

Syntax_Tools

Copyright © 2006-2017 Ericsson AB. All Rights Reserved.
Syntax_Tools 1.7
December 1, 2017

Copyright © 2006-2017 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

December 1, 2017

1 Syntax_Tools User's Guide

Syntax_Tools contains modules for handling abstract Erlang syntax trees, in a way that is compatible with the "parse trees" of the STDLIB module `erl_parse`, together with utilities for reading source files in unusual ways and pretty-printing syntax trees. Also included is an amazing module merger and renamer called Igor, as well as an automatic code-cleaner.

1.1 Erlang Syntax and Metaprogramming tools

1.1.1 Overview

This package contains modules for handling abstract syntax trees (ASTs) in Erlang, in a way that is compatible with the "abstract format" parse trees of the stdlib module `erl_parse`, together with utilities for reading source files, *pretty-printing syntax trees*, *merging and renaming modules*, *cleaning up obsolete constructs*, and doing *metaprogramming* in Erlang.

The abstract layer (defined in *erl_syntax*) is nicely structured and the node types are context-independent. The layer makes it possible to transparently attach source-code comments and user annotations to nodes of the tree. Using the abstract layer makes applications less sensitive to changes in the *erl_parse(3)* data structures, only requiring the `erl_syntax` module to be up-to-date.

The pretty printer *erl_prettypr* is implemented on top of the library module *prettypr*: this is a powerful and flexible generic pretty printing library, which is also distributed separately.

For a short demonstration of parsing and pretty-printing, simply compile the included module **demo.erl**, and execute `demo:run()` from the Erlang shell. It will compile the remaining modules and give you further instructions.

Also try the *erl_tidy* module, as follows:

```
erl_tidy:dir("any-erlang-source-dir", [test, old_guard_tests]).
```

(the `test` option assures that no files are modified).

2 Reference Manual

Syntax_Tools contains modules for handling abstract Erlang syntax trees, in a way that is compatible with the "external format" parse trees of the STDLIB module `erl_parse`, together with utilities for reading source files, pretty-printing syntax trees, merging and renaming modules, cleaning up obsolete constructs, and doing metaprogramming in Erlang.

epp_dodger

Erlang module

`epp_dodger` - bypasses the Erlang preprocessor.

This module tokenises and parses most Erlang source code without expanding preprocessor directives and macro applications, as long as these are syntactically "well-behaved". Because the normal parse trees of the `erl_parse` module cannot represent these things (normally, they are expanded by the Erlang preprocessor *epp(3)* before the parser sees them), an extended syntax tree is created, using the *erl_syntax* module.

DATA TYPES

`errorinfo() = {ErrorLine::integer(), Module::atom(), Descriptor::term()}`

This is a so-called Erlang I/O ErrorInfo structure; see the *io(3)* module for details.

Exports

`parse(Dev::IODevice) -> {ok, Forms} | {error, errorinfo()}`

Equivalent to *parse(IODevice, 1)*.

`parse(Dev::IODevice, L::StartLine) -> {ok, Forms} | {error, errorinfo()}`

Types:

`IODevice = pid()`
`StartLine = integer()`
`Forms = [syntaxTree() (see module erl_syntax)]`

Equivalent to *parse(IODevice, StartLine, [])*.

See also: parse/1.

`parse(Dev::IODevice, L0::StartLine, Options) -> {ok, Forms} | {error, errorinfo()}`

Types:

`IODevice = pid()`
`StartLine = integer()`
`Options = [term()]`
`Forms = [syntaxTree() (see module erl_syntax)]`

Reads and parses program text from an I/O stream. Characters are read from `IODevice` until end-of-file; apart from this, the behaviour is the same as for *parse_file/2*. `StartLine` is the initial line number, which should be a positive integer.

See also: parse/2, *parse_file/2*, *parse_form/2*, *quick_parse/3*.

`parse_file(File) -> {ok, Forms} | {error, errorinfo()}`

Types:

`File = filename() (see module file)`
`Forms = [syntaxTree() (see module erl_syntax)]`

Equivalent to *parse_file(File, [])*.

parse_file(File, Options) -> {ok, Forms} | {error, errorinfo()}

Types:

File = filename() (see module file)
Options = [term()]
Forms = [syntaxTree() (see module erl_syntax)]

Reads and parses a file. If successful, `{ok, Forms}` is returned, where `Forms` is a list of abstract syntax trees representing the "program forms" of the file (cf. `erl_syntax:is_form/1`). Otherwise, `{error, errorinfo()}` is returned, typically if the file could not be opened. Note that parse errors show up as error markers in the returned list of forms; they do not cause this function to fail or return `{error, errorinfo()}`.

Options:

`{no_fail, boolean()}`

If `true`, this makes `epp_dodger` replace any program forms that could not be parsed with nodes of type `text` (see *erl_syntax:text/1*), representing the raw token sequence of the form, instead of reporting a parse error. The default value is `false`.

`{clever, boolean()}`

If set to `true`, this makes `epp_dodger` try to repair the source code as it seems fit, in certain cases where parsing would otherwise fail. Currently, it inserts `++`-operators between string literals and macros where it looks like concatenation was intended. The default value is `false`.

See also: parse/2, quick_parse_file/1, erl_syntax:is_form/1.

parse_form(Dev::IODevice, L0::StartLine) -> {ok, Form, LineNo} | {eof, LineNo} | {error, errorinfo(), LineNo}

Types:

IODevice = pid()
StartLine = integer()
Form = syntaxTree() (see module erl_syntax)
LineNo = integer()

Equivalent to *parse_form(IODevice, StartLine, [])*.

See also: quick_parse_form/2.

parse_form(Dev::IODevice, L0::StartLine, Options) -> {ok, Form, LineNo} | {eof, LineNo} | {error, errorinfo(), LineNo}

Types:

IODevice = pid()
StartLine = integer()
Options = [term()]
Form = syntaxTree() (see module erl_syntax)
LineNo = integer()

Reads and parses a single program form from an I/O stream. Characters are read from `IODevice` until an end-of-form marker is found (a period character followed by whitespace), or until end-of-file; apart from this, the behaviour

is similar to that of `parse/3`, except that the return values also contain the final line number given that `StartLine` is the initial line number, and that `{eof, LineNo}` may be returned.

See also: parse/3, parse_form/2, quick_parse_form/3.

`quick_parse(Dev::IODevice) -> {ok, Forms} | {error, errorinfo()}`

Equivalent to *quick_parse(IODevice, 1)*.

`quick_parse(Dev::IODevice, L::StartLine) -> {ok, Forms} | {error, errorinfo()}`

Types:

`IODevice = pid()`
`StartLine = integer()`
`Forms = [syntaxTree() (see module erl_syntax)]`

Equivalent to *quick_parse(IODevice, StartLine, [])*.

See also: quick_parse/1.

`quick_parse(Dev::IODevice, L0::StartLine, Options) -> {ok, Forms} | {error, errorinfo()}`

Types:

`IODevice = pid()`
`StartLine = integer()`
`Options = [term()]`
`Forms = [syntaxTree() (see module erl_syntax)]`

Similar to *parse/3*, but does a more quick-and-dirty processing of the code. See *quick_parse_file/2* for details.

See also: parse/3, quick_parse/2, quick_parse_file/2, quick_parse_form/2.

`quick_parse_file(File) -> {ok, Forms} | {error, errorinfo()}`

Types:

`File = filename() (see module file)`
`Forms = [syntaxTree() (see module erl_syntax)]`

Equivalent to *quick_parse_file(File, [])*.

`quick_parse_file(File, Options) -> {ok, Forms} | {error, errorinfo()}`

Types:

`File = filename() (see module file)`
`Options = [term()]`
`Forms = [syntaxTree() (see module erl_syntax)]`

Similar to *parse_file/2*, but does a more quick-and-dirty processing of the code. Macro definitions and other preprocessor directives are discarded, and all macro calls are replaced with atoms. This is useful when only the main structure of the code is of interest, and not the details. Furthermore, the quick-parse method can usually handle more strange cases than the normal, more exact parsing.

Options: see *parse_file/2*. Note however that for `quick_parse_file/2`, the option `no_fail` is `true` by default.

See also: parse_file/2, quick_parse/2.

quick_parse_form(Dev::IODevice, L0::StartLine) -> {ok, Form, LineNo} | {eof, LineNo} | {error, errorinfo(), LineNo}

Types:

IODevice = pid()
StartLine = integer()
Form = syntaxTree() (see module erl_syntax) | none
LineNo = integer()

Equivalent to *quick_parse_form(IODevice, StartLine, [])*.

See also: parse_form/2.

quick_parse_form(Dev::IODevice, L0::StartLine, Options) -> {ok, Form, LineNo} | {eof, LineNo} | {error, errorinfo(), LineNo}

Types:

IODevice = pid()
StartLine = integer()
Options = [term()]
Form = syntaxTree() (see module erl_syntax)
LineNo = integer()

Similar to *parse_form/3*, but does a more quick-and-dirty processing of the code. See *quick_parse_file/2* for details.

See also: parse/3, *parse_form/3*, *quick_parse_form/2*.

tokens_to_string(Tokens::[term()]) -> string()

Generates a string corresponding to the given token sequence. The string can be re-tokenized to yield the same token list again.

erl_comment_scan

Erlang module

Functions for reading comment lines from Erlang source code.

DATA TYPES

```
comment() = {integer(), integer(), integer(), [string()]}
```

Exports

`file(FileName::filename() (see module file)) -> [Comment]`

Types:

```
Comment = {Line, Column, Indentation, Text}
Line = integer()
Column = integer()
Indentation = integer()
Text = [string()]
```

Extracts comments from an Erlang source code file. Returns a list of entries representing *multi-line* comments, listed in order of increasing line-numbers. For each entry, `Text` is a list of strings representing the consecutive comment lines in top-down order; the strings contain *all* characters following (but not including) the first comment-introducing `%` character on the line, up to (but not including) the line-terminating newline.

Furthermore, `Line` is the line number and `Column` the left column of the comment (i.e., the column of the comment-introducing `%` character). `Indent` is the indentation (or padding), measured in character positions between the last non-whitespace character before the comment (or the left margin), and the left column of the comment. `Line` and `Column` are always positive integers, and `Indentation` is a nonnegative integer.

Evaluation exits with reason `{read, Reason}` if a read error occurred, where `Reason` is an atom corresponding to a Posix error code; see the module *file(3)* for details.

`join_lines(Lines::[CommentLine]) -> [Comment]`

Types:

```
CommentLine = {Line, Column, Indent, string()}
Line = integer()
Column = integer()
Indent = integer()
Comment = {Line, Column, Indent, Text}
Text = [string()]
```

Joins individual comment lines into multi-line comments. The input is a list of entries representing individual comment lines, *in order of decreasing line-numbers*; see *scan_lines/1* for details. The result is a list of entries representing *multi-line* comments, *still listed in order of decreasing line-numbers*, but where for each entry, `Text` is a list of consecutive comment lines in order of *increasing* line-numbers (i.e., top-down).

See also: scan_lines/1.

scan_lines(Text::string()) -> [CommentLine]

Types:

```
CommentLine = {Line, Column, Indent, Text}
Line = integer()
Column = integer()
Indent = integer()
Text = string()
```

Extracts individual comment lines from a source code string. Returns a list of comment lines found in the text, listed in order of *decreasing* line-numbers, i.e., the last comment line in the input is first in the resulting list. `Text` is a single string, containing all characters following (but not including) the first comment-introducing `%` character on the line, up to (but not including) the line-terminating newline. For details on `Line`, `Column` and `Indent`, see *file/1*.

string(Text::string()) -> [Comment]

Types:

```
Comment = {Line, Column, Indentation, Text}
Line = integer()
Column = integer()
Indentation = integer()
Text = [string()]
```

Extracts comments from a string containing Erlang source code. Except for reading directly from a string, the behaviour is the same as for *file/1*.

See also: file/1.

erl_prettypr

Erlang module

Pretty printing of abstract Erlang syntax trees.

This module is a front end to the pretty-printing library module `prettypr`, for text formatting of abstract syntax trees defined by the module `erl_syntax`.

DATA TYPES

`context()`

A representation of the current context of the pretty-printer. Can be accessed in hook functions.

`hook() = (syntaxTree(), context(), Continuation) -> document()`

- `Continuation = (syntaxTree(), context()) -> document()`

A call-back function for user-controlled formatting. See *format/2*.

Exports

`best(Tree::syntaxTree()) -> empty | document()`

Equivalent to *best(Tree, [])*.

`best(Tree::syntaxTree(), Options::[term()]) -> empty | document()`

Creates a fixed "best" abstract layout for a syntax tree. This is similar to the `layout/2` function, except that here, the final layout has been selected with respect to the given options. The atom `empty` is returned if no such layout could be produced. For information on the options, see the `format/2` function.

See also: best/1, *format/2*, *layout/2*, *prettypr:best/3*.

`format(Tree::syntaxTree()) -> string()`

Equivalent to *format(Tree, [])*.

`format(Tree::syntaxTree(), Options::[term()]) -> string()`

Types:

`syntaxTree() (see module erl_syntax)`

Prettyprint-formats an abstract Erlang syntax tree as text. For example, if you have a `.beam` file that has been compiled with `debug_info`, the following should print the source code for the module (as it looks in the debug info representation):

```
{ok,{_,[{abstract_code,{_,AC}}]}} =
        beam_lib:chunks("myfile.beam",[abstract_code]),
io:put_chars(erl_prettypr:format(erl_syntax:form_list(AC)))
```

Available options:

{hook, none | *hook()*}

Unless the value is `none`, the given function is called for each node whose list of annotations is not empty; see below for details. The default value is `none`.

{paper, integer()}

Specifies the preferred maximum number of characters on any line, including indentation. The default value is 80.

{ribbon, integer()}

Specifies the preferred maximum number of characters on any line, not counting indentation. The default value is 65.

{user, term()}

User-specific data for use in hook functions. The default value is `undefined`.

{encoding, epp:source_encoding()}

Specifies the encoding of the generated file.

A hook function (cf. the *hook()* type) is passed the current syntax tree node, the context, and a continuation. The context can be examined and manipulated by functions such as `get_ctxt_user/1` and `set_ctxt_user/2`. The hook must return a "document" data structure (see *layout/2* and *best/2*); this may be constructed in part or in whole by applying the continuation function. For example, the following is a trivial hook:

```
fun (Node, Ctxt, Cont) -> Cont(Node, Ctxt) end
```

which yields the same result as if no hook was given. The following, however:

```
fun (Node, Ctxt, Cont) ->
    Doc = Cont(Node, Ctxt),
    prettypr:beside(prettypr:text("<b>"),
                    prettypr:beside(Doc,
                                    prettypr:text("</b>")))
end
```

will place the text of any annotated node (regardless of the annotation data) between HTML "boldface begin" and "boldface end" tags.

See also: erl_syntax, *best/2*, *format/1*, *get_ctxt_user/1*, *layout/2*, *set_ctxt_user/2*.

`get_ctxt_hook(Ctxt::context()) -> hook()`

Returns the hook function field of the prettyprinter context.

See also: set_ctxt_hook/2.

`get_ctxt_linewidth(Ctxt::context()) -> integer()`

Returns the line widh field of the prettyprinter context.

See also: set_ctxt_linewidth/2.

`get_ctxt_paperwidth(Ctxt::context()) -> integer()`

Returns the paper widh field of the prettyprinter context.

See also: set_ctxt_paperwidth/2.

`get_ctxt_precedence(Ctxt::context()) -> integer()`

Returns the operator precedence field of the prettyprinter context.

See also: set_ctxt_precedence/2.

get_ctxt_user(Ctxt::context()) -> term()

Returns the user data field of the prettyprinter context.

See also: set_ctxt_user/2.

layout(Tree::syntaxTree()) -> document()

Equivalent to *layout(Tree, [])*.

layout(Tree::syntaxTree(), Options::[term()]) -> document()

Types:

document() (see module prettypr)

Creates an abstract document layout for a syntax tree. The result represents a set of possible layouts (cf. module `prettypr`). For information on the options, see *format/2*; note, however, that the `paper` and `ribbon` options are ignored by this function.

This function provides a low-level interface to the pretty printer, returning a flexible representation of possible layouts, independent of the paper width eventually to be used for formatting. This can be included as part of another document and/or further processed directly by the functions in the `prettypr` module, or used in a hook function (see `format/2` for details).

See also: prettypr, format/2, layout/1.

set_ctxt_hook(Ctxt::context(), Hook::hook()) -> context()

Updates the hook function field of the prettyprinter context.

See also: get_ctxt_hook/1.

set_ctxt_linewidth(Ctxt::context(), W::integer()) -> context()

Updates the line widh field of the prettyprinter context.

Note: changing this value (and passing the resulting context to a continuation function) does not affect the normal formatting, but may affect user-defined behaviour in hook functions.

See also: get_ctxt_linewidth/1.

set_ctxt_paperwidth(Ctxt::context(), W::integer()) -> context()

Updates the paper widh field of the prettyprinter context.

Note: changing this value (and passing the resulting context to a continuation function) does not affect the normal formatting, but may affect user-defined behaviour in hook functions.

See also: get_ctxt_paperwidth/1.

set_ctxt_precedence(Ctxt::context(), Prec::integer()) -> context()

Updates the operator precedence field of the prettyprinter context. See the *erl_parse(3)* module for operator precedences.

See also: erl_parse(3), get_ctxt_precedence/1.

set_ctxt_user(Ctxt::context(), X::term()) -> context()

Updates the user data field of the prettyprinter context.

See also: get_ctxt_user/1.

erl_recomment

Erlang module

Inserting comments into abstract Erlang syntax trees

This module contains functions for inserting comments, described by position, indentation and text, as attachments on an abstract syntax tree, at the correct places.

Exports

quick_recomment_forms(Tree::Forms, Comments::[Comment]) -> syntaxTree()

Types:

```
Forms = syntaxTree() | [syntaxTree()]
Comment = {Line, Column, Indentation, Text}
Line = integer()
Column = integer()
Indentation = integer()
Text = [string()]
```

Like *recomment_forms/2*, but only inserts top-level comments. Comments within function definitions or declarations ("forms") are simply ignored.

recomment_forms(Tree::Forms, Comments::[Comment]) -> syntaxTree()

Types:

```
syntaxTree() (see module erl_syntax)
Forms = syntaxTree() | [syntaxTree()]
Comment = {Line, Column, Indentation, Text}
Line = integer()
Column = integer()
Indentation = integer()
Text = [string()]
```

Attaches comments to the syntax tree/trees representing a program. The given `Forms` should be a single syntax tree of type `form_list`, or a list of syntax trees representing "program forms". The syntax trees must contain valid position information (for details, see `recomment_tree/2`). The result is a corresponding syntax tree of type `form_list` in which all comments in the list `Comments` have been attached at the proper places.

Assuming `Forms` represents a program (or any sequence of "program forms"), any comments whose first lines are not directly associated with a specific program form will become standalone comments inserted between the neighbouring program forms. Furthermore, comments whose column position is less than or equal to one will not be attached to a program form that begins at a conflicting line number (this can happen with preprocessor-generated `line`-attributes).

If `Forms` is a syntax tree of some other type than `form_list`, the comments will be inserted directly using `recomment_tree/2`, and any comments left over from that process are added as postcomments on the result.

Entries in `Comments` represent multi-line comments. For each entry, `Line` is the line number and `Column` the left column of the comment (the column of the first comment-introducing "`%`" character). `Indentation` is the number of character positions between the last non-whitespace character before the comment (or the left margin) and the left column of the comment. `Text` is a list of strings representing the consecutive comment lines in top-down order,

where each string contains all characters following (but not including) the comment-introducing "%" and up to (but not including) the terminating newline. (Cf. module `erl_comment_scan`.)

Evaluation exits with reason `{bad_position, Pos}` if the associated position information `Pos` of some subtree in the input does not have a recognizable format, or with reason `{bad_tree, L, C}` if insertion of a comment at line `L`, column `C`, fails because the tree structure is ill-formed.

See also: erl_comment_scan, quick_recomment_forms/2, recomment_tree/2.

`recomment_tree(Tree::syntaxTree(), Comments::[Comment]) -> {syntaxTree(), [Comment]}`

Types:

```
Comment = {Line, Column, Indentation, Text}
Line = integer()
Column = integer()
Indentation = integer()
Text = [string()]
```

Attaches comments to a syntax tree. The result is a pair `{NewTree, Remainder}` where `NewTree` is the given `Tree` where comments from the list `Comments` have been attached at the proper places. `Remainder` is the list of entries in `Comments` which have not been inserted, because their line numbers are greater than those of any node in the tree. The entries in `Comments` are inserted in order; if two comments become attached to the same node, they will appear in the same order in the program text.

The nodes of the syntax tree must contain valid position information. This can be single integers, assumed to represent a line number, or 2- or 3-tuples where the first or second element is an integer, in which case the leftmost integer element is assumed to represent the line number. Line numbers less than one are ignored (usually, the default line number for newly created nodes is zero).

For details on the `Line`, `Column` and `Indentation` fields, and the behaviour in case of errors, see `recomment_forms/2`.

See also: recomment_forms/2.

erl_syntax

Erlang module

Abstract Erlang syntax trees.

This module defines an abstract data type for representing Erlang source code as syntax trees, in a way that is backwards compatible with the data structures created by the Erlang standard library parser module `erl_parse` (often referred to as "parse trees", which is a bit of a misnomer). This means that all `erl_parse` trees are valid abstract syntax trees, but the reverse is not true: abstract syntax trees can in general not be used as input to functions expecting an `erl_parse` tree. However, as long as an abstract syntax tree represents a correct Erlang program, the function *revert/1* should be able to transform it to the corresponding `erl_parse` representation.

A recommended starting point for the first-time user is the documentation of the *syntaxTree()* data type, and the function *type/1*.

NOTES:

This module deals with the composition and decomposition of *syntactic* entities (as opposed to semantic ones); its purpose is to hide all direct references to the data structures used to represent these entities. With few exceptions, the functions in this module perform no semantic interpretation of their inputs, and in general, the user is assumed to pass type-correct arguments - if this is not done, the effects are not defined.

With the exception of the *erl_parse()* data structures, the internal representations of abstract syntax trees are subject to change without notice, and should not be documented outside this module. Furthermore, we do not give any guarantees on how an abstract syntax tree may or may not be represented, *with the following exceptions*: no syntax tree is represented by a single atom, such as `none`, by a list constructor `[X | Y]`, or by the empty list `[]`. This can be relied on when writing functions that operate on syntax trees.

DATA TYPES

```
encoding() = utf8 | unicode | latin1
erl_parse() = abstract_form() (see module erl_parse) | abstract_expr() (see
module erl_parse)
```

The representation built by the Erlang standard library parser `erl_parse`. This is a subset of the *syntaxTree()* type.

```
forms() = syntaxTree() | [syntaxTree()]
guard() = none | syntaxTree() | [syntaxTree()] | [[syntaxTree()]]
padding() = none | integer()
syntaxTree()
```

An abstract syntax tree. The *erl_parse()* "parse tree" representation is a proper subset of the `syntaxTree()` representation.

Every abstract syntax tree node has a *type*, given by the function *type/1*. Each node also has associated *attributes*; see *get_attrs/1* for details. The functions *make_tree/2* and *subtrees/1* are generic constructor/decomposition functions for abstract syntax trees. The functions *abstract/1* and *concrete/1* convert between constant Erlang terms and their syntactic representations. The set of syntax tree nodes is extensible through the *tree/2* function.

A syntax tree can be transformed to the *erl_parse()* representation with the *revert/1* function.

```
syntaxTreeAttributes()
```

This is an abstract representation of syntax tree node attributes; see the function *get_attrs/1*.

Exports

abstract(T::term()) -> syntaxTree()

Returns the syntax tree corresponding to an Erlang term. `Term` must be a literal term, i.e., one that can be represented as a source code literal. Thus, it may not contain a process identifier, port, reference, binary or function value as a subterm. The function recognises printable strings, in order to get a compact and readable representation. Evaluation fails with reason `badarg` if `Term` is not a literal term.

See also: concrete/1, is_literal/1.

add_ann(A::term(), Node::syntaxTree()) -> syntaxTree()

Appends the term `Annotation` to the list of user annotations of `Node`.

Note: this is equivalent to `set_ann(Node, [Annotation | get_ann(Node)])`, but potentially more efficient.

See also: get_ann/1, set_ann/2.

add_postcomments(Cs::[syntaxTree()], Node::syntaxTree()) -> syntaxTree()

Appends `Comments` to the post-comments of `Node`.

Note: This is equivalent to `set_postcomments(Node, get_postcomments(Node) ++ Comments)`, but potentially more efficient.

See also: add_precomments/2, comment/2, get_postcomments/1, join_comments/2, set_postcomments/2.

add_precomments(Cs::[syntaxTree()], Node::syntaxTree()) -> syntaxTree()

Appends `Comments` to the pre-comments of `Node`.

Note: This is equivalent to `set_precomments(Node, get_precomments(Node) ++ Comments)`, but potentially more efficient.

See also: add_postcomments/2, comment/2, get_precomments/1, join_comments/2, set_precomments/2.

application(Operator::syntaxTree(), Arguments::[syntaxTree()]) -> syntaxTree()

Creates an abstract function application expression. If `Arguments` is `[A1, ..., An]`, the result represents "`Operator(A1, ..., An)`".

See also: application/3, application_arguments/1, application_operator/1.

application(Module::none | syntaxTree(), Name::syntaxTree(), Arguments::[syntaxTree()]) -> syntaxTree()

Creates an abstract function application expression. If `Module` is `none`, this is call is equivalent to `application(Function, Arguments)`, otherwise it is equivalent to `application(module_qualifier(Module, Function), Arguments)`.

(This is a utility function.)

See also: application/2, module_qualifier/2.

application_arguments(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of argument subtrees of an `application` node.

See also: application/2.

`application_operator(Node::syntaxTree()) -> syntaxTree()`

Returns the operator subtree of an `application` node.

Note: if `Node` represents "`M:F(...)`", then the result is the subtree representing "`M:F`".

See also: application/2, module_qualifier/2.

`arity_qualifier(Body::syntaxTree(), Arity::syntaxTree()) -> syntaxTree()`

Creates an abstract arity qualifier. The result represents "`Body/Arity`".

See also: arity_qualifier_argument/1, arity_qualifier_body/1.

`arity_qualifier_argument(Node::syntaxTree()) -> syntaxTree()`

Returns the argument (the arity) subtree of an `arity_qualifier` node.

See also: arity_qualifier/2.

`arity_qualifier_body(Node::syntaxTree()) -> syntaxTree()`

Returns the body subtree of an `arity_qualifier` node.

See also: arity_qualifier/2.

`atom(Name::atom() | string()) -> syntaxTree()`

Creates an abstract atom literal. The print name of the atom is the character sequence represented by `Name`.

See also: atom_literal/1, atom_name/1, atom_value/1, is_atom/2.

`atom_literal(Node::syntaxTree()) -> string()`

Returns the literal string represented by an `atom` node. This includes surrounding single-quote characters if necessary.

Note that e.g. the result of `atom("x\ny")` represents any and all of `x\ny", `x\12y", `x\012y" and `x\^Jy\"; see *string/1*.

See also: atom/1, string/1.

`atom_name(Node::syntaxTree()) -> string()`

Returns the printname of an `atom` node.

See also: atom/1.

`atom_value(Node::syntaxTree()) -> atom()`

Returns the value represented by an `atom` node.

See also: atom/1.

`attribute(Name::syntaxTree()) -> syntaxTree()`

Equivalent to *attribute(Name, none).*

attribute(Name::syntaxTree(), Args::none | [syntaxTree()]) -> syntaxTree()

Creates an abstract program attribute. If `Arguments` is `[A1, ..., An]`, the result represents "`-Name(A1, ..., An).`". Otherwise, if `Arguments` is `none`, the result represents "`-Name.`". The latter form makes it possible to represent preprocessor directives such as "`-endif.`". Attributes are source code forms.

Note: The preprocessor macro definition directive "`-define(Name, Body).`" has relatively few requirements on the syntactical form of `Body` (viewed as a sequence of tokens). The `text` node type can be used for a `Body` that is not a normal Erlang construct.

See also: attribute/1, *attribute_arguments/1*, *attribute_name/1*, *is_form/1*, *text/1*.

attribute_arguments(Node::syntaxTree()) -> none | [syntaxTree()]

Returns the list of argument subtrees of an `attribute` node, if any. If `Node` represents "`-Name.`", the result is `none`. Otherwise, if `Node` represents "`-Name(E1, ..., En).`", `[E1, ..., E1]` is returned.

See also: attribute/1.

attribute_name(Node::syntaxTree()) -> syntaxTree()

Returns the name subtree of an `attribute` node.

See also: attribute/1.

binary(List::[syntaxTree()]) -> syntaxTree()

Creates an abstract binary-object template. If `Fields` is `[F1, ..., Fn]`, the result represents "`<<F1, ..., Fn>>`".

See also: binary_field/2, *binary_fields/1*.

binary_comp(Template::syntaxTree(), Body::[syntaxTree()]) -> syntaxTree()

Creates an abstract binary comprehension. If `Body` is `[E1, ..., En]`, the result represents "`<<Template || E1, ..., En>>`".

See also: binary_comp_body/1, *binary_comp_template/1*, *generator/2*.

binary_comp_body(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of body subtrees of a `binary_comp` node.

See also: binary_comp/2.

binary_comp_template(Node::syntaxTree()) -> syntaxTree()

Returns the template subtree of a `binary_comp` node.

See also: binary_comp/2.

binary_field(Body::syntaxTree()) -> syntaxTree()

Equivalent to *binary_field(Body, [])*.

binary_field(Body::syntaxTree(), Types::[syntaxTree()]) -> syntaxTree()

Creates an abstract binary template field. If `Types` is the empty list, the result simply represents "`Body`", otherwise, if `Types` is `[T1, ..., Tn]`, the result represents "`Body/T1-...-Tn`".

See also: binary/1, *binary_field/1*, *binary_field/3*, *binary_field_body/1*, *binary_field_size/1*, *binary_field_types/1*.

binary_field(Body::syntaxTree(), Size::none | syntaxTree(), Types::[syntaxTree()]) -> syntaxTree()

Creates an abstract binary template field. If `Size` is `none`, this is equivalent to "`binary_field(Body, Types)`", otherwise it is equivalent to "`binary_field(size_qualifier(Body, Size), Types)`".

(This is a utility function.)

See also: binary/1, *binary_field/2*, *size_qualifier/2*.

binary_field_body(Node::syntaxTree()) -> syntaxTree()

Returns the body subtree of a `binary_field`.

See also: binary_field/2.

binary_field_size(Node::syntaxTree()) -> none | syntaxTree()

Returns the size specifier subtree of a `binary_field` node, if any. If `Node` represents "`Body:Size`" or "`Body:Size/T1, ..., Tn`", the result is `Size`, otherwise `none` is returned.

(This is a utility function.)

See also: binary_field/2, *binary_field/3*.

binary_field_types(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of type-specifier subtrees of a `binary_field` node. If `Node` represents "`.../T1, ..., Tn`", the result is `[T1, ..., Tn]`, otherwise the result is the empty list.

See also: binary_field/2.

binary_fields(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of field subtrees of a `binary` node.

See also: binary/1, *binary_field/2*.

binary_generator(Pattern::syntaxTree(), Body::syntaxTree()) -> syntaxTree()

Creates an abstract binary_generator. The result represents "`Pattern <- Body`".

See also: binary_comp/2, *binary_generator_body/1*, *binary_generator_pattern/1*, *list_comp/2*.

binary_generator_body(Node::syntaxTree()) -> syntaxTree()

Returns the body subtree of a `generator` node.

See also: binary_generator/2.

binary_generator_pattern(Node::syntaxTree()) -> syntaxTree()

Returns the pattern subtree of a `generator` node.

See also: binary_generator/2.

block_expr(Body::[syntaxTree()]) -> syntaxTree()

Creates an abstract block expression. If `Body` is `[B1, ..., Bn]`, the result represents "`begin B1, ..., Bn end`".

See also: block_expr_body/1.

block_expr_body(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of body subtrees of a `block_expr` node.

See also: block_expr/1.

case_expr(Argument::syntaxTree(), Clauses::[syntaxTree()]) -> syntaxTree()

Creates an abstract case-expression. If `Clauses` is `[C1, ..., Cn]`, the result represents "`case Argument of C1; ...; Cn end`". More exactly, if each `Ci` represents "`(Pi) Gi -> Bi`", then the result represents "`case Argument of P1 G1 -> B1; ...; Pn Gn -> Bn end`".

See also: case_expr_argument/1, case_expr_clauses/1, clause/3, cond_expr/1, if_expr/1.

case_expr_argument(Node::syntaxTree()) -> syntaxTree()

Returns the argument subtree of a `case_expr` node.

See also: case_expr/2.

case_expr_clauses(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of clause subtrees of a `case_expr` node.

See also: case_expr/2.

catch_expr(Expr::syntaxTree()) -> syntaxTree()

Creates an abstract catch-expression. The result represents "`catch Expr`".

See also: catch_expr_body/1.

catch_expr_body(Node::syntaxTree()) -> syntaxTree()

Returns the body subtree of a `catch_expr` node.

See also: catch_expr/1.

char(Char::char()) -> syntaxTree()

Creates an abstract character literal. The result represents "`$Name`", where `Name` corresponds to `Value`.

Note: the literal corresponding to a particular character value is not uniquely defined. E.g., the character "`a`" can be written both as "`$a`" and "`$\141`", and a Tab character can be written as "`$\11`", "`$\011`" or "`$\t`".

See also: char_literal/1, char_literal/2, char_value/1, is_char/2.

char_literal(Node::syntaxTree()) -> nonempty_string()

Returns the literal string represented by a `char` node. This includes the leading "`$`" character. Characters beyond 255 will be escaped.

See also: char/1.

char_literal(Node::syntaxTree(), X2::encoding()) -> nonempty_string()

Returns the literal string represented by a `char` node. This includes the leading "`$`" character. Depending on the encoding a character beyond 255 will be escaped ('latin1') or copied as is ('utf8').

See also: char/1.

char_value(Node::syntaxTree()) -> char()

Returns the value represented by a `char` node.

See also: char/1.

class_qualifier(Class::syntaxTree(), Body::syntaxTree()) -> syntaxTree()

Creates an abstract class qualifier. The result represents "`Class:Body`".

See also: class_qualifier_argument/1, *class_qualifier_body/1*, *try_expr/4*.

class_qualifier_argument(Node::syntaxTree()) -> syntaxTree()

Returns the argument (the class) subtree of a `class_qualifier` node.

See also: class_qualifier/2.

class_qualifier_body(Node::syntaxTree()) -> syntaxTree()

Returns the body subtree of a `class_qualifier` node.

See also: class_qualifier/2.

clause(Guard::guard(), Body::[syntaxTree()]) -> syntaxTree()

Equivalent to *clause([], Guard, Body)*.

clause(Patterns::[syntaxTree()], Guard::guard(), Body::[syntaxTree()]) -> syntaxTree()

Creates an abstract clause. If `Patterns` is `[P1, ..., Pn]` and `Body` is `[B1, ..., Bm]`, then if `Guard` is `none`, the result represents "`(P1, ..., Pn) -> B1, ..., Bm`", otherwise, unless `Guard` is a list, the result represents "`(P1, ..., Pn) when Guard -> B1, ..., Bm`".

For simplicity, the `Guard` argument may also be any of the following:

- An empty list `[]`. This is equivalent to passing `none`.
- A nonempty list `[E1, ..., Ej]` of syntax trees. This is equivalent to passing `conjunction([E1, ..., Ej])`.
- A nonempty list of lists of syntax trees `[[E1_1, ..., E1_k1], ..., [Ej_1, ..., Ej_kj]]`, which is equivalent to passing `disjunction([conjunction([E1_1, ..., E1_k1]), ..., conjunction([Ej_1, ..., Ej_kj])])`.

See also: clause/2, *clause_body/1*, *clause_guard/1*, *clause_patterns/1*.

clause_body(Node::syntaxTree()) -> [syntaxTree()]

Return the list of body subtrees of a `clause` node.

See also: clause/3.

clause_guard(Node::syntaxTree()) -> none | syntaxTree()

Returns the guard subtree of a `clause` node, if any. If `Node` represents "`(P1, ..., Pn) when Guard -> B1, ..., Bm`", `Guard` is returned. Otherwise, the result is `none`.

See also: clause/3.

clause_patterns(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of pattern subtrees of a `clause` node.

See also: clause/3.

comment(Strings::[string()]) -> syntaxTree()

Equivalent to *comment(none, Strings)*.

comment(Pad::padding(), Strings::[string()]) -> syntaxTree()

Creates an abstract comment with the given padding and text. If `Strings` is a (possibly empty) list `["Txt1", ..., "TxtN"]`, the result represents the source code text

```
%Txt1
...
%TxtN
```

`Padding` states the number of empty character positions to the left of the comment separating it horizontally from source code on the same line (if any). If `Padding` is `none`, a default positive number is used. If `Padding` is an integer less than 1, there should be no separating space. Comments are in themselves regarded as source program forms.

See also: comment/1, is_form/1.

comment_padding(Node::syntaxTree()) -> padding()

Returns the amount of padding before the comment, or `none`. The latter means that a default padding may be used.

See also: comment/2.

comment_text(Node::syntaxTree()) -> [string()]

Returns the lines of text of the abstract comment.

See also: comment/2.

compact_list(Node::syntaxTree()) -> syntaxTree()

Yields the most compact form for an abstract list skeleton. The result either represents "`[E1, ..., En | Tail]`", where `Tail` is not a list skeleton, or otherwise simply "`[E1, ..., En]`". Annotations on subtrees of `Node` that represent list skeletons may be lost, but comments will be propagated to the result. Returns `Node` itself if `Node` does not represent a list skeleton.

See also: list/2, normalize_list/1.

concrete(Node::syntaxTree()) -> term()

Returns the Erlang term represented by a syntax tree. Evaluation fails with reason `badarg` if `Node` does not represent a literal term.

Note: Currently, the set of syntax trees which have a concrete representation is larger than the set of trees which can be built using the function *abstract/1*. An abstract character will be concretised as an integer, while *abstract/1* does not at present yield an abstract character for any input. (Use the *char/1* function to explicitly create an abstract character.)

Note: `arity_qualifier` nodes are recognized. This is to follow The Erlang Parser when it comes to wild attributes: both {F, A} and F/A are recognized, which makes it possible to turn wild attributes into recognized attributes without at the same time making it impossible to compile files using the new syntax with the old version of the Erlang Compiler.

See also: abstract/1, *char/1*, *is_literal/1*.

`cond_expr(Clauses::[syntaxTree()]) -> syntaxTree()`

Creates an abstract cond-expression. If `Clauses` is `[C1, ..., Cn]`, the result represents "`cond C1; ...; Cn end`". More exactly, if each `Ci` represents "`() Ei -> Bi`", then the result represents "`cond E1 -> B1; ...; En -> Bn end`".

See also: case_expr/2, *clause/3*, *cond_expr_clauses/1*.

`cond_expr_clauses(Node::syntaxTree()) -> [syntaxTree()]`

Returns the list of clause subtrees of a `cond_expr` node.

See also: cond_expr/1.

`conjunction(Tests::[syntaxTree()]) -> syntaxTree()`

Creates an abstract conjunction. If `List` is `[E1, ..., En]`, the result represents "`E1, ..., En`".

See also: conjunction_body/1, *disjunction/1*.

`conjunction_body(Node::syntaxTree()) -> [syntaxTree()]`

Returns the list of body subtrees of a `conjunction` node.

See also: conjunction/1.

`cons(Head::syntaxTree(), Tail::syntaxTree()) -> syntaxTree()`

"Optimising" list skeleton cons operation. Creates an abstract list skeleton whose first element is `Head` and whose tail corresponds to `Tail`. This is similar to `list([Head], Tail)`, except that `Tail` may not be `none`, and that the result does not necessarily represent exactly "`[Head | Tail]`", but may depend on the `Tail` subtree. E.g., if `Tail` represents `[X, Y]`, the result may represent "`[Head, X, Y]`", rather than "`[Head | [X, Y]]`". Annotations on `Tail` itself may be lost if `Tail` represents a list skeleton, but comments on `Tail` are propagated to the result.

See also: list/2, *list_head/1*, *list_tail/1*.

`copy_ann(Source::syntaxTree(), Target::syntaxTree()) -> syntaxTree()`

Copies the list of user annotations from `Source` to `Target`.

Note: this is equivalent to `set_ann(Target, get_ann(Source))`, but potentially more efficient.

See also: get_ann/1, *set_ann/2*.

`copy_attrs(S::syntaxTree(), T::syntaxTree()) -> syntaxTree()`

Copies the attributes from `Source` to `Target`.

Note: this is equivalent to `set_attrs(Target, get_attrs(Source))`, but potentially more efficient.

See also: get_attrs/1, *set_attrs/2*.

copy_comments(Source::syntaxTree(), Target::syntaxTree()) -> syntaxTree()

Copies the pre- and postcomments from `Source` to `Target`.

Note: This is equivalent to `set_postcomments(set_precomments(Target, get_precomments(Source)), get_postcomments(Source))`, but potentially more efficient.

See also: comment/2, *get_postcomments/1*, *get_precomments/1*, *set_postcomments/2*, *set_precomments/2*.

copy_pos(Source::syntaxTree(), Target::syntaxTree()) -> syntaxTree()

Copies the position information from `Source` to `Target`.

This is equivalent to `set_pos(Target, get_pos(Source))`, but potentially more efficient.

See also: get_pos/1, *set_pos/2*.

data(Tree::syntaxTree()) -> term()

For special purposes only. Returns the associated data of a syntax tree node. Evaluation fails with reason `badarg` if `is_tree(Node)` does not yield `true`.

See also: tree/2.

disjunction(Tests::[syntaxTree()]) -> syntaxTree()

Creates an abstract disjunction. If `List` is `[E1, ..., En]`, the result represents "`E1; ...; En`".

See also: conjunction/1, *disjunction_body/1*.

disjunction_body(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of body subtrees of a `disjunction` node.

See also: disjunction/1.

eof_marker() -> syntaxTree()

Creates an abstract end-of-file marker. This represents the end of input when reading a sequence of source code forms. An end-of-file marker is itself regarded as a source code form (namely, the last in any sequence in which it occurs). It has no defined lexical form.

Note: this is retained only for backwards compatibility with existing parsers and tools.

See also: error_marker/1, *is_form/1*, *warning_marker/1*.

error_marker(Error::term()) -> syntaxTree()

Creates an abstract error marker. The result represents an occurrence of an error in the source code, with an associated Erlang I/O ErrorInfo structure given by `Error` (see module *io(3)* for details). Error markers are regarded as source code forms, but have no defined lexical form.

Note: this is supported only for backwards compatibility with existing parsers and tools.

See also: eof_marker/0, *error_marker_info/1*, *is_form/1*, *warning_marker/1*.

error_marker_info(Node::syntaxTree()) -> term()

Returns the ErrorInfo structure of an `error_marker` node.

See also: error_marker/1.

`flatten_form_list(Node::syntaxTree()) -> syntaxTree()`

Flattens sublists of a `form_list` node. Returns `Node` with all subtrees of type `form_list` recursively expanded, yielding a single "flat" abstract form sequence.

See also: form_list/1.

`float(Value::float()) -> syntaxTree()`

Creates an abstract floating-point literal. The lexical representation is the decimal floating-point numeral of `Value`.

See also: float_literal/1, *float_value/1*.

`float_literal(Node::syntaxTree()) -> string()`

Returns the numeral string represented by a `float` node.

See also: float/1.

`float_value(Node::syntaxTree()) -> float()`

Returns the value represented by a `float` node. Note that floating-point values should usually not be compared for equality.

See also: float/1.

`form_list(Forms::[syntaxTree()]) -> syntaxTree()`

Creates an abstract sequence of "source code forms". If `Forms` is `[F1, ..., Fn]`, where each `Fi` is a form (see *is_form/1*, the result represents

```
F1
...
Fn
```

where the `Fi` are separated by one or more line breaks. A node of type `form_list` is itself regarded as a source code form; see *flatten_form_list/1*.

Note: this is simply a way of grouping source code forms as a single syntax tree, usually in order to form an Erlang module definition.

See also: flatten_form_list/1, *form_list_elements/1*, *is_form/1*.

`form_list_elements(Node::syntaxTree()) -> [syntaxTree()]`

Returns the list of subnodes of a `form_list` node.

See also: form_list/1.

`fun_expr(Clauses::[syntaxTree()]) -> syntaxTree()`

Creates an abstract fun-expression. If `Clauses` is `[C1, ..., Cn]`, the result represents "`fun C1; ...; Cn end`". More exactly, if each `Ci` represents "`(Pi1, ..., Pim) Gi -> Bi`", then the result represents "`fun (P11, ..., P1m) G1 -> B1; ...; (Pn1, ..., Pnm) Gn -> Bn end`".

See also: fun_expr_arity/1, *fun_expr_clauses/1*.

fun_expr_arity(Node::syntaxTree()) -> arity()

Returns the arity of a `fun_expr` node. The result is the number of parameter patterns in the first clause of the fun-expression; subsequent clauses are ignored.

An exception is thrown if `fun_expr_clauses(Node)` returns an empty list, or if the first element of that list is not a syntax tree `C` of type `clause` such that `clause_patterns(C)` is a nonempty list.

See also: clause/3, clause_patterns/1, fun_expr/1, fun_expr_clauses/1.

fun_expr_clauses(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of clause subtrees of a `fun_expr` node.

See also: fun_expr/1.

function(Name::syntaxTree(), Clauses::[syntaxTree()]) -> syntaxTree()

Creates an abstract function definition. If `Clauses` is `[C1, ..., Cn]`, the result represents "`Name C1; ...; Name Cn.`". More exactly, if each `Ci` represents "`(Pi1, ..., Pim) Gi -> Bi`", then the result represents "`Name(P11, ..., P1m) G1 -> B1; ...; Name(Pn1, ..., Pnm) Gn -> Bn.`". Function definitions are source code forms.

See also: function_arity/1, function_clauses/1, function_name/1, is_form/1.

function_arity(Node::syntaxTree()) -> arity()

Returns the arity of a `function` node. The result is the number of parameter patterns in the first clause of the function; subsequent clauses are ignored.

An exception is thrown if `function_clauses(Node)` returns an empty list, or if the first element of that list is not a syntax tree `C` of type `clause` such that `clause_patterns(C)` is a nonempty list.

See also: clause/3, clause_patterns/1, function/2, function_clauses/1.

function_clauses(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of clause subtrees of a `function` node.

See also: function/2.

function_name(Node::syntaxTree()) -> syntaxTree()

Returns the name subtree of a `function` node.

See also: function/2.

generator(Pattern::syntaxTree(), Body::syntaxTree()) -> syntaxTree()

Creates an abstract generator. The result represents "`Pattern <- Body`".

See also: binary_comp/2, generator_body/1, generator_pattern/1, list_comp/2.

generator_body(Node::syntaxTree()) -> syntaxTree()

Returns the body subtree of a `generator` node.

See also: generator/2.

`generator_pattern(Node::syntaxTree()) -> syntaxTree()`

Returns the pattern subtree of a `generator` node.

See also: generator/2.

`get_ann(Tree::syntaxTree()) -> [term()]`

Returns the list of user annotations associated with a syntax tree node. For a newly created node, this is the empty list. The annotations may be any terms.

See also: get_attrs/1, *set_ann/2*.

`get_attrs(Tree::syntaxTree()) -> syntaxTreeAttributes()`

Returns a representation of the attributes associated with a syntax tree node. The attributes are all the extra information that can be attached to a node. Currently, this includes position information, source code comments, and user annotations. The result of this function cannot be inspected directly; only attached to another node (see *set_attrs/2*).

For accessing individual attributes, see *get_pos/1*, *get_ann/1*, *get_precomments/1* and *get_postcomments/1*.

See also: get_ann/1, *get_pos/1*, *get_postcomments/1*, *get_precomments/1*, *set_attrs/2*.

`get_pos(Tree::syntaxTree()) -> term()`

Returns the position information associated with `Node`. This is usually a nonnegative integer (indicating the source code line number), but may be any term. By default, all new tree nodes have their associated position information set to the integer zero.

See also: get_attrs/1, *set_pos/2*.

`get_postcomments(Tree::syntaxTree()) -> [syntaxTree()]`

Returns the associated post-comments of a node. This is a possibly empty list of abstract comments, in top-down textual order. When the code is formatted, post-comments are typically displayed to the right of and/or below the node. For example:

```
    {foo, X, Y}     % Post-comment of tuple
```

If possible, the comment should be moved past any following separator characters on the same line, rather than placing the separators on the following line. E.g.:

```
    foo([X | Xs], Y) ->
        foo(Xs, bar(X));     % Post-comment of 'bar(X)' node
     ...
```

(where the comment is moved past the rightmost "`)`" and the "`;`").

See also: comment/2, *get_attrs/1*, *get_precomments/1*, *set_postcomments/2*.

`get_precomments(Tree::syntaxTree()) -> [syntaxTree()]`

Returns the associated pre-comments of a node. This is a possibly empty list of abstract comments, in top-down textual order. When the code is formatted, pre-comments are typically displayed directly above the node. For example:

```
    % Pre-comment of function
    foo(X) -> {bar, X}.
```

If possible, the comment should be moved before any preceding separator characters on the same line. E.g.:

```
    foo([X | Xs]) ->
        % Pre-comment of 'bar(X)' node
        [bar(X) | foo(Xs)];
    ...
```

(where the comment is moved before the "`[`").

See also: comment/2, get_attrs/1, get_postcomments/1, set_precomments/2.

`has_comments(Tree::syntaxTree()) -> boolean()`

Yields `false` if the node has no associated comments, and `true` otherwise.

Note: This is equivalent to `(get_precomments(Node) == []) and (get_postcomments(Node) == [])`, but potentially more efficient.

See also: get_postcomments/1, get_precomments/1, remove_comments/1.

`if_expr(Clauses::[syntaxTree()]) -> syntaxTree()`

Creates an abstract if-expression. If `Clauses` is `[C1, ..., Cn]`, the result represents "`if C1; ...; Cn end`". More exactly, if each `Ci` represents "`() Gi -> Bi`", then the result represents "`if G1 -> B1; ...; Gn -> Bn end`".

See also: case_expr/2, clause/3, if_expr_clauses/1.

`if_expr_clauses(Node::syntaxTree()) -> [syntaxTree()]`

Returns the list of clause subtrees of an `if_expr` node.

See also: if_expr/1.

`implicit_fun(Name::syntaxTree()) -> syntaxTree()`

Creates an abstract "implicit fun" expression. The result represents "`fun Name`". `Name` should represent either `F/A` or `M:F/A`

See also: arity_qualifier/2, implicit_fun/2, implicit_fun/3, implicit_fun_name/1, module_qualifier/2.

`implicit_fun(Name::syntaxTree(), Arity::none | syntaxTree()) -> syntaxTree()`

Creates an abstract "implicit fun" expression. If `Arity` is `none`, this is equivalent to `implicit_fun(Name)`, otherwise it is equivalent to `implicit_fun(arity_qualifier(Name, Arity))`.

(This is a utility function.)

See also: implicit_fun/1, implicit_fun/3.

`implicit_fun(Module::none | syntaxTree(), Name::syntaxTree(), Arity::syntaxTree()) -> syntaxTree()`

Creates an abstract module-qualified "implicit fun" expression. If `Module` is `none`, this is equivalent to `implicit_fun(Name, Arity)`, otherwise it is equivalent to `implicit_fun(module_qualifier(Module, arity_qualifier(Name, Arity))`.

(This is a utility function.)

See also: implicit_fun/1, implicit_fun/2.

implicit_fun_name(Node::syntaxTree()) -> syntaxTree()

Returns the name subtree of an `implicit_fun` node.

Note: if `Node` represents "`fun N/A`" or "`fun M:N/A`", then the result is the subtree representing "`N/A`" or "`M:N/A`", respectively.

See also: arity_qualifier/2, implicit_fun/1, module_qualifier/2.

infix_expr(Left::syntaxTree(), Operator::syntaxTree(), Right::syntaxTree()) -> syntaxTree()

Creates an abstract infix operator expression. The result represents "`Left Operator Right`".

See also: infix_expr_left/1, infix_expr_operator/1, infix_expr_right/1, prefix_expr/2.

infix_expr_left(Node::syntaxTree()) -> syntaxTree()

Returns the left argument subtree of an `infix_expr` node.

See also: infix_expr/3.

infix_expr_operator(Node::syntaxTree()) -> syntaxTree()

Returns the operator subtree of an `infix_expr` node.

See also: infix_expr/3.

infix_expr_right(Node::syntaxTree()) -> syntaxTree()

Returns the right argument subtree of an `infix_expr` node.

See also: infix_expr/3.

integer(Value::integer()) -> syntaxTree()

Creates an abstract integer literal. The lexical representation is the canonical decimal numeral of `Value`.

See also: integer_literal/1, integer_value/1, is_integer/2.

integer_literal(Node::syntaxTree()) -> string()

Returns the numeral string represented by an `integer` node.

See also: integer/1.

integer_value(Node::syntaxTree()) -> integer()

Returns the value represented by an `integer` node.

See also: integer/1.

is_atom(Node::syntaxTree(), Value::atom()) -> boolean()

Returns `true` if `Node` has type `atom` and represents `Value`, otherwise `false`.

See also: atom/1.

is_char(Node::syntaxTree(), Value::char()) -> boolean()

Returns `true` if `Node` has type `char` and represents `Value`, otherwise `false`.

See also: char/1.

is_form(Node::syntaxTree()) -> boolean()

Returns `true` if `Node` is a syntax tree representing a so-called "source code form", otherwise `false`. Forms are the Erlang source code units which, placed in sequence, constitute an Erlang program. Current form types are:

`attribute` `comment` `error_marker` `eof_marker`
`form_list` `function` `warning_marker` `text`

See also: attribute/2, comment/2, eof_marker/0, error_marker/1, form_list/1, function/2, type/1, warning_marker/1.

is_integer(Node::syntaxTree(), Value::integer()) -> boolean()

Returns `true` if `Node` has type `integer` and represents `Value`, otherwise `false`.

See also: integer/1.

is_leaf(Node::syntaxTree()) -> boolean()

Returns `true` if `Node` is a leaf node, otherwise `false`. The currently recognised leaf node types are:

`atom` `char` `comment` `eof_marker` `error_marker`
`float` `integer` `nil` `operator` `string`
`text` `underscore` `variable` `warning_marker`

A node of type `tuple` is a leaf node if and only if its arity is zero.

Note: not all literals are leaf nodes, and vice versa. E.g., tuples with nonzero arity and nonempty lists may be literals, but are not leaf nodes. Variables, on the other hand, are leaf nodes but not literals.

See also: is_literal/1, type/1.

is_list_skeleton(Node::syntaxTree()) -> boolean()

Returns `true` if `Node` has type `list` or `nil`, otherwise `false`.

See also: list/2, nil/0.

is_literal(T::syntaxTree()) -> boolean()

Returns `true` if `Node` represents a literal term, otherwise `false`. This function returns `true` if and only if the value of `concrete(Node)` is defined.

See also: abstract/1, concrete/1.

is_proper_list(Node::syntaxTree()) -> boolean()

Returns `true` if `Node` represents a proper list, and `false` otherwise. A proper list is a list skeleton either on the form "`[]`" or "`[E1, ..., En]`", or "`[... | Tail]`" where recursively `Tail` also represents a proper list.

Note: Since `Node` is a syntax tree, the actual run-time values corresponding to its subtrees may often be partially or completely unknown. Thus, if `Node` represents e.g. "`[... | Ns]`" (where `Ns` is a variable), then the function will return `false`, because it is not known whether `Ns` will be bound to a list at run-time. If `Node` instead represents e.g. "`[1, 2, 3]`" or "`[A | []]`", then the function will return `true`.

See also: list/2.

is_string(Node::syntaxTree(), Value::string()) -> boolean()

Returns `true` if `Node` has type `string` and represents `Value`, otherwise `false`.

See also: string/1.

`is_tree(Tree::syntaxTree()) -> boolean()`

For special purposes only. Returns `true` if `Tree` is an abstract syntax tree and `false` otherwise.

Note: this function yields `false` for all "old-style" `erl_parse`-compatible "parse trees".

See also: tree/2.

`join_comments(Source::syntaxTree(), Target::syntaxTree()) -> syntaxTree()`

Appends the comments of `Source` to the current comments of `Target`.

Note: This is equivalent to `add_postcomments(get_postcomments(Source), add_precomments(get_precomments(Source), Target))`, but potentially more efficient.

See also: add_postcomments/2, *add_precomments/2*, *comment/2*, *get_postcomments/1*, *get_precomments/1*.

`list(List::[syntaxTree()]) -> syntaxTree()`

Equivalent to *list(List, none)*.

`list(Elements::[syntaxTree()], Tail::none | syntaxTree()) -> syntaxTree()`

Constructs an abstract list skeleton. The result has type `list` or `nil`. If `List` is a nonempty list `[E1, ..., En]`, the result has type `list` and represents either "`[E1, ..., En]`", if `Tail` is `none`, or otherwise "`[E1, ..., En | Tail]`". If `List` is the empty list, `Tail` *must* be `none`, and in that case the result has type `nil` and represents "`[]`" (see *nil/0*).

The difference between lists as semantic objects (built up of individual "cons" and "nil" terms) and the various syntactic forms for denoting lists may be bewildering at first. This module provides functions both for exact control of the syntactic representation as well as for the simple composition and deconstruction in terms of cons and head/tail operations.

Note: in `list(Elements, none)`, the "nil" list terminator is implicit and has no associated information (see *get_attrs/1*), while in the seemingly equivalent `list(Elements, Tail)` when `Tail` has type `nil`, the list terminator subtree `Tail` may have attached attributes such as position, comments, and annotations, which will be preserved in the result.

See also: compact_list/1, *cons/2*, *get_attrs/1*, *is_list_skeleton/1*, *is_proper_list/1*, *list/1*, *list_elements/1*, *list_head/1*, *list_length/1*, *list_prefix/1*, *list_suffix/1*, *list_tail/1*, *nil/0*, *normalize_list/1*.

`list_comp(Template::syntaxTree(), Body::[syntaxTree()]) -> syntaxTree()`

Creates an abstract list comprehension. If `Body` is `[E1, ..., En]`, the result represents "`[Template || E1, ..., En]`".

See also: generator/2, *list_comp_body/1*, *list_comp_template/1*.

`list_comp_body(Node::syntaxTree()) -> [syntaxTree()]`

Returns the list of body subtrees of a `list_comp` node.

See also: list_comp/2.

`list_comp_template(Node::syntaxTree()) -> syntaxTree()`

Returns the template subtree of a `list_comp` node.

See also: list_comp/2.

`list_elements(Node::syntaxTree()) -> [syntaxTree()]`

Returns the list of element subtrees of a list skeleton. `Node` must represent a proper list. E.g., if `Node` represents "`[X1, X2 | [X3, X4 | []]`", then `list_elements(Node)` yields the list `[X1, X2, X3, X4]`.

See also: is_proper_list/1, list/2.

`list_head(Node::syntaxTree()) -> syntaxTree()`

Returns the head element subtree of a `list` node. If `Node` represents "`[Head ...]`", the result will represent "`Head`".

See also: cons/2, list/2, list_tail/1.

`list_length(Node::syntaxTree()) -> non_neg_integer()`

Returns the number of element subtrees of a list skeleton. `Node` must represent a proper list. E.g., if `Node` represents "`[X1 | [X2, X3 | [X4, X5, X6]]]`", then `list_length(Node)` returns the integer 6.

Note: this is equivalent to `length(list_elements(Node))`, but potentially more efficient.

See also: is_proper_list/1, list/2, list_elements/1.

`list_prefix(Node::syntaxTree()) -> [syntaxTree()]`

Returns the prefix element subtrees of a `list` node. If `Node` represents "`[E1, ..., En]`" or "`[E1, ..., En | Tail]`", the returned value is `[E1, ..., En]`.

See also: list/2.

`list_suffix(Node::syntaxTree()) -> none | syntaxTree()`

Returns the suffix subtree of a `list` node, if one exists. If `Node` represents "`[E1, ..., En | Tail]`", the returned value is `Tail`, otherwise, i.e., if `Node` represents "`[E1, ..., En]`", `none` is returned.

Note that even if this function returns some `Tail` that is not `none`, the type of `Tail` can be `nil`, if the tail has been given explicitly, and the list skeleton has not been compacted (see *compact_list/1*).

See also: compact_list/1, list/2, nil/0.

`list_tail(Node::syntaxTree()) -> syntaxTree()`

Returns the tail of a `list` node. If `Node` represents a single-element list "`[E]`", then the result has type `nil`, representing "`[]`". If `Node` represents "`[E1, E2 ...]`", the result will represent "`[E2 ...]`", and if `Node` represents "`[Head | Tail]`", the result will represent "`Tail`".

See also: cons/2, list/2, list_head/1.

`macro(Name::syntaxTree()) -> syntaxTree()`

Equivalent to *macro(Name, none)*.

`macro(Name::syntaxTree(), Arguments::none | [syntaxTree()]) -> syntaxTree()`

Creates an abstract macro application. If `Arguments` is `none`, the result represents "`?Name`", otherwise, if `Arguments` is `[A1, ..., An]`, the result represents "`?Name(A1, ..., An)`".

Notes: if `Arguments` is the empty list, the result will thus represent "`?Name()`", including a pair of matching parentheses.

The only syntactical limitation imposed by the preprocessor on the arguments to a macro application (viewed as sequences of tokens) is that they must be balanced with respect to parentheses, brackets, `begin ... end`, `case ... end`, etc. The `text` node type can be used to represent arguments which are not regular Erlang constructs.

See also: macro/1, *macro_arguments/1*, *macro_name/1*, *text/1*.

`macro_arguments(Node::syntaxTree()) -> none | [syntaxTree()]`

Returns the list of argument subtrees of a `macro` node, if any. If `Node` represents "`?Name`", `none` is returned. Otherwise, if `Node` represents "`?Name(A1, ..., An)`", `[A1, ..., An]` is returned.

See also: macro/2.

`macro_name(Node::syntaxTree()) -> syntaxTree()`

Returns the name subtree of a `macro` node.

See also: macro/2.

`make_tree(X1::atom(), X2::[[syntaxTree()]]) -> syntaxTree()`

Creates a syntax tree with the given type and subtrees. `Type` must be a node type name (see *type/1*) that does not denote a leaf node type (see *is_leaf/1*). `Groups` must be a *nonempty* list of groups of syntax trees, representing the subtrees of a node of the given type, in left-to-right order as they would occur in the printed program text, grouped by category as done by *subtrees/1*.

The result of `copy_attrs(Node, make_tree(type(Node), subtrees(Node)))` (see *update_tree/2*) represents the same source code text as the original `Node`, assuming that `subtrees(Node)` yields a nonempty list. However, it does not necessarily have the same data representation as `Node`.

See also: copy_attrs/2, *is_leaf/1*, *subtrees/1*, *type/1*, *update_tree/2*.

`map_expr(Fields::[syntaxTree()]) -> syntaxTree()`

Equivalent to *map_expr(none, Fields)*.

`map_expr(Argument::none | syntaxTree(), Fields::[syntaxTree()]) -> syntaxTree()`

Creates an abstract map expression. If `Fields` is `[F1, ..., Fn]`, then if `Argument` is `none`, the result represents "`#{F1, ..., Fn}`", otherwise it represents "`Argument#{F1, ..., Fn}`".

See also: map_expr/1, *map_expr_argument/1*, *map_expr_fields/1*, *map_field_assoc/2*, *map_field_exact/2*.

`map_expr_argument(Node::syntaxTree()) -> none | syntaxTree()`

Returns the argument subtree of a `map_expr` node, if any. If `Node` represents "`#{...}`", `none` is returned. Otherwise, if `Node` represents "`Argument#{...}`", `Argument` is returned.

See also: map_expr/2.

`map_expr_fields(Node::syntaxTree()) -> [syntaxTree()]`

Returns the list of field subtrees of a `map_expr` node.

See also: map_expr/2.

map_field_assoc(Name::syntaxTree(), Value::syntaxTree()) -> syntaxTree()

Creates an abstract map assoc field. The result represents "`Name => Value`".

See also: map_expr/2, map_field_assoc_name/1, map_field_assoc_value/1.

map_field_assoc_name(Node::syntaxTree()) -> syntaxTree()

Returns the name subtree of a `map_field_assoc` node.

See also: map_field_assoc/2.

map_field_assoc_value(Node::syntaxTree()) -> syntaxTree()

Returns the value subtree of a `map_field_assoc` node.

See also: map_field_assoc/2.

map_field_exact(Name::syntaxTree(), Value::syntaxTree()) -> syntaxTree()

Creates an abstract map exact field. The result represents "`Name := Value`".

See also: map_expr/2, map_field_exact_name/1, map_field_exact_value/1.

map_field_exact_name(Node::syntaxTree()) -> syntaxTree()

Returns the name subtree of a `map_field_exact` node.

See also: map_field_exact/2.

map_field_exact_value(Node::syntaxTree()) -> syntaxTree()

Returns the value subtree of a `map_field_exact` node.

See also: map_field_exact/2.

match_expr(Pattern::syntaxTree(), Body::syntaxTree()) -> syntaxTree()

Creates an abstract match-expression. The result represents "`Pattern = Body`".

See also: match_expr_body/1, match_expr_pattern/1.

match_expr_body(Node::syntaxTree()) -> syntaxTree()

Returns the body subtree of a `match_expr` node.

See also: match_expr/2.

match_expr_pattern(Node::syntaxTree()) -> syntaxTree()

Returns the pattern subtree of a `match_expr` node.

See also: match_expr/2.

meta(T::syntaxTree()) -> syntaxTree()

Creates a meta-representation of a syntax tree. The result represents an Erlang expression "`MetaTree`" which, if evaluated, will yield a new syntax tree representing the same source code text as `Tree` (although the actual data representation may be different). The expression represented by `MetaTree` is *implementation independent* with regard to the data structures used by the abstract syntax tree implementation. Comments attached to nodes of `Tree` will be preserved, but other attributes are lost.

Any node in `Tree` whose node type is `variable` (see *type/1*), and whose list of annotations (see *get_ann/1*) contains the atom `meta_var`, will remain unchanged in the resulting tree, except that exactly one occurrence of `meta_var` is removed from its annotation list.

The main use of the function `meta/1` is to transform a data structure `Tree`, which represents a piece of program code, into a form that is *representation independent when printed*. E.g., suppose `Tree` represents a variable named "V". Then (assuming a function `print/1` for printing syntax trees), evaluating `print(abstract(Tree))` - simply using *abstract/1* to map the actual data structure onto a syntax tree representation - would output a string that might look something like "`{tree, variable, ..., "V", ...}`", which is obviously dependent on the implementation of the abstract syntax trees. This could e.g. be useful for caching a syntax tree in a file. However, in some situations like in a program generator generator (with two "generator"), it may be unacceptable. Using `print(meta(Tree))` instead would output a *representation independent* syntax tree generating expression; in the above case, something like "`erl_syntax:variable("V")`".

See also: abstract/1, *get_ann/1*, *type/1*.

`module_qualifier(Module::syntaxTree(), Body::syntaxTree()) -> syntaxTree()`

Creates an abstract module qualifier. The result represents "`Module:Body`".

See also: module_qualifier_argument/1, *module_qualifier_body/1*.

`module_qualifier_argument(Node::syntaxTree()) -> syntaxTree()`

Returns the argument (the module) subtree of a `module_qualifier` node.

See also: module_qualifier/2.

`module_qualifier_body(Node::syntaxTree()) -> syntaxTree()`

Returns the body subtree of a `module_qualifier` node.

See also: module_qualifier/2.

`named_fun_expr(Name::syntaxTree(), Clauses::[syntaxTree()]) -> syntaxTree()`

Creates an abstract named fun-expression. If `Clauses` is `[C1, ..., Cn]`, the result represents "`fun Name C1; ...; Name Cn end`". More exactly, if each `Ci` represents "`(Pi1, ..., Pim) Gi -> Bi`", then the result represents "`fun Name(P11, ..., P1m) G1 -> B1; ...; Name(Pn1, ..., Pnm) Gn -> Bn end`".

See also: named_fun_expr_arity/1, *named_fun_expr_clauses/1*, *named_fun_expr_name/1*.

`named_fun_expr_arity(Node::syntaxTree()) -> arity()`

Returns the arity of a `named_fun_expr` node. The result is the number of parameter patterns in the first clause of the named fun-expression; subsequent clauses are ignored.

An exception is thrown if `named_fun_expr_clauses(Node)` returns an empty list, or if the first element of that list is not a syntax tree `C` of type `clause` such that `clause_patterns(C)` is a nonempty list.

See also: clause/3, *clause_patterns/1*, *named_fun_expr/2*, *named_fun_expr_clauses/1*.

`named_fun_expr_clauses(Node::syntaxTree()) -> [syntaxTree()]`

Returns the list of clause subtrees of a `named_fun_expr` node.

See also: named_fun_expr/2.

named_fun_expr_name(Node::syntaxTree()) -> syntaxTree()

Returns the name subtree of a `named_fun_expr` node.

See also: named_fun_expr/2.

nil() -> syntaxTree()

Creates an abstract empty list. The result represents "`[]`". The empty list is traditionally called "nil".

See also: is_list_skeleton/1, list/2.

normalize_list(Node::syntaxTree()) -> syntaxTree()

Expands an abstract list skeleton to its most explicit form. If `Node` represents "`[E1, ..., En | Tail]`", the result represents "`[E1 | ... [En | Tail1] ... ]`", where `Tail1` is the result of `normalize_list(Tail)`. If `Node` represents "`[E1, ..., En]`", the result simply represents "`[E1 | ... [En | []] ... ]`". If `Node` does not represent a list skeleton, `Node` itself is returned.

See also: compact_list/1, list/2.

operator(Name::atom() | string()) -> syntaxTree()

Creates an abstract operator. The name of the operator is the character sequence represented by `Name`. This is analogous to the print name of an atom, but an operator is never written within single-quotes; e.g., the result of `operator(`+ +')' represents "++" rather than "`++"".

See also: atom/1, operator_literal/1, operator_name/1.

operator_literal(Node::syntaxTree()) -> string()

Returns the literal string represented by an `operator` node. This is simply the operator name as a string.

See also: operator/1.

operator_name(Node::syntaxTree()) -> atom()

Returns the name of an `operator` node. Note that the name is returned as an atom.

See also: operator/1.

parentheses(Expr::syntaxTree()) -> syntaxTree()

Creates an abstract parenthesised expression. The result represents "`(Body)`", independently of the context.

See also: parentheses_body/1.

parentheses_body(Node::syntaxTree()) -> syntaxTree()

Returns the body subtree of a `parentheses` node.

See also: parentheses/1.

prefix_expr(Operator::syntaxTree(), Argument::syntaxTree()) -> syntaxTree()

Creates an abstract prefix operator expression. The result represents "`Operator Argument`".

See also: infix_expr/3, prefix_expr_argument/1, prefix_expr_operator/1.

prefix_expr_argument(Node::syntaxTree()) -> syntaxTree()

Returns the argument subtree of a `prefix_expr` node.

See also: prefix_expr/2.

prefix_expr_operator(Node::syntaxTree()) -> syntaxTree()

Returns the operator subtree of a `prefix_expr` node.

See also: prefix_expr/2.

receive_expr(Clauses::[syntaxTree()]) -> syntaxTree()

Equivalent to *receive_expr(Clauses, none, [])*.

receive_expr(Clauses::[syntaxTree()], Timeout::none | syntaxTree(), Action::[syntaxTree()]) -> syntaxTree()

Creates an abstract receive-expression. If `Timeout` is `none`, the result represents "`receive C1; ...; Cn end`" (the `Action` argument is ignored). Otherwise, if `Clauses` is `[C1, ..., Cn]` and `Action` is `[A1, ..., Am]`, the result represents "`receive C1; ...; Cn after Timeout -> A1, ..., Am end`". More exactly, if each `Ci` represents "`(Pi) Gi -> Bi`", then the result represents "`receive P1 G1 -> B1; ...; Pn Gn -> Bn ... end`".

Note that in Erlang, a receive-expression must have at least one clause if no timeout part is specified.

See also: case_expr/2, clause/3, receive_expr/1, receive_expr_action/1, receive_expr_clauses/1, receive_expr_timeout/1.

receive_expr_action(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of action body subtrees of a `receive_expr` node. If `Node` represents "`receive C1; ...; Cn end`", this is the empty list.

See also: receive_expr/3.

receive_expr_clauses(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of clause subtrees of a `receive_expr` node.

See also: receive_expr/3.

receive_expr_timeout(Node::syntaxTree()) -> none | syntaxTree()

Returns the timeout subtree of a `receive_expr` node, if any. If `Node` represents "`receive C1; ...; Cn end`", `none` is returned. Otherwise, if `Node` represents "`receive C1; ...; Cn after Timeout -> ... end`", `Timeout` is returned.

See also: receive_expr/3.

record_access(Argument::syntaxTree(), Type::syntaxTree(), Field::syntaxTree()) -> syntaxTree()

Creates an abstract record field access expression. The result represents "`Argument#Type.Field`".

See also: record_access_argument/1, record_access_field/1, record_access_type/1, record_expr/3.

record_access_argument(Node::syntaxTree()) -> syntaxTree()

Returns the argument subtree of a `record_access` node.

See also: record_access/3.

record_access_field(Node::syntaxTree()) -> syntaxTree()

Returns the field subtree of a `record_access` node.

See also: record_access/3.

record_access_type(Node::syntaxTree()) -> syntaxTree()

Returns the type subtree of a `record_access` node.

See also: record_access/3.

record_expr(Type::syntaxTree(), Fields::[syntaxTree()]) -> syntaxTree()

Equivalent to *record_expr(none, Type, Fields)*.

record_expr(Argument::none | syntaxTree(), Type::syntaxTree(), Fields::[syntaxTree()]) -> syntaxTree()

Creates an abstract record expression. If `Fields` is `[F1, ..., Fn]`, then if `Argument` is `none`, the result represents "`#Type{F1, ..., Fn}`", otherwise it represents "`Argument#Type{F1, ..., Fn}`".

See also: record_access/3, *record_expr/2*, *record_expr_argument/1*, *record_expr_fields/1*, *record_expr_type/1*, *record_field/2*, *record_index_expr/2*.

record_expr_argument(Node::syntaxTree()) -> none | syntaxTree()

Returns the argument subtree of a `record_expr` node, if any. If `Node` represents "`#Type{...}`", `none` is returned. Otherwise, if `Node` represents "`Argument#Type{...}`", `Argument` is returned.

See also: record_expr/3.

record_expr_fields(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of field subtrees of a `record_expr` node.

See also: record_expr/3.

record_expr_type(Node::syntaxTree()) -> syntaxTree()

Returns the type subtree of a `record_expr` node.

See also: record_expr/3.

record_field(Name::syntaxTree()) -> syntaxTree()

Equivalent to *record_field(Name, none)*.

record_field(Name::syntaxTree(), Value::none | syntaxTree()) -> syntaxTree()

Creates an abstract record field specification. If `Value` is `none`, the result represents simply "`Name`", otherwise it represents "`Name = Value`".

See also: record_expr/3, *record_field_name/1*, *record_field_value/1*.

`record_field_name(Node::syntaxTree()) -> syntaxTree()`

Returns the name subtree of a `record_field` node.

See also: record_field/2.

`record_field_value(Node::syntaxTree()) -> none | syntaxTree()`

Returns the value subtree of a `record_field` node, if any. If `Node` represents "`Name`", `none` is returned. Otherwise, if `Node` represents "`Name = Value`", `Value` is returned.

See also: record_field/2.

`record_index_expr(Type::syntaxTree(), Field::syntaxTree()) -> syntaxTree()`

Creates an abstract record field index expression. The result represents "`#Type.Field`".

(Note: the function name `record_index/2` is reserved by the Erlang compiler, which is why that name could not be used for this constructor.)

See also: record_expr/3, *record_index_expr_field/1*, *record_index_expr_type/1*.

`record_index_expr_field(Node::syntaxTree()) -> syntaxTree()`

Returns the field subtree of a `record_index_expr` node.

See also: record_index_expr/2.

`record_index_expr_type(Node::syntaxTree()) -> syntaxTree()`

Returns the type subtree of a `record_index_expr` node.

See also: record_index_expr/2.

`remove_comments(Node::syntaxTree()) -> syntaxTree()`

Clears the associated comments of `Node`.

Note: This is equivalent to `set_precomments(set_postcomments(Node, []), [])`, but potentially more efficient.

See also: set_postcomments/2, *set_precomments/2*.

`revert(Node::syntaxTree()) -> syntaxTree()`

Returns an `erl_parse`-compatible representation of a syntax tree, if possible. If `Tree` represents a well-formed Erlang program or expression, the conversion should work without problems. Typically, *is_tree/1* yields `true` if conversion failed (i.e., the result is still an abstract syntax tree), and `false` otherwise.

The *is_tree/1* test is not completely foolproof. For a few special node types (e.g. `arity_qualifier`), if such a node occurs in a context where it is not expected, it will be left unchanged as a non-reverted subtree of the result. This can only happen if `Tree` does not actually represent legal Erlang code.

See also: erl_parse(3), *revert_forms/1*.

`revert_forms(Forms::forms()) -> [erl_parse()]`

Reverts a sequence of Erlang source code forms. The sequence can be given either as a `form_list` syntax tree (possibly nested), or as a list of "program form" syntax trees. If successful, the corresponding flat list of `erl_parse`-compatible syntax trees is returned (see *revert/1*). If some program form could not be reverted, `{error, Form}` is thrown. Standalone comments in the form sequence are discarded.

See also: form_list/1, is_form/1, revert/1.

`set_ann(Node::syntaxTree(), As::[term()]) -> syntaxTree()`

Sets the list of user annotations of `Node` to `Annotations`.

See also: add_ann/2, copy_ann/2, get_ann/1.

`set_attrs(Node::syntaxTree(), Attr::syntaxTreeAttributes()) -> syntaxTree()`

Sets the attributes of `Node` to `Attributes`.

See also: copy_attrs/2, get_attrs/1.

`set_pos(Node::syntaxTree(), Pos::term()) -> syntaxTree()`

Sets the position information of `Node` to `Pos`.

See also: copy_pos/2, get_pos/1.

`set_postcomments(Node::syntaxTree(), Cs::[syntaxTree()]) -> syntaxTree()`

Sets the post-comments of `Node` to `Comments`. `Comments` should be a possibly empty list of abstract comments, in top-down textual order

See also: add_postcomments/2, comment/2, copy_comments/2, get_postcomments/1, join_comments/2, remove_comments/1, set_precomments/2.

`set_precomments(Node::syntaxTree(), Cs::[syntaxTree()]) -> syntaxTree()`

Sets the pre-comments of `Node` to `Comments`. `Comments` should be a possibly empty list of abstract comments, in top-down textual order.

See also: add_precomments/2, comment/2, copy_comments/2, get_precomments/1, join_comments/2, remove_comments/1, set_postcomments/2.

`size_qualifier(Body::syntaxTree(), Size::syntaxTree()) -> syntaxTree()`

Creates an abstract size qualifier. The result represents "`Body:Size`".

See also: size_qualifier_argument/1, size_qualifier_body/1.

`size_qualifier_argument(Node::syntaxTree()) -> syntaxTree()`

Returns the argument subtree (the size) of a `size_qualifier` node.

See also: size_qualifier/2.

`size_qualifier_body(Node::syntaxTree()) -> syntaxTree()`

Returns the body subtree of a `size_qualifier` node.

See also: size_qualifier/2.

`string(String::string()) -> syntaxTree()`

Creates an abstract string literal. The result represents `"Text"` (including the surrounding double-quotes), where `Text` corresponds to the sequence of characters in `Value`, but not representing a *specific* string literal.

For example, the result of `string("x\ny")` represents any and all of `"x\ny"`, `"x\12y"`, `"x\012y"` and `"x\^Jy"`; see *char/1*.

See also: char/1, is_string/2, string_literal/1, string_literal/2, string_value/1.

`string_literal(Node::syntaxTree()) -> nonempty_string()`

Returns the literal string represented by a `string` node. This includes surrounding double-quote characters. Characters beyond 255 will be escaped.

See also: string/1.

`string_literal(Node::syntaxTree(), X2::encoding()) -> nonempty_string()`

Returns the literal string represented by a `string` node. This includes surrounding double-quote characters. Depending on the encoding characters beyond 255 will be escaped ('latin1') or copied as is ('utf8').

See also: string/1.

`string_value(Node::syntaxTree()) -> string()`

Returns the value represented by a `string` node.

See also: string/1.

`subtrees(T::syntaxTree()) -> [[syntaxTree()]]`

Returns the grouped list of all subtrees of a syntax tree. If `Node` is a leaf node (see *is_leaf/1*), this is the empty list, otherwise the result is always a nonempty list, containing the lists of subtrees of `Node`, in left-to-right order as they occur in the printed program text, and grouped by category. Often, each group contains only a single subtree.

Depending on the type of `Node`, the size of some groups may be variable (e.g., the group consisting of all the elements of a tuple), while others always contain the same number of elements - usually exactly one (e.g., the group containing the argument expression of a case-expression). Note, however, that the exact structure of the returned list (for a given node type) should in general not be depended upon, since it might be subject to change without notice.

The function *subtrees/1* and the constructor functions *make_tree/2* and *update_tree/2* can be a great help if one wants to traverse a syntax tree, visiting all its subtrees, but treat nodes of the tree in a uniform way in most or all cases. Using these functions makes this simple, and also assures that your code is not overly sensitive to extensions of the syntax tree data type, because any node types not explicitly handled by your code can be left to a default case.

For example:

```
postorder(F, Tree) ->
   F(case subtrees(Tree) of
       [] -> Tree;
       List -> update_tree(Tree,
                           [[postorder(F, Subtree)
                             || Subtree &lt;- Group]
                            || Group &lt;- List])
     end).
```

maps the function `F` on `Tree` and all its subtrees, doing a post-order traversal of the syntax tree. (Note the use of *update_tree/2* to preserve node attributes.) For a simple function like:

```
f(Node) ->
   case type(Node) of
       atom -> atom("a_" ++ atom_name(Node));
       _ -> Node
```

```
        end.
```

the call `postorder(fun f/1, Tree)` will yield a new representation of `Tree` in which all atom names have been extended with the prefix "a_", but nothing else (including comments, annotations and line numbers) has been changed.

See also: copy_attrs/2, is_leaf/1, make_tree/2, type/1.

`text(String::string()) -> syntaxTree()`

Creates an abstract piece of source code text. The result represents exactly the sequence of characters in `String`. This is useful in cases when one wants full control of the resulting output, e.g., for the appearance of floating-point numbers or macro definitions.

See also: text_string/1.

`text_string(Node::syntaxTree()) -> string()`

Returns the character sequence represented by a `text` node.

See also: text/1.

`tree(Type::atom()) -> #tree{}`

Equivalent to *tree(Type, [])*.

`tree(Type::atom(), Data::term()) -> #tree{}`

For special purposes only. Creates an abstract syntax tree node with type tag `Type` and associated data `Data`.

This function and the related *is_tree/1* and *data/1* provide a uniform way to extend the set of `erl_parse` node types. The associated data is any term, whose format may depend on the type tag.

Notes:

- Any nodes created outside of this module must have type tags distinct from those currently defined by this module; see *type/1* for a complete list.
- The type tag of a syntax tree node may also be used as a primary tag by the `erl_parse` representation; in that case, the selector functions for that node type *must* handle both the abstract syntax tree and the `erl_parse` form. The function `type(T)` should return the correct type tag regardless of the representation of `T`, so that the user sees no difference between `erl_syntax` and `erl_parse` nodes.

See also: data/1, is_tree/1, type/1.

`try_after_expr(Body::[syntaxTree()], After::[syntaxTree()]) -> syntaxTree()`

Equivalent to *try_expr(Body, [], [], After)*.

`try_expr(Body::[syntaxTree()], Handlers::[syntaxTree()]) -> syntaxTree()`

Equivalent to *try_expr(Body, [], Handlers)*.

`try_expr(Body::[syntaxTree()], Clauses::[syntaxTree()], Handlers::[syntaxTree()]) -> syntaxTree()`

Equivalent to *try_expr(Body, Clauses, Handlers, [])*.

try_expr(Body::[syntaxTree()], Clauses::[syntaxTree()], Handlers::[syntaxTree()], After::[syntaxTree()]) -> syntaxTree()

Creates an abstract try-expression. If `Body` is `[B1, ..., Bn]`, `Clauses` is `[C1, ..., Cj]`, `Handlers` is `[H1, ..., Hk]`, and `After` is `[A1, ..., Am]`, the result represents "`try B1, ..., Bn of C1; ...; Cj catch H1; ...; Hk after A1, ..., Am end`". More exactly, if each `Ci` represents "`(CPi) CGi -> CBi`", and each `Hi` represents "`(HPi) HGi -> HBi`", then the result represents "`try B1, ..., Bn of CP1 CG1 -> CB1; ...; CPj CGj -> CBj catch HP1 HG1 -> HB1; ...; HPk HGk -> HBk after A1, ..., Am end`"; see *case_expr/2*. If `Clauses` is the empty list, the `of ...` section is left out. If `After` is the empty list, the `after ...` section is left out. If `Handlers` is the empty list, and `After` is nonempty, the `catch ...` section is left out.

See also: case_expr/2, class_qualifier/2, clause/3, try_after_expr/2, try_expr/2, try_expr/3, try_expr_after/1, try_expr_body/1, try_expr_clauses/1, try_expr_handlers/1.

try_expr_after(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of "after" subtrees of a `try_expr` node.

See also: try_expr/4.

try_expr_body(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of body subtrees of a `try_expr` node.

See also: try_expr/4.

try_expr_clauses(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of case-clause subtrees of a `try_expr` node. If `Node` represents "`try Body catch H1; ...; Hn end`", the result is the empty list.

See also: try_expr/4.

try_expr_handlers(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of handler-clause subtrees of a `try_expr` node.

See also: try_expr/4.

tuple(List::[syntaxTree()]) -> syntaxTree()

Creates an abstract tuple. If `Elements` is `[X1, ..., Xn]`, the result represents "`{X1, ..., Xn}`".

Note: The Erlang language has distinct 1-tuples, i.e., `{X}` is always distinct from `X` itself.

See also: tuple_elements/1, tuple_size/1.

tuple_elements(Node::syntaxTree()) -> [syntaxTree()]

Returns the list of element subtrees of a `tuple` node.

See also: tuple/1.

tuple_size(Node::syntaxTree()) -> non_neg_integer()

Returns the number of elements of a `tuple` node.

Note: this is equivalent to `length(tuple_elements(Node))`, but potentially more efficient.

See also: tuple/1, tuple_elements/1.

type(Tree::syntaxTree()) -> atom()

Returns the type tag of `Node`. If `Node` does not represent a syntax tree, evaluation fails with reason `badarg`. Node types currently defined by this module are:

application arity_qualifier atom attribute
binary binary_field block_expr case_expr
catch_expr char class_qualifier clause
comment cond_expr conjunction disjunction
eof_marker error_marker float form_list
fun_expr function generator if_expr
implicit_fun infix_expr integer list
list_comp macro map_expr map_field_assoc
map_field_exact match_expr module_qualifier named_fun_expr
nil operator parentheses prefix_expr
receive_expr record_access record_expr record_field
record_index_expr size_qualifier string text
try_expr tuple underscore variable
warning_marker

The user may (for special purposes) create additional nodes with other type tags, using the *tree/2* function.

Note: The primary constructor functions for a node type should always have the same name as the node type itself.

See also: application/3, arity_qualifier/2, atom/1, attribute/2, binary/1, binary_field/2, block_expr/1, case_expr/2, catch_expr/1, char/1, class_qualifier/2, clause/3, comment/2, cond_expr/1, conjunction/1, disjunction/1, eof_marker/0, error_marker/1, float/1, form_list/1, fun_expr/1, function/2, generator/2, if_expr/1, implicit_fun/2, infix_expr/3, integer/1, list/2, list_comp/2, macro/2, map_expr/2, map_field_assoc/2, map_field_exact/2, match_expr/2, module_qualifier/2, named_fun_expr/2, nil/0, operator/1, parentheses/1, prefix_expr/2, receive_expr/3, record_access/3, record_expr/2, record_field/2, record_index_expr/2, size_qualifier/2, string/1, text/1, tree/2, try_expr/3, tuple/1, underscore/0, variable/1, warning_marker/1.

underscore() -> syntaxTree()

Creates an abstract universal pattern ("_"). The lexical representation is a single underscore character. Note that this is *not* a variable, lexically speaking.

See also: variable/1.

update_tree(Node::syntaxTree(), Groups::[[syntaxTree()]]) -> syntaxTree()

Creates a syntax tree with the same type and attributes as the given tree. This is equivalent to `copy_attrs(Node, make_tree(type(Node), Groups))`.

See also: copy_attrs/2, make_tree/2, type/1.

variable(Name::atom() | string()) -> syntaxTree()

Creates an abstract variable with the given name. `Name` may be any atom or string that represents a lexically valid variable name, but *not* a single underscore character; see *underscore/0*.

Note: no checking is done whether the character sequence represents a proper variable name, i.e., whether or not its first character is an uppercase Erlang character, or whether it does not contain control characters, whitespace, etc.

See also: underscore/0, variable_literal/1, variable_name/1.

variable_literal(Node::syntaxTree()) -> string()

Returns the name of a `variable` node as a string.

See also: variable/1.

variable_name(Node::syntaxTree()) -> atom()

Returns the name of a `variable` node as an atom.

See also: variable/1.

warning_marker(Warning::term()) -> syntaxTree()

Creates an abstract warning marker. The result represents an occurrence of a possible problem in the source code, with an associated Erlang I/O ErrorInfo structure given by `Error` (see module *io(3)* for details). Warning markers are regarded as source code forms, but have no defined lexical form.

Note: this is supported only for backwards compatibility with existing parsers and tools.

See also: eof_marker/0, *error_marker/1*, *is_form/1*, *warning_marker_info/1*.

warning_marker_info(Node::syntaxTree()) -> term()

Returns the ErrorInfo structure of a `warning_marker` node.

See also: warning_marker/1.

erl_syntax_lib

Erlang module

Support library for abstract Erlang syntax trees.

This module contains utility functions for working with the abstract data type defined in the module *erl_syntax*.

DATA TYPES

```
info_pair() = {key(), term()}
key() = attributes | errors | exports | functions | imports | module |
records | warnings
ordset(T) = ordset(T) (see module //stdlib/ordsets)
syntaxTree() = syntaxTree() (see module erl_syntax)
```

An abstract syntax tree. See the *erl_syntax* module for details.

Exports

`analyze_application(Node::syntaxTree()) -> FunctionName | Arity`

Types:

`FunctionName = {atom(), Arity} | {ModuleName, FunctionName}`

`Arity = integer()`

`ModuleName = atom()`

Returns the name of a called function. The result is a representation of the name of the applied function `F/A`, if `Node` represents a function application "F(X_1, ..., X_A)". If the function is not explicitly named (i.e., `F` is given by some expression), only the arity `A` is returned.

The evaluation throws `syntax_error` if `Node` does not represent a well-formed application expression.

See also: analyze_function_name/1.

`analyze_attribute(Node::syntaxTree()) -> preprocessor | {atom(), atom()}`

Analyzes an attribute node. If `Node` represents a preprocessor directive, the atom `preprocessor` is returned. Otherwise, if `Node` represents a module attribute "-Name...", a tuple `{Name, Info}` is returned, where `Info` depends on `Name`, as follows:

`{module, Info}`

where `Info = analyze_module_attribute(Node)`.

`{export, Info}`

where `Info = analyze_export_attribute(Node)`.

`{import, Info}`

where `Info = analyze_import_attribute(Node)`.

`{file, Info}`

where `Info = analyze_file_attribute(Node)`.

`{record, Info}`

where `Info = analyze_record_attribute(Node)`.

`{Name, Info}`

where `{Name, Info} = analyze_wild_attribute(Node)`.

The evaluation throws `syntax_error` if `Node` does not represent a well-formed module attribute.

See also: analyze_export_attribute/1, *analyze_file_attribute/1*, *analyze_import_attribute/1*, *analyze_module_attribute/1*, *analyze_record_attribute/1*, *analyze_wild_attribute/1*.

`analyze_export_attribute(Node::syntaxTree()) -> [FunctionName]`

Types:

`FunctionName = atom() | {atom(), integer()} | {ModuleName, FunctionName}`

`ModuleName = atom()`

Returns the list of function names declared by an export attribute. We do not guarantee that each name occurs at most once in the list. The order of listing is not defined.

The evaluation throws `syntax_error` if `Node` does not represent a well-formed export attribute.

See also: analyze_attribute/1.

`analyze_file_attribute(Node::syntaxTree()) -> {string(), integer()}`

Returns the file name and line number of a `file` attribute. The result is the pair `{File, Line}` if `Node` represents "`-file(File, Line).`".

The evaluation throws `syntax_error` if `Node` does not represent a well-formed `file` attribute.

See also: analyze_attribute/1.

`analyze_form(Node::syntaxTree()) -> {atom(), term()} | atom()`

Analyzes a "source code form" node. If `Node` is a "form" type (cf. `erl_syntax:is_form/1`), the returned value is a tuple `{Type, Info}` where `Type` is the node type and `Info` depends on `Type`, as follows:

`{attribute, Info}`

where `Info = analyze_attribute(Node)`.

`{error_marker, Info}`

where `Info = erl_syntax:error_marker_info(Node)`.

`{function, Info}`

where `Info = analyze_function(Node)`.

`{warning_marker, Info}`

where `Info = erl_syntax:warning_marker_info(Node)`.

For other types of forms, only the node type is returned.

The evaluation throws `syntax_error` if `Node` is not well-formed.

See also: analyze_attribute/1, *analyze_function/1*, *erl_syntax:error_marker_info/1*, *erl_syntax:is_form/1*, *erl_syntax:warning_marker_info/1*.

`analyze_forms(Forms) -> [{Key, term()}]`

Types:

`Forms = syntaxTree() | [syntaxTree()]`

Key = attributes | errors | exports | functions | imports | module | records | warnings

Analyzes a sequence of "program forms". The given `Forms` may be a single syntax tree of type `form_list`, or a list of "program form" syntax trees. The returned value is a list of pairs `{Key, Info}`, where each value of `Key` occurs at most once in the list; the absence of a particular key indicates that there is no well-defined value for that key.

Each entry in the resulting list contains the following corresponding information about the program forms:

`{attributes, Attributes}`

- `Attributes = [{atom(), term()}]`

`Attributes` is a list of pairs representing the names and corresponding values of all so-called "wild" attributes (as e.g. "`-compile(...)`") occurring in `Forms` (cf. `analyze_wild_attribute/1`). We do not guarantee that each name occurs at most once in the list. The order of listing is not defined.

`{errors, Errors}`

- `Errors = [term()]`

`Errors` is the list of error descriptors of all `error_marker` nodes that occur in `Forms`. The order of listing is not defined.

`{exports, Exports}`

- `Exports = [FunctionName]`
- `FunctionName = atom() | {atom(), integer()} | {ModuleName, FunctionName}`
- `ModuleName = atom()`

`Exports` is a list of representations of those function names that are listed by export declaration attributes in `Forms` (cf. `analyze_export_attribute/1`). We do not guarantee that each name occurs at most once in the list. The order of listing is not defined.

`{functions, Functions}`

- `Functions = [{atom(), integer()}]`

`Functions` is a list of the names of the functions that are defined in `Forms` (cf. `analyze_function/1`). We do not guarantee that each name occurs at most once in the list. The order of listing is not defined.

`{imports, Imports}`

- `Imports = [{Module, Names}]`
- `Module = atom()`
- `Names = [FunctionName]`
- `FunctionName = atom() | {atom(), integer()} | {ModuleName, FunctionName}`
- `ModuleName = atom()`

`Imports` is a list of pairs representing those module names and corresponding function names that are listed by import declaration attributes in `Forms` (cf. `analyze_import_attribute/1`), where each `Module` occurs at most once in `Imports`. We do not guarantee that each name occurs at most once in the lists of function names. The order of listing is not defined.

`{module, ModuleName}`

- `ModuleName = atom()`

`ModuleName` is the name declared by a module attribute in `Forms`. If no module name is defined in `Forms`, the result will contain no entry for the `module` key. If multiple module name declarations should occur, all but the first will be ignored.

`{records, Records}`

- `Records = [{atom(), Fields}]`

- `Fields = [{atom(), Default}]`
- `Default = none | syntaxTree()`

`Records` is a list of pairs representing the names and corresponding field declarations of all record declaration attributes occurring in `Forms`. For fields declared without a default value, the corresponding value for `Default` is the atom `none` (cf. `analyze_record_attribute/1`). We do not guarantee that each record name occurs at most once in the list. The order of listing is not defined.

`{warnings, Warnings}`

- `Warnings = [term()]`

`Warnings` is the list of error descriptors of all `warning_marker` nodes that occur in `Forms`. The order of listing is not defined.

The evaluation throws `syntax_error` if an ill-formed Erlang construct is encountered.

See also: analyze_export_attribute/1, *analyze_function/1*, *analyze_import_attribute/1*, *analyze_record_attribute/1*, *analyze_wild_attribute/1*, *erl_syntax:error_marker_info/1*, *erl_syntax:warning_marker_info/1*.

`analyze_function(Node::syntaxTree()) -> {atom(), integer()}`

Returns the name and arity of a function definition. The result is a pair `{Name, A}` if `Node` represents a function definition "`Name(<em>P_1</em>, ..., <em>P_A</em>) -> ...`".

The evaluation throws `syntax_error` if `Node` does not represent a well-formed function definition.

`analyze_function_name(Node::syntaxTree()) -> FunctionName`

Types:

`FunctionName = atom() | {atom(), integer()} | {ModuleName, FunctionName}`
`ModuleName = atom()`

Returns the function name represented by a syntax tree. If `Node` represents a function name, such as "`foo/1`" or "`bloggs:fred/2`", a uniform representation of that name is returned. Different nestings of arity and module name qualifiers in the syntax tree does not affect the result.

The evaluation throws `syntax_error` if `Node` does not represent a well-formed function name.

`analyze_implicit_fun(Node::syntaxTree()) -> FunctionName`

Types:

`FunctionName = atom() | {atom(), integer()} | {ModuleName, FunctionName}`
`ModuleName = atom()`

Returns the name of an implicit fun expression "`fun <em>F</em>`". The result is a representation of the function name `F`. (Cf. `analyze_function_name/1`.)

The evaluation throws `syntax_error` if `Node` does not represent a well-formed implicit fun.

See also: analyze_function_name/1.

`analyze_import_attribute(Node::syntaxTree()) -> {atom(), [FunctionName]} | atom()`

Types:

`FunctionName = atom() | {atom(), integer()} | {ModuleName, FunctionName}`
`ModuleName = atom()`

Returns the module name and (if present) list of function names declared by an import attribute. The returned value is an atom `Module` or a pair `{Module, Names}`, where `Names` is a list of function names declared as imported from the module named by `Module`. We do not guarantee that each name occurs at most once in `Names`. The order of listing is not defined.

The evaluation throws `syntax_error` if `Node` does not represent a well-formed import attribute.

See also: analyze_attribute/1.

analyze_module_attribute(Node::syntaxTree()) -> Name::atom() | {Name::atom(), Variables::[atom()]}

Returns the module name and possible parameters declared by a module attribute. If the attribute is a plain module declaration such as `-module(name)`, the result is the module name. If the attribute is a parameterized module declaration, the result is a tuple containing the module name and a list of the parameter variable names.

The evaluation throws `syntax_error` if `Node` does not represent a well-formed module attribute.

See also: analyze_attribute/1.

analyze_record_attribute(Node::syntaxTree()) -> {atom(), Fields}

Types:

```
Fields = [{atom(), none | syntaxTree()}]
```

Returns the name and the list of fields of a record declaration attribute. The result is a pair `{Name, Fields}`, if `Node` represents "`-record(Name, {...}).`", where `Fields` is a list of pairs `{Label, Default}` for each field "`Label`" or "`Label = <em>Default</em>`" in the declaration, listed in left-to-right order. If the field has no default-value declaration, the value for `Default` will be the atom `none`. We do not guarantee that each label occurs at most one in the list.

The evaluation throws `syntax_error` if `Node` does not represent a well-formed record declaration attribute.

See also: analyze_attribute/1, *analyze_record_field/1*.

analyze_record_expr(Node::syntaxTree()) -> {atom(), Info} | atom()

Types:

```
Info = {atom(), [{atom(), Value}]} | {atom(), atom()} | atom()
Value = none | syntaxTree()
```

Returns the record name and field name/names of a record expression. If `Node` has type `record_expr`, `record_index_expr` or `record_access`, a pair `{Type, Info}` is returned, otherwise an atom `Type` is returned. `Type` is the node type of `Node`, and `Info` depends on `Type`, as follows:

`record_expr`:

```
{atom(), [{atom(), Value}]}
```

`record_access`:

```
{atom(), atom()}
```

`record_index_expr`:

```
{atom(), atom()}
```

For a `record_expr` node, `Info` represents the record name and the list of descriptors for the involved fields, listed in the order they appear. (See `analyze_record_field/1` for details on the field descriptors). For a `record_access` node, `Info` represents the record name and the field name. For a `record_index_expr` node, `Info` represents the record name and the name field name.

The evaluation throws `syntax_error` if `Node` represents a record expression that is not well-formed.

See also: analyze_record_attribute/1, analyze_record_field/1.

`analyze_record_field(Node::syntaxTree()) -> {atom(), Value}`

Types:

`Value = none | syntaxTree()`

Returns the label and value-expression of a record field specifier. The result is a pair `{Label, Value}`, if `Node` represents "`Label = <em>Value</em>`" or "`Label`", where in the first case, `Value` is a syntax tree, and in the second case `Value` is `none`.

The evaluation throws `syntax_error` if `Node` does not represent a well-formed record field specifier.

See also: analyze_record_attribute/1, analyze_record_expr/1.

`analyze_wild_attribute(Node::syntaxTree()) -> {atom(), term()}`

Returns the name and value of a "wild" attribute. The result is the pair `{Name, Value}`, if `Node` represents "`-Name(Value)`".

Note that no checking is done whether `Name` is a reserved attribute name such as `module` or `export`: it is assumed that the attribute is "wild".

The evaluation throws `syntax_error` if `Node` does not represent a well-formed wild attribute.

See also: analyze_attribute/1.

`annotate_bindings(Tree::syntaxTree()) -> syntaxTree()`

Adds or updates annotations on nodes in a syntax tree. Equivalent to `annotate_bindings(Tree, Bindings)` where the top-level environment `Bindings` is taken from the annotation `{env, Bindings}` on the root node of `Tree`. An exception is thrown if no such annotation should exist.

See also: annotate_bindings/2.

`annotate_bindings(Tree::syntaxTree(), Bindings::ordset(atom())) -> syntaxTree()`

Adds or updates annotations on nodes in a syntax tree. `Bindings` specifies the set of bound variables in the environment of the top level node. The following annotations are affected:

- `{env, Vars}`, representing the input environment of the subtree.
- `{bound, Vars}`, representing the variables that are bound in the subtree.
- `{free, Vars}`, representing the free variables in the subtree.

`Bindings` and `Vars` are ordered-set lists (cf. module `ordsets`) of atoms representing variable names.

See also: ordsets(3), annotate_bindings/1.

`fold(F::Function, Start::term(), Tree::syntaxTree()) -> term()`

Types:

`Function = (syntaxTree(), term()) -> term()`

Folds a function over all nodes of a syntax tree. The result is the value of `Function(X1, Function(X2, ... Function(Xn, Start) ... ))`, where `[X1, X2, ..., Xn]` are the nodes of `Tree` in a post-order traversal.

See also: fold_subtrees/3, foldl_listlist/3.

fold_subtrees(F::Function, Start::term(), Tree::syntaxTree()) -> term()

Types:

```
Function = (syntaxTree(), term()) -> term()
```

Folds a function over the immediate subtrees of a syntax tree. This is similar to `fold/3`, but only on the immediate subtrees of `Tree`, in left-to-right order; it does not include the root node of `Tree`.

See also: fold/3.

foldl_listlist(F::Function, Start::term(), Ls::[[term()]]) -> term()

Types:

```
Function = (term(), term()) -> term()
```

Like `lists:foldl/3`, but over a list of lists.

See also: lists:foldl/3, *fold/3*.

function_name_expansions(Names::[Name]) -> [{ShortName, Name}]

Types:

```
Name = ShortName | {atom(), Name}
ShortName = atom() | {atom(), integer()}
```

Creates a mapping from corresponding short names to full function names. Names are represented by nested tuples of atoms and integers (cf. `analyze_function_name/1`). The result is a list containing a pair `{ShortName, Name}` for each element `Name` in the given list, where the corresponding `ShortName` is the rightmost-innermost part of `Name`. The list thus represents a finite mapping from unqualified names to the corresponding qualified names.

Note: the resulting list can contain more than one tuple `{ShortName, Name}` for the same `ShortName`, possibly with different values for `Name`, depending on the given list.

See also: analyze_function_name/1.

is_fail_expr(Tree::syntaxTree()) -> boolean()

Returns `true` if `Tree` represents an expression which never terminates normally. Note that the reverse does not apply. Currently, the detected cases are calls to `exit/1`, `throw/1`, `erlang:error/1` and `erlang:error/2`.

See also: erlang:error/1, *erlang:error/2*, *erlang:exit/1*, *erlang:throw/1*.

limit(Tree, Depth) -> syntaxTree()

Equivalent to `limit(Tree, Depth, Text)` using the text `"..."` as default replacement.

See also: limit/3, *erl_syntax:text/1*.

limit(Tree::syntaxTree(), Depth::integer(), Node::syntaxTree()) -> syntaxTree()

Limits a syntax tree to a specified depth. Replaces all non-leaf subtrees in `Tree` at the given `Depth` by `Node`. If `Depth` is negative, the result is always `Node`, even if `Tree` has no subtrees.

When a group of subtrees (as e.g., the argument list of an `application` node) is at the specified depth, and there are two or more subtrees in the group, these will be collectively replaced by `Node` even if they are leaf nodes. Groups of subtrees that are above the specified depth will be limited in size, as if each subsequent tree in the group were one level deeper than the previous. E.g., if `Tree` represents a list of integers "`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`", the result of `limit(Tree, 5)` will represent `[1, 2, 3, 4, ...]`.

The resulting syntax tree is typically only useful for pretty-printing or similar visual formatting.

See also: limit/2.

`map(F::Function, Tree::syntaxTree()) -> syntaxTree()`

Types:

`Function = (syntaxTree()) -> syntaxTree()`

Applies a function to each node of a syntax tree. The result of each application replaces the corresponding original node. The order of traversal is bottom-up.

See also: map_subtrees/2.

`map_subtrees(F::Function, Tree::syntaxTree()) -> syntaxTree()`

Types:

`Function = (Tree) -> Tree1`

Applies a function to each immediate subtree of a syntax tree. The result of each application replaces the corresponding original node.

See also: map/2.

`mapfold(F::Function, Start::term(), Tree::syntaxTree()) -> {syntaxTree(), term()}`

Types:

`Function = (syntaxTree(), term()) -> {syntaxTree(), term()}`

Combines map and fold in a single operation. This is similar to `map/2`, but also propagates an extra value from each application of the `Function` to the next, while doing a post-order traversal of the tree like `fold/3`. The value `Start` is passed to the first function application, and the final result is the result of the last application.

See also: fold/3, *map/2*.

`mapfold_subtrees(F::Function, Start::term(), Tree::syntaxTree()) -> {syntaxTree(), term()}`

Types:

`Function = (syntaxTree(), term()) -> {syntaxTree(), term()}`

Does a mapfold operation over the immediate subtrees of a syntax tree. This is similar to `mapfold/3`, but only on the immediate subtrees of `Tree`, in left-to-right order; it does not include the root node of `Tree`.

See also: mapfold/3.

`mapfoldl_listlist(F::Function, S::State, Ls::[[term()]]) -> {[[term()]], term()}`

Types:

`Function = (term(), term()) -> {term(), term()}`

Like `lists:mapfoldl/3`, but over a list of lists. The list of lists in the result has the same structure as the given list of lists.

new_variable_name(Used::set(atom())) -> atom()

Returns an atom which is not already in the set `Used`. This is equivalent to `new_variable_name(Function, Used)`, where `Function` maps a given integer `N` to the atom whose name consists of "`V`" followed by the numeral for `N`.

See also: new_variable_name/2.

new_variable_name(F::Function, Used::set(atom())) -> atom()

Types:

Function = (integer()) -> atom()

Returns a user-named atom which is not already in the set `Used`. The atom is generated by applying the given `Function` to a generated integer. Integers are generated using an algorithm which tries to keep the names randomly distributed within a reasonably small range relative to the number of elements in the set.

This function uses the module `random` to generate new keys. The seed it uses may be initialized by calling `random:seed/0` or `random:seed/3` before this function is first called.

See also: random(3), sets(3), new_variable_name/1.

new_variable_names(N::integer(), Used::set(atom())) -> [atom()]

Like `new_variable_name/1`, but generates a list of `N` new names.

See also: new_variable_name/1.

new_variable_names(N::integer(), F::Function, Used::set(atom())) -> [atom()]

Types:

Function = (integer()) -> atom()

Like `new_variable_name/2`, but generates a list of `N` new names.

See also: new_variable_name/2.

strip_comments(Tree::syntaxTree()) -> syntaxTree()

Removes all comments from all nodes of a syntax tree. All other attributes (such as position information) remain unchanged. Standalone comments in form lists are removed; any other standalone comments are changed into null-comments (no text, no indentation).

to_comment(Tree) -> syntaxTree()

Equivalent to *to_comment(Tree, "% ")*.

to_comment(Tree::syntaxTree(), Prefix::string()) -> syntaxTree()

Equivalent to `to_comment(Tree, Prefix, F)` for a default formatting function `F`. The default `F` simply calls `erl_prettypr:format/1`.

See also: to_comment/3, erl_prettypr:format/1.

to_comment(Tree::syntaxTree(), Prefix::string(), F::Printer) -> syntaxTree()

Types:

Printer = (syntaxTree()) -> string()

Transforms a syntax tree into an abstract comment. The lines of the comment contain the text for `Node`, as produced by the given `Printer` function. Each line of the comment is prefixed by the string `Prefix` (this does not include the initial "%" character of the comment line).

For example, the result of `to_comment(erl_syntax:abstract([a,b,c]))` represents

```
%% [a,b,c]
```

(cf. `to_comment/1`).

Note: the text returned by the formatting function will be split automatically into separate comment lines at each line break. No extra work is needed.

See also: to_comment/1, to_comment/2.

`variables(Tree::syntaxTree()) -> set(atom())`

Types:

`set(T) (see module //stdlib/sets)`

Returns the names of variables occurring in a syntax tree, The result is a set of variable names represented by atoms. Macro names are not included.

See also: sets(3).

erl_tidy

Erlang module

Tidies and pretty-prints Erlang source code, removing unused functions, updating obsolete constructs and function calls, etc.

Caveats: It is possible that in some intricate uses of macros, the automatic addition or removal of parentheses around uses or arguments could cause the resulting program to be rejected by the compiler; however, we have found no such case in existing code. Programs defining strange macros can usually not be read by this program, and in those cases, no changes will be made.

If you really, really want to, you may call it "Inga".

Disclaimer: The author accepts no responsibility for errors introduced in code that has been processed by the program. It has been reasonably well tested, but the possibility of errors remains. Keep backups of your original code safely stored, until you feel confident that the new, modified code can be trusted.

Exports

`dir() -> ok`

Equivalent to *dir("")*.

`dir(Dir) -> ok`

Equivalent to *dir(Dir, [])*.

`dir(Directory::filename(), Options::[term()]) -> ok`

Types:

`filename() (see module file)`

Tidies Erlang source files in a directory and its subdirectories.

Available options:

{follow_links, boolean()}

If the value is `true`, symbolic directory links will be followed. The default value is `false`.

{recursive, boolean()}

If the value is `true`, subdirectories will be visited recursively. The default value is `true`.

{regexp, string()}

The value denotes a regular expression (see module `re`). Tidying will only be applied to those regular files whose names match this pattern. The default value is `".*\\.erl$"`, which matches normal Erlang source file names.

{test, boolean()}

If the value is `true`, no files will be modified. The default value is `false`.

{verbose, boolean()}

If the value is `true`, progress messages will be output while the program is running, unless the `quiet` option is `true`. The default value when calling *dir/2* is `true`.

See the function *file/2* for further options.

See also: re(3), file/2.

file(Name) -> ok

Equivalent to *file(Name, [])*.

file(Name::filename(), Options::[term()]) -> ok

Tidies an Erlang source code file.

Available options are:

{backup_suffix, string()}

Specifies the file name suffix to be used when a backup file is created; the default value is `".bak"` (cf. the `backups` option).

{backups, boolean()}

If the value is `true`, existing files will be renamed before new files are opened for writing. The new names are formed by appending the string given by the `backup_suffix` option to the original name. The default value is `true`.

{dir, filename()}

Specifies the name of the directory in which the output file is to be written. By default, the current directory is used. If the value is an empty string, the current directory is used.

{outfile, filename()}

Specifies the name of the file (without suffix) to which the resulting source code is to be written. If this option is not specified, the `Name` argument is used.

{printer, Function}

- `Function = (syntaxTree()) -> string()`

Specifies a function for prettyprinting Erlang syntax trees. This is used for outputting the resulting module definition. The function is assumed to return formatted text for the given syntax tree, and should raise an exception if an error occurs. The default formatting function calls `erl_prettypr:format/2`.

{test, boolean()}

If the value is `true`, no files will be modified; this is typically most useful if the `verbose` flag is enabled, to generate reports about the program files without affecting them. The default value is `false`.

{stdout, boolean()}

If the value is `true`, instead of the file being written to disk it will be printed to stdout. The default value is `false`.

See the function `module/2` for further options.

See also: module/2, *erl_prettypr:format/2*.

module(Forms) -> syntaxTree()

Equivalent to *module(Forms, [])*.

module(Forms, Options::[term()]) -> syntaxTree()

Types:

Forms = syntaxTree() | [syntaxTree()]
syntaxTree() (see module erl_syntax)

Tidies a syntax tree representation of a module definition. The given `Forms` may be either a single syntax tree of type `form_list`, or a list of syntax trees representing "program forms". In either case, `Forms` must represent a single

complete module definition. The returned syntax tree has type `form_list` and represents a tidied-up version of the same source code.

Available options are:

{auto_export_vars, boolean()}

If the value is `true`, all matches "`{V1, ..., Vn} = E`" where `E` is a case-, if- or receive-expression whose branches all return n-tuples (or explicitly throw exceptions) will be rewritten to bind and export the variables `V1`, ..., `Vn` directly. The default value is `false`.

For example:

```
{X, Y} = case ... of
             ... -> {17, foo()};
             ... -> {42, bar()}
         end
```

will be rewritten to:

```
case ... of
    ... -> X = 17, Y = foo(), {X, Y};
    ... -> X = 42, Y = bar(), {X, Y}
end
```

{auto_list_comp, boolean()}

If the value is `true`, calls to `lists:map/2` and `lists:filter/2` will be rewritten using list comprehensions. The default value is `true`.

{file, string()}

Specifies the name of the file from which the source code was taken. This is only used for generation of error reports. The default value is the empty string.

{idem, boolean()}

If the value is `true`, all options that affect how the code is modified are set to "no changes". For example, to only update guard tests, and nothing else, use the options `[new_guard_tests, idem]`. (Recall that options closer to the beginning of the list have higher precedence.)

{keep_unused, boolean()}

If the value is `true`, unused functions will not be removed from the code. The default value is `false`.

{new_guard_tests, boolean()}

If the value is `true`, guard tests will be updated to use the new names, e.g. "`is_integer(X)`" instead of "`integer(X)`". The default value is `true`. See also `old_guard_tests`.

{no_imports, boolean()}

If the value is `true`, all import statements will be removed and calls to imported functions will be expanded to explicit remote calls. The default value is `false`.

{old_guard_tests, boolean()}

If the value is `true`, guard tests will be changed to use the old names instead of the new ones, e.g. "`integer(X)`" instead of "`is_integer(X)`". The default value is `false`. This option overrides the `new_guard_tests` option.

{quiet, boolean()}

If the value is `true`, all information messages and warning messages will be suppressed. The default value is `false`.

{rename, [{{atom(), atom(), integer()}, {atom(), atom()}}]}

The value is a list of pairs, associating tuples `{Module, Name, Arity}` with tuples `{NewModule, NewName}`, specifying renamings of calls to remote functions. By default, the value is the empty list.

The renaming affects only remote calls (also when disguised by import declarations); local calls within a module are not affected, and no function definitions are renamed. Since the arity cannot change, the new name is represented by `{NewModule, NewName}` only. Only calls matching the specified arity will match; multiple entries are necessary for renaming calls to functions that have the same module and function name, but different arities.

This option can also be used to override the default renaming of calls which use obsolete function names.

{verbose, boolean()}

If the value is `true`, progress messages will be output while the program is running, unless the `quiet` option is `true`. The default value is `false`.

igor

Erlang module

Igor: the Module Merger and Renamer.

The program Igor merges the source code of one or more Erlang modules into a single module, which can then replace the original set of modules. Igor is also able to rename a set of (possibly interdependent) modules, without joining them into a single module.

The main user interface consists of the functions *merge/3* and *rename/3*. See also the function *parse_transform/2*.

A note of warning: Igor cannot do anything about the case when the name of a remote function is passed to the built-in functions `apply` and `spawn` *unless* the module and function names are explicitly stated in the call, as in e.g. `apply(lists, reverse, [Xs])`. In all other cases, Igor leaves such calls unchanged, and warns the user that manual editing might be necessary.

Also note that Erlang records will be renamed as necessary to avoid non-equivalent definitions using the same record name. This does not work if the source code accesses the name field of such record tuples by `element/2` or similar methods. Always use the record syntax to handle record tuples, if possible.

Disclaimer: the author of this program takes no responsibility for the correctness of the produced output, or for any effects of its execution. In particular, the author may not be held responsible should Igor include the code of a deceased madman in the result.

For further information on Igors in general, see e.g. "Young Frankenstein", Mel Brooks, 1974, and "The Fifth Elephant", Terry Pratchett, 1999.

DATA TYPES

`stubDescriptor() = {ModuleName, Functions, [Attribute]}`

- `ModuleName = atom()`
- `Functions = [{FunctionName, {ModuleName, FunctionName}}]`
- `FunctionName = {atom(), integer()}`
- `Attribute = {atom(), term()}`

  A stub module descriptor contains the module name, a list of exported functions, and a list of module attributes. Each function is described by its name (which includes its arity), and the corresponding module and function that it calls. (The arities should always match.) The attributes are simply described by key-value pairs.

Exports

`create_stubs(Stubs::[stubDescriptor()], Options::[term()]) -> [string()]`

Creates stub module source files corresponding to the given stub descriptors. The returned value is the list of names of the created files. See `merge_sources/3` for more information about stub descriptors.

Options:

```
{backup_suffix, string()}
{backups, boolean()}
{printer, Function}
{stub_dir, filename()}
{suffix, string()}
{verbose, boolean()}
```

See `merge/3` for details on these options.

See also: merge/3, merge_sources/3.

`merge(Name::atom(), Files::[filename()]) -> [filename()]`

Equivalent to *merge(Name, Files, [])*.

`merge(Name::atom(), Files::[filename()], Options::[term()]) -> [filename()]`

Types:

`filename() (see module file)`

Merges source code files to a single file. `Name` specifies the name of the resulting module - not the name of the output file. `Files` is a list of file names and/or module names of source modules to be read and merged (see `merge_files/4` for details). All the input modules must be distinctly named.

The resulting source code is written to a file named "`<em>Name</em>.erl`" in the current directory, unless otherwise specified by the options `dir` and `outfile` described below.

Examples:

- given a module `m` in file "`m.erl`" which uses the standard library module `lists`, calling `igor:merge(m, [m, lists])` will create a new file "`m.erl` which contains the code from `m` and exports the same functions, and which includes the referenced code from the `lists` module. The original file will be renamed to "`m.erl.bak`".
- given modules `m1` and `m2`, in corresponding files, calling `igor:merge(m, [m1, m2])` will create a file "`m.erl`" which contains the code from `m1` and `m2` and exports the functions of `m1`.

Stub module files are created for those modules that are to be exported by the target module (see options `export`, `stubs` and `stub_dir`).

The function returns the list of file names of all created modules, including any automatically created stub modules. The file name of the target module is always first in the list.

Note: If you get a "syntax error" message when trying to merge files (and you know those files to be correct), then try the `preprocess` option. It typically means that your code contains too strange macros to be handled without actually performing the preprocessor expansions.

Options:

`{backup_suffix, string()}`

Specifies the file name suffix to be used when a backup file is created; the default value is `".bak"`.

`{backups, boolean()}`

If the value is `true`, existing files will be renamed before new files are opened for writing. The new names are formed by appending the string given by the `backup_suffix` option to the original name. The default value is `true`.

`{dir, filename()}`

Specifies the name of the directory in which the output file is to be written. An empty string is interpreted as the current directory. By default, the current directory is used.

`{outfile, filename()}`

Specifies the name of the file (without suffix) to which the resulting source code is to be written. By default, this is the same as the `Name` argument.

`{preprocess, boolean()}`

If the value is `true`, preprocessing will be done when reading the source code. See `merge_files/4` for details.

`{printer, Function}`

- `Function = (syntaxTree()) -> string()`

  Specifies a function for prettyprinting Erlang syntax trees. This is used for outputting the resulting module definition, as well as for creating stub files. The function is assumed to return formatted text for the given syntax tree, and should raise an exception if an error occurs. The default formatting function calls `erl_prettypr:format/2`.

`{stub_dir, filename()}`

Specifies the name of the directory to which any generated stub module files are written. The default value is `"stubs"`.

`{stubs, boolean()}`

If the value is `true`, stub module files will be automatically generated for all exported modules that do not have the same name as the target module. The default value is `true`.

`{suffix, string()}`

Specifies the suffix to be used for the output file names; the default value is `".erl"`.

See `merge_files/4` for further options.

See also: merge/2, merge_files/4.

`merge_files(Name::atom(), Files::[filename()], Options::[term()]) -> {syntaxTree(), [stubDescriptor()]}`

Equivalent to *merge_files(Name, [], Files, Options)*.

`merge_files(Name::atom(), Sources::[Forms], Files::[filename()], Options::[term()]) -> {syntaxTree(), [stubDescriptor()]}`

Types:

`Forms = syntaxTree() | [syntaxTree()]`

Merges source code files and syntax trees to a single syntax tree. This is a file-reading front end to `merge_sources/3`. `Name` specifies the name of the resulting module - not the name of the output file. `Sources` is a list of syntax trees and/or lists of "source code form" syntax trees, each entry representing a module definition. `Files` is a list of file names and/or module names of source modules to be read and included. All the input modules must be distinctly named.

If a name in `Files` is not the name of an existing file, Igor assumes it represents a module name, and tries to locate and read the corresponding source file. The parsed files are appended to `Sources` and passed on to `merge_sources/3`, i.e., entries in `Sources` are listed before entries read from files.

If no exports are listed by an `export` option (see `merge_sources/3` for details), then if `Name` is also the name of one of the input modules, that module will be exported; otherwise, the first listed module will be exported. Cf. the examples under `merge/3`.

The result is a pair `{Tree, Stubs}`, where `Tree` represents the source code that is the result of merging all the code in `Sources` and `Files`, and `Stubs` is a list of stub module descriptors (see `merge_sources/3` for details).

Options:

`{comments, boolean()}`

If the value is `true`, source code comments in the original files will be preserved in the output. The default value is `true`.

`{find_src_rules, [{string(), string()}]}`

Specifies a list of rules for associating object files with source files, to be passed to the function `filename:find_src/2`. This can be used to change the way Igor looks for source files. If this option is not specified, the default system rules are used. The first occurrence of this option completely overrides any later in the option list.

`{includes, [filename()]}`

Specifies a list of directory names for the Erlang preprocessor, if used, to search for include files (cf. the `preprocess` option). The default value is the empty list. The directory of the source file and the current directory are automatically appended to the list.

`{macros, [{atom(), term()}]}`

Specifies a list of "pre-defined" macro definitions for the Erlang preprocessor, if used (cf. the `preprocess` option). The default value is the empty list.

`{preprocess, boolean()}`

If the value is `false`, Igor will read source files without passing them through the Erlang preprocessor (`epp`), in order to avoid expansion of preprocessor directives such as `-include(...).`, `-define(...).` and `-ifdef(...)`, and macro calls such as `?LINE` and `?MY_MACRO(x, y)`. The default value is `false`, i.e., preprocessing is not done. (See the module `epp_dodger` for details.)

Notes: If a file contains too exotic definitions or uses of macros, it will not be possible to read it without preprocessing. Furthermore, Igor does not currently try to sort out multiple inclusions of the same file, or redefinitions of the same macro name. Therefore, when preprocessing is turned off, it may become necessary to edit the resulting source code, removing such re-inclusions and redefinitions.

See `merge_sources/3` for further options.

See also: epp_dodger, filename:find_src/2, merge/3, merge_files/3, merge_sources/3.

`merge_sources(Name::atom(), Sources::[Forms], Options::[term()]) -> {syntaxTree(), [stubDescriptor()]}`

Types:

`Forms = syntaxTree() | [syntaxTree()]`

Merges syntax trees to a single syntax tree. This is the main code merging "engine". `Name` specifies the name of the resulting module. `Sources` is a list of syntax trees of type `form_list` and/or lists of "source code form" syntax trees, each entry representing a module definition. All the input modules must be distinctly named.

Unless otherwise specified by the options, all modules are assumed to be at least "static", and all except the target module are assumed to be "safe". See the `static` and `safe` options for details.

If `Name` is also the name of one of the input modules, the code from that module will occur at the top of the resulting code, and no extra "header" comments will be added. In other words, the look of that module will be preserved.

The result is a pair `{Tree, Stubs}`, where `Tree` represents the source code that is the result of merging all the code in `Sources`, and `Stubs` is a list of stub module descriptors (see below).

`Stubs` contains one entry for each exported input module (cf. the `export` option), each entry describing a stub module that redirects calls of functions in the original module to the corresponding (possibly renamed) functions in the new module. The stub descriptors can be used to automatically generate stub modules; see `create_stubs/2`.

Options:

`{export, [atom()]}`

Specifies a list of names of input modules whose interfaces should be exported by the output module. A stub descriptor is generated for each specified module, unless its name is `Name`. If no modules are specified, then if

`Name` is also the name of an input module, that module will be exported; otherwise the first listed module in `Sources` will be exported. The default value is the empty list.

`{export_all, boolean()}`

If the value is `true`, this is equivalent to listing all of the input modules in the `export` option. The default value is `false`.

`{file_attributes, Preserve}`

- `Preserve = yes | comment | no`

If the value is `yes`, all file attributes `-file(...)` in the input sources will be preserved in the resulting code. If the value is `comment`, they will be turned into comments, but remain in their original positions in the code relative to the other source code forms. If the value is `no`, all file attributes will be removed from the code, unless they have attached comments, in which case they will be handled as in the `comment` case. The default value is `no`.

`{no_banner, boolean()}`

If the value is `true`, no banner comment will be added at the top of the resulting module, even if the target module does not have the same name as any of the input modules. Instead, Igor will try to preserve the look of the module whose code is at the top of the output. The default value is `false`.

`{no_headers, boolean()}`

If the value is `true`, no header comments will be added to the resulting module at the beginning of each section of code that originates from a particular input module. The default value is `false`, which means that section headers are normally added whenever more than two or more modules are merged.

`{no_imports, boolean()}`

If the value is `true`, all `-import(...)` declarations in the original code will be expanded in the result; otherwise, as much as possible of the original import declarations will be preserved. The default value is `false`.

`{notes, Notes}`

- `Notes = always | yes | no`

If the value is `yes`, comments will be inserted where important changes have been made in the code. If the value is `always`, *all* changes to the code will be commented. If the value is `no`, changes will be made without comments. The default value is `yes`.

`{redirect, [{atom(), atom()}]}`

Specifies a list of pairs of module names, representing a mapping from old names to new. *The set of old names may not include any of the names of the input modules.* All calls to the listed old modules will be rewritten to refer to the corresponding new modules. *The redirected calls will not be further processed, even if the new destination is in one of the input modules.* This option mainly exists to support module renaming; cf. `rename/3`. The default value is the empty list.

`{safe, [atom()]}`

Specifies a list of names of input modules such that calls to these "safe" modules may be turned into direct local calls, that do not test for code replacement. Typically, this can be done for e.g. standard library modules. If a module is "safe", it is per definition also "static" (cf. below). The list may be empty. By default, all involved modules *except the target module* are considered "safe".

`{static, [atom()]}`

Specifies a list of names of input modules which will be assumed never to be replaced (reloaded) unless the target module is also first replaced. The list may be empty. The target module itself (which may also be one of the input modules) is always regarded as "static", regardless of the value of this option. By default, all involved modules are assumed to be static.

`{tidy, boolean()}`

If the value is `true`, the resulting code will be processed using the `erl_tidy` module, which removes unused functions and does general code cleanup. (See `erl_tidy:module/2` for additional options.) The default value is `true`.

`{verbose, boolean()}`

If the value is `true`, progress messages will be output while the program is running; the default value is `false`.

Note: The distinction between "static" and "safe" modules is necessary in order not to break the semantics of dynamic code replacement. A "static" source module will not be replaced unless the target module also is. Now imagine a state machine implemented by placing the code for each state in a separate module, and suppose that we want to merge this into a single target module, marking all source modules as static. At each point in the original code where a call is made from one of the modules to another (i.e., the state transitions), code replacement is expected to be detected. Then, if we in the merged code do not check at these points if the *target* module (the result of the merge) has been replaced, we can not be sure in general that we will be able to do code replacement of the merged state machine - it could run forever without detecting the code change. Therefore, all such calls must remain remote-calls (detecting code changes), but may call the target module directly.

If we are sure that this kind of situation cannot ensue, we may specify the involved modules as "safe", and all calls between them will become local. Note that if the target module itself is specified as safe, "remote" calls to itself will be turned into local calls. This would destroy the code replacement properties of e.g. a typical server loop.

See also: create_stubs/2, rename/3, erl_tidy:module/2.

`parse_transform(Forms::[syntaxTree()], Options::[term()]) -> [syntaxTree()]`

Types:

`syntaxTree() (see module erl_syntax)`

Allows Igor to work as a component of the Erlang compiler. Including the term `{parse_transform, igor}` in the compile options when compiling an Erlang module (cf. `compile:file/2`), will call upon Igor to process the source code, allowing automatic inclusion of other source files. No files are created or overwritten when this function is used.

Igor will look for terms `{igor, List}` in the compile options, where `List` is a list of Igor-specific options, as follows:

`{files, [filename()]}`

The value specifies a list of source files to be merged with the file being compiled; cf. `merge_files/4`.

See `merge_files/4` for further options. Note, however, that some options are preset by this function and cannot be overridden by the user; in particular, all cosmetic features are turned off, for efficiency. Preprocessing is turned on.

See also: compile:file/2, merge_files/4.

`rename(Files::[filename()], Renamings) -> [string()]`

Equivalent to *rename(Files, Renamings, [])*.

`rename(Files::[filename()], Renamings, Options::[term()]) -> [string()]`

Types:

`Renamings = [{atom(), atom()}]`

Renames a set of possibly interdependent source code modules. `Files` is a list of file names of source modules to be processed. `Renamings` is a list of pairs of *module names*, representing a mapping from old names to new. The returned value is the list of output file names.

Each file in the list will be read and processed separately. For every file, each reference to some module M, such that there is an entry {M, M1} in Renamings, will be changed to the corresponding M1. Furthermore, if a file F defines module M, and there is an entry {M, M1} in Renamings, a new file named M1.erl will be created in the same directory as F, containing the source code for module M, renamed to M1. If M does not have an entry in Renamings, the module is not renamed, only updated, and the resulting source code is written to M.erl (typically, this overwrites the original file). The suffix option (see below) can be used to change the default ".erl" suffix for the generated files.

Stub modules will automatically be created (see the stubs and stub_dir options below) for each module that is renamed. These can be used to redirect any calls still using the old module names. The stub files are created in the same directory as the source file (typically overwriting the original file).

Options:

```
{backup_suffix, string()}
{backups, boolean()}
{printer, Function}
{stubs, boolean()}
{suffix, string()}
```

See merge/3 for details on these options.

```
{comments, boolean()}
{preprocess, boolean()}
```

See merge_files/4 for details on these options.

```
{no_banner, boolean()}
```

For the rename function, this option is true by default. See merge_sources/3 for details.

```
{tidy, boolean()}
```

For the rename function, this option is false by default. See merge_sources/3 for details.

```
{no_headers, boolean()}
{stub_dir, filename()}
```

These options are preset by the rename function and cannot be overridden by the user.

See merge_sources/3 for further options.

See also: merge/3, merge_files/4, merge_sources/3.

merl

Erlang module

Metaprogramming in Erlang. Merl is a more user friendly interface to the `erl_syntax` module, making it easy both to build new ASTs from scratch and to match and decompose existing ASTs. For details that are outside the scope of Merl itself, please see the documentation of *erl_syntax*.

Quick start

To enable the full power of Merl, your module needs to include the Merl header file:

```
-include_lib("syntax_tools/include/merl.hrl").
```

Then, you can use the `?Q(Text)` macros in your code to create ASTs or match on existing ASTs. For example:

```
Tuple = ?Q("{foo, 42}"),
?Q("{foo, _@Number}") = Tuple,
Call = ?Q("foo:bar(_@Number)")
```

Calling `merl:print(Call)` will then print the following code:

```
foo:bar(42)
```

The `?Q` macros turn the quoted code fragments into ASTs, and lifts metavariables such as `_@Tuple` and `_@Number` to the level of your Erlang code, so you can use the corresponding Erlang variables `Tuple` and `Number` directly. This is the most straightforward way to use Merl, and in many cases it's all you need.

You can even write case switches using `?Q` macros as patterns. For example:

```
case AST of
    ?Q("{foo, _@Foo}") -> handle(Foo);
    ?Q("{bar, _@Bar}") when erl_syntax:is_integer(Bar) -> handle(Bar);
    _ -> handle_default()
end
```

These case switches only allow `?Q(...)` or _ as clause patterns, and the guards may contain any expressions, not just Erlang guard expressions.

If the macro `MERL_NO_TRANSFORM` is defined before the `merl.hrl` header file is included, the parse transform used by Merl will be disabled, and in that case, the match expressions `?Q(...) = ...`, case switches using `?Q(...)` patterns, and automatic metavariables like `_@Tuple` cannot be used in your code, but the Merl macros and functions still work. To do metavariable substitution, you need to use the `?Q(Text, Map)` macro, e.g.:

```
Tuple = ?Q("{foo, _@bar, _@baz}", [{bar, Bar}, {baz,Baz}])
```

The text given to a `?Q(Text)` macro can be either a single string, or a list of strings. The latter is useful when you need to split a long expression over multiple lines, e.g.:

```
?Q(["case _@Expr of",
    "  {foo, X} -> f(X);",
    "  {bar, X} -> g(X)",
    "  _ -> h(X)"
```

```
        "end"])
```

If there is a syntax error somewhere in the text (like the missing semicolon in the second clause above) this allows Merl to generate an error message pointing to the exact line in your source code. (Just remember to comma-separate the strings in the list, otherwise Erlang will concatenate the string fragments as if they were a single string.)

Metavariable syntax

There are several ways to write a metavariable in your quoted code:

- Atoms starting with `@`, for example `'@foo'` or `'@Foo'`
- Variables starting with `_@`, for example `_@bar` or `_@Bar`
- Strings starting with `"'@`, for example `"'@File"`
- Integers starting with 909, for example `9091` or `909123`

Following the prefix, one or more `_` or `0` characters may be used to indicate "lifting" of the variable one or more levels, and after that, a `@` or `9` character indicates a glob metavariable (matching zero or more elements in a sequence) rather than a normal metavariable. For example:

- `'@_foo'` is lifted one level, and `_@__foo` is lifted two levels
- `_@@bar` is a glob variable, and `_@_@bar` is a lifted glob variable
- `90901` is a lifted variable,`90991` is a glob variable, and `9090091` is a glob variable lifted two levels

(Note that the last character in the name is never considered to be a lift or glob marker, hence, `_@__` and `90900` are only lifted one level, not two. Also note that globs only matter for matching; when doing substitutions, a non-glob variable can be used to inject a sequence of elements, and vice versa.)

If the name after the prefix and any lift and glob markers is `_` or `0`, the variable is treated as an anonymous catch-all pattern in matches. For example, `_@_`, `_@@_`, `_@__`, or even `_@__@_`.

Finally, if the name without any prefixes or lift/glob markers begins with an uppercase character, as in `_@Foo` or `_@_@Foo`, it will become a variable on the Erlang level, and can be used to easily deconstruct and construct syntax trees:

```
    case Input of
        ?Q("{foo, _@Number}") -> ?Q("foo:bar(_@Number)");
        ...
```

We refer to these as "automatic metavariables". If in addition the name ends with `@`, as in `_@Foo@`, the value of the variable as an Erlang term will be automatically converted to the corresponding abstract syntax tree when used to construct a larger tree. For example, in:

```
    Bar = {bar, 42},
    Foo = ?Q("{foo, _@Bar@}")
```

(where Bar is just some term, not a syntax tree) the result `Foo` will be a syntax tree representing `{foo, {bar, 42}}`. This avoids the need for temporary variables in order to inject data, as in

```
    TmpBar = erl_syntax:abstract(Bar),
    Foo = ?Q("{foo, _@TmpBar}")
```

If the context requires an integer rather than a variable, an atom, or a string, you cannot use the uppercase convention to mark an automatic metavariable. Instead, if the integer (without the `909`-prefix and lift/glob markers) ends in a `9`, the integer will become an Erlang-level variable prefixed with `Q`, and if it ends with `99` it will also be automatically abstracted. For example, the following will increment the arity of the exported function f:

```
case Form of
    ?Q("-export([f/90919]).") ->
        Q2 = erl_syntax:concrete(Q1) + 1,
        ?Q("-export([f/909299]).");
    ...
```

When to use the various forms of metavariables

Merl can only parse a fragment of text if it follows the basic syntactical rules of Erlang. In most places, a normal Erlang variable can be used as metavariable, for example:

```
?Q("f(_@Arg)") = Expr
```

but if you want to match on something like the name of a function, you have to use an atom as metavariable:

```
?Q("'@Name'() -> _@@_." = Function
```

(note the anonymous glob variable `_@@_` to ignore the function body).

In some contexts, only a string or an integer is allowed. For example, the directive `-file(Name, Line)` requires that `Name` is a string literal and `Line` an integer literal:

```
?Q("-file(\"'@File\", 9090).") = ?Q("-file(\"foo.erl\", 42).")).
```

This will extract the string literal `"foo.erl"` into the variable `Foo`. Note the use of the anonymous variable `9090` to ignore the line number. To match and also bind a metavariable that must be an integer literal, we can use the convention of ending the integer with a 9, turning it into a Q-prefixed variable on the Erlang level (see the previous section).

Globs

Whenever you want to match out a number of elements in a sequence (zero or more) rather than a fixed set of elements, you need to use a glob. For example:

```
?Q("{_@@Elements}") = ?Q({a, b, c})
```

will bind Elements to the list of individual syntax trees representing the atoms `a`, `b`, and `c`. This can also be used with static prefix and suffix elements in the sequence. For example:

```
?Q("{a, b, _@@Elements}") = ?Q({a, b, c, d})
```

will bind Elements to the list of the `c` and `d` subtrees, and

```
?Q("{_@@Elements, c, d}") = ?Q({a, b, c, d})
```

will bind Elements to the list of the `a` and `b` subtrees. You can even use plain metavariables in the prefix or suffix:

```
?Q("{_@First, _@@Rest}") = ?Q({a, b, c})
```

or

```
?Q("{_@@_, _@Last}") = ?Q({a, b, c})
```

(ignoring all but the last element). You cannot however have two globs as part of the same sequence.

Lifted metavariables

In some cases, the Erlang syntax rules make it impossible to place a metavariable directly where you would like it. For example, you cannot write:

```
?Q("-export([_@@Name]).")
```

to match out all name/arity pairs in the export list, or to insert a list of exports in a declaration, because the Erlang parser only allows elements on the form `A/I` (where `A` is an atom and `I` an integer) in the export list. A variable like the above is not allowed, but neither is a single atom or integer, so `'@@Name'` or `909919` wouldn't work either.

What you have to do in such cases is to write your metavariable in a syntactically valid position, and use lifting markers to denote where it should really apply, as in:

```
?Q("-export(['@_@Name'/0]).")
```

This causes the variable to be lifted (after parsing) to the next higher level in the syntax tree, replacing that entire subtree. In this case, the `'@_@Name'/0` will be replaced with `'@@Name'`, and the `/0` part was just used as dummy notation and will be discarded.

You may even need to apply lifting more than once. To match the entire export list as a single syntax tree, you can write:

```
?Q("-export(['@__Name'/0]).")
```

using two underscores, but with no glob marker this time. This will make the entire `['@__Name'/0]` part be replaced with `'@Name'`.

Sometimes, the tree structure of a code fragment isn't very obvious, and parts of the structure may be invisible when printed as source code. For instance, a simple function definition like the following:

```
zero() -> 0.
```

consists of the name (the atom `zero`), and a list of clauses containing the single clause `() -> 0`. The clause consists of an argument list (empty), a guard (empty), and a body (which is always a list of expressions) containing the single expression `0`. This means that to match out the name and the list of clauses of any function, you'll need to use a pattern like `?Q("'@Name'() -> _@_@Body.")`, using a dummy clause whose body is a glob lifted one level.

To visualize the structure of a syntax tree, you can use the function `merl:show(T)`, which prints a summary. For example, entering

```
merl:show(merl:quote("inc(X, Y) when Y > 0 -> X + Y."))
```

in the Erlang shell will print the following (where the `+` signs separate groups of subtrees on the same level):

```
function: inc(X, Y) when ... -> X + Y.
  atom: inc
  +
  clause: (X, Y) when ... -> X + Y
    variable: X
    variable: Y
    +
    disjunction: Y > 0
      conjunction: Y > 0
```

```
            infix_expr: Y > 0
              variable: Y
              +
              operator: >
              +
              integer: 0
        +
        infix_expr: X + Y
          variable: X
          +
          operator: +
          +
          variable: Y
```

This shows another important non-obvious case: a clause guard, even if it's as simple as `Y > 0`, always consists of a single disjunction of one or more conjunctions of tests, much like a tuple of tuples. Thus:

- `"when _@Guard ->"` will only match a guard with exactly one test
- `"when _@@Guard ->"` will match a guard with one or more comma-separated tests (but no semicolons), binding `Guard` to the list of tests
- `"when _@_Guard ->"` will match just like the previous pattern, but binds `Guard` to the conjunction subtree
- `"when _@_@Guard ->"` will match an arbitrary nonempty guard, binding `Guard` to the list of conjunction subtrees
- `"when _@__Guard ->"` will match like the previous pattern, but binds `Guard` to the whole disjunction subtree
- and finally, `"when _@__@Guard ->"` will match any clause, binding `Guard` to `[]` if the guard is empty and to `[Disjunction]` otherwise

Thus, the following pattern matches all possible clauses:

```
    "(_@Args) when _@__@Guard -> _@Body"
```

DATA TYPES

```
default_action() = () -> any()
env() = [{Key::id(), pattern_or_patterns()}]
guard_test() = (env()) -> boolean()
guarded_action() = switch_action() | {guard_test(), switch_action()}
guarded_actions() = guarded_action() | [guarded_action()]
id() = atom() | integer()
location() = location() (see module erl_anno)
pattern() = tree() | template()
pattern_or_patterns() = pattern() | [pattern()]
switch_action() = (env()) -> any()
switch_clause() = {pattern_or_patterns(), guarded_actions()} |
{pattern_or_patterns(), guard_test(), switch_action()} | default_action()
template() = tree() | {id()} | {'*', id()} | {template, atom(), term(),
[[template()]]}
template_or_templates() = template() | [template()]
text() = string() | binary() | [string()] | [binary()]
tree() = syntaxTree() (see module erl_syntax)
tree_or_trees() = tree() | [tree()]
```

Exports

alpha(Trees::pattern_or_patterns(), Env::[{id(), id()}]) -> template_or_templates()

Alpha converts a pattern (renames variables). Similar to tsubst/1, but only renames variables (including globs).

See also: tsubst/2.

compile(Code) -> term()

Equivalent to *compile(Code, []).*

compile(Code, Options) -> term()

Compile a syntax tree or list of syntax trees representing a module into a binary BEAM object.

See also: compile/1, compile_and_load/2.

compile_and_load(Code) -> term()

Equivalent to *compile_and_load(Code, []).*

compile_and_load(Code, Options) -> term()

Compile a syntax tree or list of syntax trees representing a module and load the resulting module into memory.

See also: compile/2, compile_and_load/1.

match(Patterns::pattern_or_patterns(), Trees::tree_or_trees()) -> {ok, env()} | error

Match a pattern against a syntax tree (or patterns against syntax trees) returning an environment mapping variable names to subtrees; the environment is always sorted on keys. Note that multiple occurrences of metavariables in the pattern is not allowed, but is not checked.

See also: switch/2, template/1.

meta_template(Templates::template_or_templates()) -> tree_or_trees()

Turn a template into a syntax tree representing the template. Meta-variables in the template are turned into normal Erlang variables if their names (after the metavariable prefix characters) begin with an uppercase character. E.g., `_@Foo` in the template becomes the variable `Foo` in the meta-template. Furthermore, variables ending with `@` are automatically wrapped in a call to merl:term/1, so e.g. `_@Foo@ in the template becomes` `` `merl:term(Foo) `` in the meta-template.

print(Ts) -> term()

Pretty-print a syntax tree or template to the standard output. This is a utility function for development and debugging.

qquote(Text::text(), Env::env()) -> tree_or_trees()

Parse text and substitute meta-variables.

qquote(StartPos::location(), Text::text(), Env::env()) -> tree_or_trees()

Parse text and substitute meta-variables. Takes an initial scanner starting position as first argument.

The macro `?Q(Text, Env)` expands to `merl:qquote(?LINE, Text, Env)`.

See also: quote/2.

quote(Text::text()) -> tree_or_trees()

Parse text.

quote(StartPos::location(), Text::text()) -> tree_or_trees()

Parse text. Takes an initial scanner starting position as first argument.

The macro `?Q(Text)` expands to `merl:quote(?LINE, Text, Env)`.

See also: quote/1.

show(Ts) -> term()

Print the structure of a syntax tree or template to the standard output. This is a utility function for development and debugging.

subst(Trees::pattern_or_patterns(), Env::env()) -> tree_or_trees()

Substitute metavariables in a pattern or list of patterns, yielding a syntax tree or list of trees as result. Both for normal metavariables and glob metavariables, the substituted value may be a single element or a list of elements. For example, if a list representing `1, 2, 3` is substituted for `var` in either of `[foo, _@var, bar]` or `[foo, _@var, bar]`, the result represents `[foo, 1, 2, 3, bar]`.

switch(Trees::tree_or_trees(), Cs::[switch_clause()]) -> any()

Match against one or more clauses with patterns and optional guards.

Note that clauses following a default action will be ignored.

See also: match/2.

template(Trees::pattern_or_patterns()) -> template_or_templates()

Turn a syntax tree or list of trees into a template or templates. Templates can be instantiated or matched against, and reverted back to normal syntax trees using *tree/1*. If the input is already a template, it is not modified further.

See also: match/2, subst/2, tree/1.

template_vars(Template::template_or_templates()) -> [id()]

Return an ordered list of the metavariables in the template.

term(Term::term()) -> tree()

Create a syntax tree for a constant term.

tree(Templates::template_or_templates()) -> tree_or_trees()

Revert a template to a normal syntax tree. Any remaining metavariables are turned into `@`-prefixed atoms or `909`-prefixed integers.

See also: template/1.

`tsubst(Trees::pattern_or_patterns(), Env::env()) -> template_or_templates()`

Like subst/2, but does not convert the result from a template back to a tree. Useful if you want to do multiple separate substitutions.

See also: subst/2, tree/1.

`var(Name::atom()) -> tree()`

Create a variable.

merl_transform

Erlang module

Parse transform for merl. Enables the use of automatic metavariables and using quasi-quotes in matches and case switches. Also optimizes calls to functions in `merl` by partially evaluating them, turning strings to templates, etc., at compile-time.

Using `-include_lib("syntax_tools/include/merl.hrl").` enables this transform, unless the macro `MERL_NO_TRANSFORM` is defined first.

Exports

`parse_transform(Forms, Options) -> term()`

prettypr

Erlang module

A generic pretty printer library. This module uses a strict-style context passing implementation of John Hughes algorithm, described in "The design of a Pretty-printing Library". The paragraph-style formatting, empty documents, floating documents, and null strings are my own additions to the algorithm.

To get started, you should read about the *document()* data type; the main constructor functions: *text/1*, *above/2*, *beside/2*, *nest/2*, *sep/1*, and *par/2*; and the main layout function *format/3*.

If you simply want to format a paragraph of plain text, you probably want to use the *text_par/2* function, as in the following example:

```
prettypr:format(prettypr:text_par("Lorem ipsum dolor sit amet"), 20)
```

DATA TYPES

`document()`

An abstract character-based "document" representing a number of possible layouts, which can be processed to produce a single concrete layout. A concrete layout can then be rendered as a sequence of characters containing linebreaks, which can be passed to a printer or terminal that uses a fixed-width font.

For example, a document `sep([text("foo"), text("bar")])` represents the two layouts

```
foo bar
```

and

```
foo
bar
```

Which layout is chosen depends on the available horizontal space. When processing a document, the main parameters are the *paper width* and the *line width* (also known as the "ribbon width"). In the resulting layout, no text should be printed beyond the paper width (which by default is 80 characters) as long as it can be avoided, and each single line of text (its indentation not counted, hence "ribbon") should preferably be no wider than the specified line width (which by default is 65).

Documents can be joined into a single new document using the constructor functions of this module. Note that the new document often represents a larger number of possible layouts than just the sum of the components.

Exports

`above(D1::document(), D2::document()) -> document()`

Concatenates documents vertically. Returns a document representing the concatenation of the documents `D1` and `D2` such that the first line of `D2` follows directly below the last line of `D1`, and the first character of `D2` is in the same horizontal column as the first character of `D1`, in all possible layouts.

Examples:

```
ab  cd  =>  ab
            cd
```

```
                abc
abc   fgh  =>   de
 de    ij      fgh
                ij
```

beside(D1::document(), D2::document()) -> document()

Concatenates documents horizontally. Returns a document representing the concatenation of the documents `D1` and `D2` such that the last character of `D1` is horizontally adjacent to the first character of `D2`, in all possible layouts. (Note: any indentation of `D2` is lost.)

Examples:

```
ab  cd  =>  abcd

ab  ef      ab
cd  gh  =>  cdef
              gh
```

best(D::document(), PaperWidth::integer(), LineWidth::integer()) -> empty | document()

Selects a "best" layout for a document, creating a corresponding fixed-layout document. If no layout could be produced, the atom `empty` is returned instead. For details about `PaperWidth` and `LineWidth`, see *format/3*. The function is idempotent.

One possible use of this function is to compute a fixed layout for a document, which can then be included as part of a larger document. For example:

```
above(text("Example:"), nest(8, best(D, W - 12, L - 6)))
```

will format `D` as a displayed-text example indented by 8, whose right margin is indented by 4 relative to the paper width `W` of the surrounding document, and whose maximum individual line length is shorter by 6 than the line length `L` of the surrounding document.

This function is used by the *format/3* function to prepare a document before being laid out as text.

break(D::document()) -> document()

Forces a line break at the end of the given document. This is a utility function; see *empty/0* for details.

empty() -> document()

Yields the empty document, which has neither height nor width. (`empty` is thus different from an empty *text* string, which has zero width but height 1.)

Empty documents are occasionally useful; in particular, they have the property that `above(X, empty())` will force a new line after `X` without leaving an empty line below it; since this is a common idiom, the utility function *break/1* will place a given document in such a context.

See also: text/1.

floating(D::document()) -> document()

Equivalent to *floating(D, 0, 0)*.

floating(D::document(), Hp::integer(), Vp::integer()) -> document()

Creates a "floating" document. The result represents the same set of layouts as `D`; however, a floating document may be moved relative to other floating documents immediately beside or above it, according to their relative horizontal and vertical priorities. These priorities are set with the `Hp` and `Vp` parameters; if omitted, both default to zero.

Notes: Floating documents appear to work well, but are currently less general than you might wish, losing effect when embedded in certain contexts. It is possible to nest floating-operators (even with different priorities), but the effects may be difficult to predict. In any case, note that the way the algorithm reorders floating documents amounts to a "bubblesort", so don't expect it to be able to sort large sequences of floating documents quickly.

follow(D1::document(), D2::document()) -> document()

Equivalent to *follow(D1, D2, 0)*.

follow(D1::document(), D2::document(), Offset::integer()) -> document()

Separates two documents by either a single space, or a line break and intentation. In other words, one of the layouts

```
    abc def
```

or

```
    abc
     def
```

will be generated, using the optional offset in the latter case. This is often useful for typesetting programming language constructs.

This is a utility function; see *par/2* for further details.

See also: follow/2.

format(D::document()) -> string()

Equivalent to *format(D, 80)*.

format(D::document(), PaperWidth::integer()) -> string()

Equivalent to *format(D, PaperWidth, 65)*.

format(D::document(), PaperWidth::integer(), LineWidth::integer()) -> string()

Computes a layout for a document and returns the corresponding text. See *document()* for further information. Throws `no_layout` if no layout could be selected.

`PaperWidth` specifies the total width (in character positions) of the field for which the text is to be laid out. `LineWidth` specifies the desired maximum width (in number of characters) of the text printed on any single line, disregarding leading and trailing white space. These parameters need to be properly balanced in order to produce good layouts. By default, `PaperWidth` is 80 and `LineWidth` is 65.

See also: best/3.

`nest(N::integer(), D::document()) -> document()`

Indents a document a number of character positions to the right. Note that `N` may be negative, shifting the text to the left, or zero, in which case `D` is returned unchanged.

`null_text(Characters::string()) -> document()`

Similar to *text/1*, but the result is treated as having zero width. This is regardless of the actual length of the string. Null text is typically used for markup, which is supposed to have no effect on the actual layout.

The standard example is when formatting source code as HTML to be placed within `<pre>...</pre>` markup, and using e.g. `<i>` and `<b>` to make parts of the source code stand out. In this case, the markup does not add to the width of the text when viewed in an HTML browser, so the layout engine should simply pretend that the markup has zero width.

See also: empty/0, *text/1*.

`par(Docs::[document()]) -> document()`

Equivalent to *par(Ds, 0)*.

`par(Docs::[document()], Offset::integer()) -> document()`

Arranges documents in a paragraph-like layout. Returns a document representing all possible left-aligned paragraph-like layouts of the (nonempty) sequence `Docs` of documents. Elements in `Docs` are separated horizontally by a single space character and vertically with a single line break. All lines following the first (if any) are indented to the same left column, whose indentation is specified by the optional `Offset` parameter relative to the position of the first element in `Docs`. For example, with an offset of -4, the following layout can be produced, for a list of documents representing the numbers 0 to 15:

```
    0 1 2 3
4 5 6 7 8 9
10 11 12 13
14 15
```

or with an offset of +2:

```
0 1 2 3 4 5 6
  7 8 9 10 11
  12 13 14 15
```

The utility function *text_par/2* can be used to easily transform a string of text into a `par` representation by splitting it into words.

Note that whenever a document in `Docs` contains a line break, it will be placed on a separate line. Thus, neither a layout such as

```
ab cd
   ef
```

nor

```
ab
cd ef
```

will be generated. However, a useful idiom for making the former variant possible (when wanted) is `beside(par([D1, text("")], N), D2)` for two documents `D1` and `D2`. This will break the line between `D1` and `D2` if `D1` contains a line break (or if otherwise necessary), and optionally further indent `D2` by `N` character positions. The utility function *follow/3* creates this context for two documents `D1` and `D2`, and an optional integer `N`.

See also: par/1, *text_par/2*.

`sep(Docs::[document()]) -> document()`

Arranges documents horizontally or vertically, separated by whitespace. Returns a document representing two alternative layouts of the (nonempty) sequence `Docs` of documents, such that either all elements in `Docs` are concatenated horizontally, and separated by a space character, or all elements are concatenated vertically (without extra separation).

Note: If some document in `Docs` contains a line break, the vertical layout will always be selected.

Examples:

```
                                ab
ab  cd  ef  =>  ab cd ef  |     cd
                                ef

ab           ab
cd  ef  =>   cd
             ef
```

See also: par/2.

`text(Characters::string()) -> document()`

Yields a document representing a fixed, unbreakable sequence of characters. The string should contain only *printable* characters (tabs allowed but not recommended), and *not* newline, line feed, vertical tab, etc. A tab character (`\t`) is interpreted as padding of 1-8 space characters to the next column of 8 characters *within the string*.

See also: empty/0, *null_text/1*, *text_par/2*.

`text_par(Text::string()) -> document()`

Equivalent to *text_par(Text, 0)*.

`text_par(Text::string(), Indentation::integer()) -> document()`

Yields a document representing paragraph-formatted plain text. The optional `Indentation` parameter specifies the extra indentation of the first line of the paragraph. For example, `text_par("Lorem ipsum dolor sit amet", N)` could represent

```
    Lorem ipsum dolor
    sit amet
```

if `N` = 0, or

```
      Lorem ipsum
    dolor sit amet
```

if `N` = 2, or

```
    Lorem ipsum dolor
      sit amet
```

if N = -2.

(The sign of the indentation is thus reversed compared to the *par/2* function, and the behaviour varies slightly depending on the sign in order to match the expected layout of a paragraph of text.)

Note that this is just a utility function, which does all the work of splitting the given string into words separated by whitespace and setting up a *par* with the proper indentation, containing a list of *text* elements.

See also: par/2, *text/1*, *text_par/1*.