The return value from a user-defined analysis is ignored.

pretty(AnalyseResult) -> ok

This function pretty-prints a verbose textual representation of AnalyseResult which must be the output from analyse(Type[,Arg]). The result from a user-defined analysis cannot be used as input to this function.

See Also

digraph(3)

instrument (Module)

The module `instrument` contains support for studying the resource usage in an Erlang system. Currently, only the allocation of memory can be studied.

> **Note:**
> Note that this whole module is experimental, and the representations used as well as the functionality is likely to change in the future.

Some of the functions in this module are only available in Erlang compiled with instrumentation; otherwise they exit with `badarg`. This is noted below for the individual functions. To start an Erlang system with instrumentation, use the command-line option `-instr` to the `erl` command.

The basic object of study in the case of memory allocation is a memory allocation list, which contains one descriptor for each allocated memory block. Currently, a descriptor is a 3-tuple of integers

        {Type, Address, Size}

where `Type` indicates what the block is used for, `Address` is its place in memory, and `Size` is its size, in bytes.

Various details about memory allocation:

On Unix (for example, Solaris), memory for a process is allocated linearly, usually from 0. The current size of the process cannot be obtained from within Erlang, but can be seen with one of the system statistics tools, e.g., `ps` or `top`. (There may be a hole above the highest used memory block; in that case the functions in the `instrument` module can't tell you about it; you have to compare the `High` value from `mem_limits/1` with the value which the system reports for Erlang.)

In the memory allocation list, certain small objects don't show up individually, since they are allocated from blocks of 20 objects (called "fixalloc" blocks). The blocks themselves do show up, but the amount of internal fragmentation in them currently can't be observed.

Overhead for instrumentation: instrumented memory allocation uses 24 bytes extra for each block. The time overhead for managing the list is negligible.

Exports

`holes(AllocList) -> ok`

>   Types:
>
>   • AllocList = [Desc]
>   • Desc = {int(), int(), int()}
>
>   Prints out the size of each hole (i.e., the space between allocated blocks) on the
>   terminal. The list must be sorted (see `sort/1`).

`mem_limits(AllocList) -> {Low, High}`

>   Types:
>
>   • AllocList = [Desc]
>   • Desc = {int(), int(), int()}
>   • Low = High = int()
>
>   returns a tuple {`Low, High`} indicating the lowest and highest address used. The list
>   must be sorted (see `sort/1`).

`memory_data() -> AllocList`

>   Types:
>
>   • AllocList = [Desc]
>   • Desc = {int(), int(), int()}
>
>   Returns the memory allocation list. Only available in an Erlang system compiled for
>   instrumentation. Blocks execution of other processes while the list is collected.

`read_memory_data(File) -> {ok, AllocList} | {error, Reason}`

>   Types:
>
>   • File = string()
>   • AllocList = [Desc]
>   • Desc = {int(), int(), int()}
>
>   Reads a memory allocation list from the file `File`. The file is assumed to have been
>   created by `store_memory_data/1`. The error codes are the same as for `file:consult/1`.

`sort(AllocList) -> AllocList`

>   Types:
>
>   • AllocList = [Desc]
>   • Desc = {int(), int(), int()}
>
>   Sorts a memory allocation list so the addresses are in ascending order. The list
>   arguments to many of the functions in this module must be sorted. No other function in
>   this module returns a sorted list.

`store_memory_data(File) -> ok`

>   Types:

- File = string()

Stores the memory allocation list on the file `File`. The contents of the file can later be read using `read_memory_data/1`. Only available in an Erlang system compiled for instrumentation. Blocks execution of other processes while the list is collected (the time to write the data is around 0.1 ms/line on a Sun Ultra 1).

Failure: `badarg` if the file couldn't be written.

`sum_blocks(AllocList) -> int()`

Types:

- AllocList = [Desc]
- Desc = {int(), int(), int()}

Returns the total size of the memory blocks in the list. The list must be sorted (see `sort/1`).

`type_string(Type) -> string()`

Types:

- Type = int()

Translates a memory block type number into a readable string, which is a short description of the block type.

Failure: `badarg` if the argument is not a valid block type number.

make (Module)

These functions are similar to the UNIX type `Make` functions. They can be used to develop programs which consist of several files. `make` can also be used to recompile entire directories. If updates are made, `make` exits with the value `up_to_date`.

Exports

`all() -> up_to_date | error`

> This function is the same as `all([])`.

`all(Options) -> up_to_date | error`

> Checks all Erlang files in the current directory and compiles those files which have been modified after the creation of the object file. `Options` is a list of valid options for `make`, together with valid options for `compile`.
>
> Compares time stamps of `.erl` and object code files. If the time stamp of the source file is later than the object file, or the object file is missing, the source file is recompiled.
>
> The list of files to be compared is taken from the file `Emakefile`, if it exists. Failing this, it is taken from the current directory.
>
> This function returns `error` if compilation fails for any file.
>
> The elements of `Options` can be:
>
> `noexec` No execution mode. It just specifies that the files should be compiled.
>
> `load` Load mode. Loads all recompiled files.
>
> `netload` Net load mode. Loads all recompiled files on the compiling node, and all other nodes in the network, with `net:broadcast/3`,
>
> `par` `make` is used in parallel on all nodes included in the expression (`node ()` | `nodes ()`)
>
> For example:
>
>> `1> make:all ([par, netload, trace]).`
>
> `make` is used in parallel on all nodes. This ensures that the produced object files are loaded on all nodes and the `trace` flag is passed to the compiler. This produces traceable code.

`files(FileList) -> up_to_date`

`files(FileList, Options) -> up_to_date | error`

This is the same as `all/0` and `all/1`, but with an explicit list of files.

This function returns `error` if compilation fails for any file or if a non-existing file is specified.

Files

This program assumes that a file named `Emakefile` exists and that it is located in the current directory. The file must be named `Emakefile` and it must contain the names of the files concerned as atoms, each followed by a period. For example:

```
file1.
file2.
'../foo/file3'.
'File4'.
         ^ (a new line )
```

If the `Emakefile` does not exist, all Erlang files in the current directory are used as input. This is useful when recompiling entire directories.

tags (Module)

A `TAGS` file is used by Emacs to find function and variable definitions in any source file in large projects. This module can generate a `TAGS` file from Erlang source files. It recognises functions, records, and macro definitions.

Exports

`file(File [, Options])`

> Create a `TAGS` file for the file `File`.

`files(FileList [, Options])`

> Create a TAGS file for the files in the list `FileList`.

`dir(Dir [, Options])`

> Create a TAGS file for all files in directory `Dir`.

`dirs(DirList [, Options])`

> Create a TAGS file for all files in any directory in `DirList`.

`subdir(Dir [, Options])`

> Descend recursively down the directory `Dir` and create a `TAGS` file based on all files found.

`subdirs(DirList [, Options])`

> Descend recursively down all the directories in `DirList` and create a `TAGS` file based on all files found.

`root([Options])`

> Create a `TAGS` file covering all files in the Erlang distribution.

OPTIONS

The functions above have an optional argument, `Options`. It is a list which can contain the following elements:

- {`outfile, NameOfTAGSFile`} Create a TAGS file named `NameOfTAGSFile`.
- {`outdir, NameOfDirectory`} Create a file named `TAGS` in the directory `NameOfDirectory`.

The default behaviour is to create a file named `TAGS` in the current directory.

Examples

- `tags:root([{outfile, "root.TAGS"}]).`
  This command will create a file named `root.TAGS` in the current directory. The file will contain references to all Erlang source files in the Erlang distribution.
- `tags:files(["foo.erl", "bar.erl", "baz.erl"], [{outdir, "../projectdir"}]).`
  Here we create file named `TAGS` placed it in the directory `../projectdir`. The file contains information about the functions, records, and macro definitions of the three files.

SEE ALSO

- Richard M. Stallman. GNU Emacs Manual, chapter "Editing Programs", section "Tag Tables". Free Software Foundation, 1995.
- Anders Lindgren. The Erlang editing mode for Emacs. Ericsson, 1998.

List of Figures

List of Tables

Index

Modules are typed in *this way*.
Functions are typed in `this way`.