STDLIB

version 1.15

Typeset in LaTeX from SGML source using the DocBuilder-0.9.8.4 Document System.

Contents

STDLIB Reference Manual

Short Summaries

- Erlang Module gen_fsm [page 198] – Generic Finite State Machine Behaviour
- Erlang Module gen_server [page 209] – Generic Server Behaviour
- Erlang Module io [page 219] – Standard IO Server Interface Functions
- Erlang Module io_lib [page 230] – IO Library Functions
- Erlang Module lib [page 234] – A number of useful library functions
- Erlang Module lists [page 236] – List Processing Functions
- Erlang Module log_mf_h [page 251] – An Event Handler which Logs Events to Disk
- Erlang Module math [page 252] – Mathematical Functions
- Erlang Module ms_transform [page 254] – Parse_transform that translates fun syntax into match specifications.
- Erlang Module orddict [page 265] – Key-Value Dictionary as Ordered List
- Erlang Module ordsets [page 270] – Functions for Manipulating Sets as Ordered Lists
- Erlang Module pg [page 273] – Distributed, Named Process Groups
- Erlang Module pool [page 275] – Load Distribution Facility
- Erlang Module proc_lib [page 277] – Functions for asynchronous and synchronous start of processes adhering to the OTP design principles.
- Erlang Module proplists [page 282] – Support functions for property lists
- Erlang Module qlc [page 287] – Query Interface to Mnesia, ETS, Dets, etc
- Erlang Module queue [page 305] – Abstract Data Type for FIFO Queues
- Erlang Module random [page 311] – Pseudo random number generation
- Erlang Module re [page 313] – Perl like regular expressions for Erlang
- Erlang Module regexp [page 349] – Regular Expression Functions for Strings
- Erlang Module sets [page 354] – Functions for Set Manipulation
- Erlang Module shell [page 357] – The Erlang Shell
- Erlang Module shell_default [page 368] – Customizing the Erlang Environment
- Erlang Module slave [page 369] – Functions to Starting and Controlling Slave Nodes
- Erlang Module sofs [page 372] – Functions for Manipulating Sets of Sets
- Erlang Module string [page 395] – String Processing Functions
- Erlang Module supervisor [page 401] – Generic Supervisor Behaviour
- Erlang Module supervisor_bridge [page 408] – Generic Supervisor Bridge Behaviour.
- Erlang Module sys [page 411] – A Functional Interface to System Messages
- Erlang Module timer [page 418] – Timer Functions
- Erlang Module win32reg [page 422] – win32reg provides access to the registry on Windows
- Erlang Module zip [page 426] – Utility for reading and creating 'zip' arhcives.

STDLIB

No functions are exported.

array

The following functions are exported:

- `default(Array::array()) -> term()`
  [page 58] Get the value used for uninitialized entries.
- `fix(Array::array()) -> array()`
  [page 58] Fix the size of the array.
- `foldl(Function, InitialAcc::term(), Array::array()) -> term()`
  [page 58] Fold the elements of the array using the given function and initial accumulator value.
- `foldr(Function, InitialAcc::term(), Array::array()) -> term()`
  [page 58] Fold the elements of the array right-to-left using the given function and initial accumulator value.
- `from_list(List::list()) -> array()`
  [page 58] Equivalent to from_list(List, undefined).
- `from_list(List::list(), Default::term()) -> array()`
  [page 58] Convert a list to an extendible array.
- `from_orddict(Orddict::list()) -> array()`
  [page 59] Equivalent to from_orddict(Orddict, undefined).
- `from_orddict(List::list(), Default::term()) -> array()`
  [page 59] Convert an ordered list of pairs {Index, Value} to a corresponding extendible array.
- `get(I::integer(), Array::array()) -> term()`
  [page 59] Get the value of entry I.
- `is_array(X::term()) -> bool()`
  [page 59] Returns true if X appears to be an array, otherwise false.
- `is_fix(Array::array()) -> bool()`
  [page 59] Check if the array has fixed size.
- `map(Function, Array::array()) -> array()`
  [page 59] Map the given function onto each element of the array.
- `new() -> array()`
  [page 59] Create a new, extendible array with initial size zero.
- `new(Options::term()) -> array()`
  [page 59] Create a new array according to the given options.
- `new(Size::integer(), Options::term()) -> array()`
  [page 59] Create a new array according to the given size and options.
- `relax(Array::array()) -> array()`
  [page 60] Make the array resizable.
- `reset(I::integer(), Array::array()) -> array()`
  [page 60] Reset entry I to the default value for the array.
- `resize(Array::array()) -> array()`
  [page 61] Change the size of the array to that reported by sparse_size/1.
- `resize(Size::integer(), Array::array()) -> array()`
  [page 61] Change the size of the array.
- `set(I::integer(), Value::term(), Array::array()) -> array()`
  [page 61] Set entry I of the array to Value.

- `size(Array::array()) -> integer()`
  [page 61] Get the number of entries in the array.
- `sparse_foldl(Function, InitialAcc::term(), Array::array()) -> term()`
  [page 61] Fold the elements of the array using the given function and initial accumulator value, skipping default-valued entries.
- `sparse_foldr(Function, InitialAcc::term(), Array::array()) -> term()`
  [page 61] Fold the elements of the array right-to-left using the given function and initial accumulator value, skipping default-valued entries.
- `sparse_map(Function, Array::array()) -> array()`
  [page 62] Map the given function onto each element of the array, skipping default-valued entries.
- `sparse_size(A::array()) -> integer()`
  [page 62] Get the number of entries in the array up until the last non-default valued entry.
- `sparse_to_list(Array::array()) -> list()`
  [page 62] Converts the array to a list, skipping default-valued entries.
- `sparse_to_orddict(Array::array()) -> [{Index::integer(), Value::term()}]`
  [page 62] Convert the array to an ordered list of pairs {Index, Value}, skipping default-valued entries.
- `to_list(Array::array()) -> list()`
  [page 62] Converts the array to a list.
- `to_orddict(Array::array()) -> [{Index::integer(), Value::term()}]`
  [page 62] Convert the array to an ordered list of pairs {Index, Value}.

base64

The following functions are exported:

- `encode(Data) -> Base64`
  [page 63] Encodes data into base64.
- `encode_to_string(Data) -> Base64String`
  [page 63] Encodes data into base64.
- `decode(Base64) -> Data`
  [page 63] Decodes a base64 encoded string to data.
- `decode_to_string(Base64) -> DataString`
  [page 63] Decodes a base64 encoded string to data.
- `mime_decode(Base64) -> Data`
  [page 63] Decodes a base64 encoded string to data.
- `mime_decode_to_string(Base64) -> DataString`
  [page 63] Decodes a base64 encoded string to data.

beam_lib

The following functions are exported:

- `chunks(Beam, [ChunkRef]) -> {ok, {Module, [ChunkData]}} | {error, beam_lib, Reason}`
  [page 67] Read selected chunks from a BEAM file or binary

- `chunks(Beam, [ChunkRef], [Option]) -> {ok, {Module, [ChunkResult]}} | {error, beam_lib, Reason}`
  [page 67] Read selected chunks from a BEAM file or binary
- `version(Beam) -> {ok, {Module, [Version]}} | {error, beam_lib, Reason}`
  [page 67] Read the BEAM file's module version
- `md5(Beam) -> {ok, {Module, MD5}} | {error, beam_lib, Reason}`
  [page 68] Read the BEAM file's module version
- `info(Beam) -> [{Item, Info}] | {error, beam_lib, Reason1}`
  [page 68] Information about a BEAM file
- `cmp(Beam1, Beam2) -> ok | {error, beam_lib, Reason}`
  [page 69] Compare two BEAM files
- `cmp_dirs(Dir1, Dir2) -> {Only1, Only2, Different} | {error, beam_lib, Reason1}`
  [page 69] Compare the BEAM files in two directories
- `diff_dirs(Dir1, Dir2) -> ok | {error, beam_lib, Reason1}`
  [page 69] Compare the BEAM files in two directories
- `strip(Beam1) -> {ok, {Module, Beam2}} | {error, beam_lib, Reason1}`
  [page 69] Removes chunks not needed by the loader from a BEAM file
- `strip_files(Files) -> {ok, [{Module, Beam2}]} | {error, beam_lib, Reason1}`
  [page 69] Removes chunks not needed by the loader from BEAM files
- `strip_release(Dir) -> {ok, [{Module, Filename]}} | {error, beam_lib, Reason1}`
  [page 70] Removes chunks not needed by the loader from all BEAM files of a release
- `format_error(Reason) -> Chars`
  [page 70] Return an English description of a BEAM read error reply
- `crypto_key_fun(CryptoKeyFun) -> ok | {error, Reason}`
  [page 70] Register a fun that provides a crypto key
- `clear_crypto_key_fun() -> {ok, Result}`
  [page 71] Unregister the current crypto key fun

c

The following functions are exported:

- `bt(Pid) -> void()`
  [page 72] Stack backtrace for a process
- `c(File) -> {ok, Module} | error`
  [page 72] Compile and load code in a file
- `c(File, Options) -> {ok, Module} | error`
  [page 72] Compile and load code in a file
- `cd(Dir) -> void()`
  [page 72] Change working directory
- `flush() -> void()`
  [page 73] Flush any messages sent to the shell

calendar

The following functions are exported:

- `date_to_gregorian_days(Date) -> Days`
  [page 77] Compute the number of days from year 0 up to the given date
- `date_to_gregorian_days(Year, Month, Day) -> Days`
  [page 77] Compute the number of days from year 0 up to the given date
- `datetime_to_gregorian_seconds({Date, Time}) -> Seconds`
  [page 77] Compute the number of seconds from year 0 up to the given date and time
- `day_of_the_week(Date) -> DayNumber`
  [page 77] Compute the day of the week
- `day_of_the_week(Year, Month, Day) -> DayNumber`
  [page 77] Compute the day of the week
- `gregorian_days_to_date(Days) -> Date`
  [page 77] Compute the date given the number of gregorian days
- `gregorian_seconds_to_datetime(Seconds) -> {Date, Time}`
  [page 77] Compute the date given the number of gregorian days
- `is_leap_year(Year) -> bool()`
  [page 77] Check if a year is a leap year
- `last_day_of_the_month(Year, Month) -> int()`
  [page 77] Compute the number of days in a month
- `local_time() -> {Date, Time}`
  [page 78] Compute local time
- `local_time_to_universal_time({Date1, Time1}) -> {Date2, Time2}`
  [page 78] Convert from local time to universal time (deprecated)
- `local_time_to_universal_time_dst({Date1, Time1}) -> [{Date, Time}]`
  [page 78] Convert from local time to universal time(s)
- `now_to_local_time(Now) -> {Date, Time}`
  [page 78] Convert now to local date and time
- `now_to_universal_time(Now) -> {Date, Time}`
  [page 79] Convert now to date and time
- `now_to_datetime(Now) -> {Date, Time}`
  [page 79] Convert now to date and time
- `seconds_to_daystime(Seconds) -> {Days, Time}`
  [page 79] Compute days and time from seconds
- `seconds_to_time(Seconds) -> Time`
  [page 79] Compute time from seconds
- `time_difference(T1, T2) -> {Days, Time}`
  [page 79] Compute the difference between two times (deprecated)
- `time_to_seconds(Time) -> Seconds`
  [page 79] Compute the number of seconds since midnight up to the given time
- `universal_time() -> {Date, Time}`
  [page 79] Compute universal time
- `universal_time_to_local_time({Date1, Time1}) -> {Date2, Time2}`
  [page 80] Convert from universal time to local time

- `valid_date(Date) -> bool()`
  [page 80] Check if a date is valid
- `valid_date(Year, Month, Day) -> bool()`
  [page 80] Check if a date is valid

dets

The following functions are exported:

- `all() -> [Name]`
  [page 82] Return a list of the names of all open Dets tables on this node.
- `bchunk(Name, Continuation) -> {Continuation2, Data} | '$end_of_table' | {error, Reason}`
  [page 82] Return a chunk of objects stored in a Dets table.
- `close(Name) -> ok | {error, Reason}`
  [page 83] Close a Dets table.
- `delete(Name, Key) -> ok | {error, Reason}`
  [page 83] Delete all objects with a given key from a Dets table.
- `delete_all_objects(Name) -> ok | {error, Reason}`
  [page 83] Delete all objects from a Dets table.
- `delete_object(Name, Object) -> ok | {error, Reason}`
  [page 83] Delete a given object from a Dets table.
- `first(Name) -> Key | '$end_of_table'`
  [page 83] Return the first key stored in a Dets table.
- `foldl(Function, Acc0, Name) -> Acc1 | {error, Reason}`
  [page 84] Fold a function over a Dets table.
- `foldr(Function, Acc0, Name) -> Acc1 | {error, Reason}`
  [page 84] Fold a function over a Dets table.
- `from_ets(Name, EtsTab) -> ok | {error, Reason}`
  [page 84] Replace the objects of a Dets table with the objects of an Ets table.
- `info(Name) -> InfoList | undefined`
  [page 84] Return information about a Dets table.
- `info(Name, Item) -> Value | undefined`
  [page 85] Return the information associated with a given item for a Dets table.
- `init_table(Name, InitFun [, Options]) -> ok | {error, Reason}`
  [page 85] Replace all objects of a Dets table.
- `insert(Name, Objects) -> ok | {error, Reason}`
  [page 86] Insert one or more objects into a Dets table.
- `insert_new(Name, Objects) -> Bool`
  [page 86] Insert one or more objects into a Dets table.
- `is_compatible_bchunk_format(Name, BchunkFormat) -> Bool`
  [page 87] Test compatibility of a table's chunk data.
- `is_dets_file(FileName) -> Bool | {error, Reason}`
  [page 87] Test for a Dets table.
- `lookup(Name, Key) -> [Object] | {error, Reason}`
  [page 87] Return all objects with a given key stored in a Dets table.

- `match(Continuation) -> {[Match], Continuation2} | '$end_of_table' | {error, Reason}`
  [page 87] Match a chunk of objects stored in a Dets table and return a list of variable bindings.
- `match(Name, Pattern) -> [Match] | {error, Reason}`
  [page 88] Match the objects stored in a Dets table and return a list of variable bindings.
- `match(Name, Pattern, N) -> {[Match], Continuation} | '$end_of_table' | {error, Reason}`
  [page 88] Match the first chunk of objects stored in a Dets table and return a list of variable bindings.
- `match_delete(Name, Pattern) -> N | {error, Reason}`
  [page 88] Delete all objects that match a given pattern from a Dets table.
- `match_object(Continuation) -> {[Object], Continuation2} | '$end_of_table' | {error, Reason}`
  [page 89] Match a chunk of objects stored in a Dets table and return a list of objects.
- `match_object(Name, Pattern) -> [Object] | {error, Reason}`
  [page 89] Match the objects stored in a Dets table and return a list of objects.
- `match_object(Name, Pattern, N) -> {[Object], Continuation} | '$end_of_table' | {error, Reason}`
  [page 89] Match the first chunk of objects stored in a Dets table and return a list of objects.
- `member(Name, Key) -> Bool | {error, Reason}`
  [page 90] Test for occurrence of a key in a Dets table.
- `next(Name, Key1) -> Key2 | '$end_of_table'`
  [page 90] Return the next key in a Dets table.
- `open_file(Filename) -> {ok, Reference} | {error, Reason}`
  [page 90] Open an existing Dets table.
- `open_file(Name, Args) -> {ok, Name} | {error, Reason}`
  [page 90] Open a Dets table.
- `pid2name(Pid) -> {ok, Name} | undefined`
  [page 92] Return the name of the Dets table handled by a pid.
- `repair_continuation(Continuation, MatchSpec) -> Continuation2`
  [page 92] Repair a continuation from select/1 or select/3.
- `safe_fixtable(Name, Fix)`
  [page 92] Fix a Dets table for safe traversal.
- `select(Continuation) -> {Selection, Continuation2} | '$end_of_table' | {error, Reason}`
  [page 93] Apply a match specification to some objects stored in a Dets table.
- `select(Name, MatchSpec) -> Selection | {error, Reason}`
  [page 93] Apply a match specification to all objects stored in a Dets table.
- `select(Name, MatchSpec, N) -> {Selection, Continuation} | '$end_of_table' | {error, Reason}`
  [page 93] Apply a match specification to the first chunk of objects stored in a Dets table.
- `select_delete(Name, MatchSpec) -> N | {error, Reason}`
  [page 94] Delete all objects that match a given pattern from a Dets table.

- `slot(Name, I) -> '$end_of_table' | [Object] | {error, Reason}`
  [page 94] Return the list of objects associated with a slot of a Dets table.
- `sync(Name) -> ok | {error, Reason}`
  [page 94] Ensure that all updates made to a Dets table are written to disk.
- `table(Name [, Options]) -> QueryHandle`
  [page 95] Return a QLC query handle.
- `to_ets(Name, EtsTab) -> EtsTab | {error, Reason}`
  [page 95] Insert all objects of a Dets table into an Ets table.
- `traverse(Name, Fun) -> Return | {error, Reason}`
  [page 96] Apply a function to all or some objects stored in a Dets table.
- `update_counter(Name, Key, Increment) -> Result`
  [page 96] Update a counter object stored in a Dets table.

dict

The following functions are exported:

- `append(Key, Value, Dict1) -> Dict2`
  [page 97] Append a value to keys in a dictionary
- `append_list(Key, ValList, Dict1) -> Dict2`
  [page 97] Append new values to keys in a dictionary
- `erase(Key, Dict1) -> Dict2`
  [page 97] Erase a key from a dictionary
- `fetch(Key, Dict) -> Value`
  [page 98] Look-up values in a dictionary
- `fetch_keys(Dict) -> Keys`
  [page 98] Return all keys in a dictionary
- `filter(Pred, Dict1) -> Dict2`
  [page 98] Choose elements which satisfy a predicate
- `find(Key, Dict) -> {ok, Value} | error`
  [page 98] Search for a key in a dictionary
- `fold(Fun, Acc0, Dict) -> Acc1`
  [page 98] Fold a function over a dictionary
- `from_list(List) -> Dict`
  [page 98] Convert a list of pairs to a dictionary
- `is_key(Key, Dict) -> bool()`
  [page 99] Test if a key is in a dictionary
- `map(Fun, Dict1) -> Dict2`
  [page 99] Map a function over a dictionary
- `merge(Fun, Dict1, Dict2) -> Dict3`
  [page 99] Merge two dictionaries
- `new() -> dictionary()`
  [page 99] Create a dictionary
- `size(Dict) -> int()`
  [page 99] Return the number of elements in a dictionary
- `store(Key, Value, Dict1) -> Dict2`
  [page 99] Store a value in a dictionary

- `to_list(Dict) -> List`
  [page 100] Convert a dictionary to a list of pairs
- `update(Key, Fun, Dict1) -> Dict2`
  [page 100] Update a value in a dictionary
- `update(Key, Fun, Initial, Dict1) -> Dict2`
  [page 100] Update a value in a dictionary
- `update_counter(Key, Increment, Dict1) -> Dict2`
  [page 100] Increment a value in a dictionary

digraph

The following functions are exported:

- `add_edge(G, E, V1, V2, Label) -> edge() | {error, Reason}`
  [page 102] Add an edge to a digraph.
- `add_edge(G, V1, V2, Label) -> edge() | {error, Reason}`
  [page 102] Add an edge to a digraph.
- `add_edge(G, V1, V2) -> edge() | {error, Reason}`
  [page 102] Add an edge to a digraph.
- `add_vertex(G, V, Label) -> vertex()`
  [page 103] Add or modify a vertex of a digraph.
- `add_vertex(G, V) -> vertex()`
  [page 103] Add or modify a vertex of a digraph.
- `add_vertex(G) -> vertex()`
  [page 103] Add or modify a vertex of a digraph.
- `del_edge(G, E) -> true`
  [page 103] Delete an edge from a digraph.
- `del_edges(G, Edges) -> true`
  [page 103] Delete edges from a digraph.
- `del_path(G, V1, V2) -> true`
  [page 103] Delete paths from a digraph.
- `del_vertex(G, V) -> true`
  [page 104] Delete a vertex from a digraph.
- `del_vertices(G, Vertices) -> true`
  [page 104] Delete vertices from a digraph.
- `delete(G) -> true`
  [page 104] Delete a digraph.
- `edge(G, E) -> {E, V1, V2, Label} | false`
  [page 104] Return the vertices and the label of an edge of a digraph.
- `edges(G) -> Edges`
  [page 104] Return all edges of a digraph.
- `edges(G, V) -> Edges`
  [page 104] Return the edges emanating from or incident on a vertex of a digraph.
- `get_cycle(G, V) -> Vertices | false`
  [page 105] Find one cycle in a digraph.
- `get_path(G, V1, V2) -> Vertices | false`
  [page 105] Find one path in a digraph.

- `get_short_cycle(G, V) -> Vertices | false`
  [page 105] Find one short cycle in a digraph.
- `get_short_path(G, V1, V2) -> Vertices | false`
  [page 105] Find one short path in a digraph.
- `in_degree(G, V) -> integer()`
  [page 106] Return the in-degree of a vertex of a digraph.
- `in_edges(G, V) -> Edges`
  [page 106] Return all edges incident on a vertex of a digraph.
- `in_neighbours(G, V) -> Vertices`
  [page 106] Return all in-neighbours of a vertex of a digraph.
- `info(G) -> InfoList`
  [page 106] Return information about a digraph.
- `new() -> digraph()`
  [page 107] Return a protected empty digraph, where cycles are allowed.
- `new(Type) -> digraph() | {error, Reason}`
  [page 107] Create a new empty digraph.
- `no_edges(G) -> integer() >= 0`
  [page 107] Return the number of edges of the a digraph.
- `no_vertices(G) -> integer() >= 0`
  [page 107] Return the number of vertices of a digraph.
- `out_degree(G, V) -> integer()`
  [page 107] Return the out-degree of a vertex of a digraph.
- `out_edges(G, V) -> Edges`
  [page 107] Return all edges emanating from a vertex of a digraph.
- `out_neighbours(G, V) -> Vertices`
  [page 107] Return all out-neighbours of a vertex of a digraph.
- `vertex(G, V) -> {V, Label} | false`
  [page 108] Return the label of a vertex of a digraph.
- `vertices(G) -> Vertices`
  [page 108] Return all vertices of a digraph.

digraph_utils

The following functions are exported:

- `arborescence_root(Digraph) -> no | {yes, Root}`
  [page 110] Check if a digraph is an arborescence.
- `components(Digraph) -> [Component]`
  [page 110] Return the components of a digraph.
- `condensation(Digraph) -> CondensedDigraph`
  [page 110] Return a condensed graph of a digraph.
- `cyclic_strong_components(Digraph) -> [StrongComponent]`
  [page 110] Return the cyclic strong components of a digraph.
- `is_acyclic(Digraph) -> bool()`
  [page 110] Check if a digraph is acyclic.
- `is_arborescence(Digraph) -> bool()`
  [page 111] Check if a digraph is an arborescence.

- `is_tree(Digraph) -> bool()`
  [page 111] Check if a digraph is a tree.
- `loop_vertices(Digraph) -> Vertices`
  [page 111] Return the vertices of a digraph included in some loop.
- `postorder(Digraph) -> Vertices`
  [page 111] Return the vertices of a digraph in post-order.
- `preorder(Digraph) -> Vertices`
  [page 111] Return the vertices of a digraph in pre-order.
- `reachable(Vertices, Digraph) -> Vertices`
  [page 111] Return the vertices reachable from some vertices of a digraph.
- `reachable_neighbours(Vertices, Digraph) -> Vertices`
  [page 112] Return the neighbours reachable from some vertices of a digraph.
- `reaching(Vertices, Digraph) -> Vertices`
  [page 112] Return the vertices that reach some vertices of a digraph.
- `reaching_neighbours(Vertices, Digraph) -> Vertices`
  [page 112] Return the neighbours that reach some vertices of a digraph.
- `strong_components(Digraph) -> [StrongComponent]`
  [page 112] Return the strong components of a digraph.
- `subgraph(Digraph, Vertices [, Options]) -> Subgraph | {error, Reason}`
  [page 112] Return a subgraph of a digraph.
- `topsort(Digraph) -> Vertices | false`
  [page 113] Return a topological sorting of the vertices of a digraph.

epp

The following functions are exported:

- `open(FileName, IncludePath) -> {ok,Epp} | {error, ErrorDescriptor}`
  [page 114] Open a file for preprocessing
- `open(FileName, IncludePath, PredefMacros) -> {ok,Epp} | {error, ErrorDescriptor}`
  [page 114] Open a file for preprocessing
- `close(Epp) -> ok`
  [page 114] Close the preprocessing of the file associated with `Epp`
- `parse_erl_form(Epp) -> {ok, AbsForm} | {eof, Line} | {error, ErrorInfo}`
  [page 114] Return the next Erlang form from the opened Erlang source file
- `parse_file(FileName,IncludePath,PredefMacro) -> {ok,[Form]} | {error,OpenError}`
  [page 114] Preprocesse and parse an Erlang source file

erl_eval

The following functions are exported:

- `exprs(Expressions, Bindings) -> {value, Value, NewBindings}`
  [page 116] Evaluate expressions
- `exprs(Expressions, Bindings, LocalFunctionHandler) -> {value, Value, NewBindings}`
  [page 116] Evaluate expressions
- `exprs(Expressions, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {value, Value, NewBindings}`
  [page 116] Evaluate expressions
- `expr(Expression, Bindings) -> { value, Value, NewBindings }`
  [page 116] Evaluate expression
- `expr(Expression, Bindings, LocalFunctionHandler) -> { value, Value, NewBindings }`
  [page 116] Evaluate expression
- `expr(Expression, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> { value, Value, NewBindings }`
  [page 116] Evaluate expression
- `expr_list(ExpressionList, Bindings) -> {ValueList, NewBindings}`
  [page 117] Evaluate a list of expressions
- `expr_list(ExpressionList, Bindings, LocalFunctionHandler) -> {ValueList, NewBindings}`
  [page 117] Evaluate a list of expressions
- `expr_list(ExpressionList, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {ValueList, NewBindings}`
  [page 117] Evaluate a list of expressions
- `new_bindings() -> BindingStruct`
  [page 117] Return a bindings structure
- `bindings(BindingStruct) -> Bindings`
  [page 117] Return bindings
- `binding(Name, BindingStruct) -> Binding`
  [page 117] Return bindings
- `add_binding(Name, Value, Bindings) -> BindingStruct`
  [page 117] Add a binding
- `del_binding(Name, Bindings) -> BindingStruct`
  [page 117] Delete a binding

erl_expand_records

The following functions are exported:

- `module(AbsForms, CompileOptions) -> AbsForms`
  [page 119] Expand all records in a module

erl_id_trans

The following functions are exported:

- `parse_transform(Forms, Options) -> Forms`
  [page 120] Transform Erlang forms

erl_internal

The following functions are exported:

- `bif(Name, Arity) -> bool()`
  [page 121] Test for an Erlang BIF
- `guard_bif(Name, Arity) -> bool()`
  [page 121] Test for an Erlang BIF allowed in guards
- `type_test(Name, Arity) -> bool()`
  [page 121] Test for a valid type test
- `arith_op(OpName, Arity) -> bool()`
  [page 121] Test for an arithmetic operator
- `bool_op(OpName, Arity) -> bool()`
  [page 121] Test for a Boolean operator
- `comp_op(OpName, Arity) -> bool()`
  [page 122] Test for a comparison operator
- `list_op(OpName, Arity) -> bool()`
  [page 122] Test for a list operator
- `send_op(OpName, Arity) -> bool()`
  [page 122] Test for a send operator
- `op_type(OpName, Arity) -> Type`
  [page 122] Return operator type

erl_lint

The following functions are exported:

- `module(AbsForms) -> {ok,Warnings} | {error,Errors,Warnings}`
  [page 123] Check a module for errors
- `module(AbsForms, FileName) -> {ok,Warnings} | {error,Errors,Warnings}`
  [page 123] Check a module for errors
- `module(AbsForms, FileName, CompileOptions) -> {ok,Warnings} | {error,Errors,Warnings}`
  [page 123] Check a module for errors
- `is_guard_test(Expr) -> bool()`
  [page 124] Test for a guard test
- `format_error(ErrorDescriptor) -> Chars`
  [page 124] Format an error descriptor

erl_parse

The following functions are exported:

- `parse_form(Tokens) -> {ok, AbsForm} | {error, ErrorInfo}`
  [page 125] Parse an Erlang form
- `parse_exprs(Tokens) -> {ok, Expr_list} | {error, ErrorInfo}`
  [page 125] Parse Erlang expressions
- `parse_term(Tokens) -> {ok, Term} | {error, ErrorInfo}`
  [page 125] Parse an Erlang term
- `format_error(ErrorDescriptor) -> Chars`
  [page 126] Format an error descriptor
- `tokens(AbsTerm) -> Tokens`
  [page 126] Generate a list of tokens for an expression
- `tokens(AbsTerm, MoreTokens) -> Tokens`
  [page 126] Generate a list of tokens for an expression
- `normalise(AbsTerm) -> Data`
  [page 126] Convert abstract form to an Erlang term
- `abstract(Data) -> AbsTerm`
  [page 126] Convert an Erlang term into an abstract form

erl_pp

The following functions are exported:

- `form(Form) -> DeepCharList`
  [page 128] Pretty print a form
- `form(Form, HookFunction) -> DeepCharList`
  [page 128] Pretty print a form
- `attribute(Attribute) -> DeepCharList`
  [page 128] Pretty print an attribute
- `attribute(Attribute, HookFunction) -> DeepCharList`
  [page 128] Pretty print an attribute
- `function(Function) -> DeepCharList`
  [page 128] Pretty print a function
- `function(Function, HookFunction) -> DeepCharList`
  [page 128] Pretty print a function
- `guard(Guard) -> DeepCharList`
  [page 128] Pretty print a guard
- `guard(Guard, HookFunction) -> DeepCharList`
  [page 128] Pretty print a guard
- `exprs(Expressions) -> DeepCharList`
  [page 129] Pretty print `Expressions`
- `exprs(Expressions, HookFunction) -> DeepCharList`
  [page 129] Pretty print `Expressions`
- `exprs(Expressions, Indent, HookFunction) -> DeepCharList`
  [page 129] Pretty print `Expressions`

- `expr(Expression) -> DeepCharList`
  [page 129] Pretty print one `Expression`
- `expr(Expression, HookFunction) -> DeepCharList`
  [page 129] Pretty print one `Expression`
- `expr(Expression, Indent, HookFunction) -> DeepCharList`
  [page 129] Pretty print one `Expression`
- `expr(Expression, Indent, Precedence, HookFunction) ->-> DeepCharList`
  [page 129] Pretty print one `Expression`

erl_scan

The following functions are exported:

- `string(CharList,StartLine]) -> {ok, Tokens, EndLine} | Error`
  [page 131] Scan a string and returns the Erlang tokens
- `string(CharList) -> {ok, Tokens, EndLine} | Error`
  [page 131] Scan a string and returns the Erlang tokens
- `tokens(Continuation, CharList, StartLine) ->Return`
  [page 131] Re-entrant scanner
- `reserved_word(Atom) -> bool()`
  [page 132] Test for a reserved word
- `format_error(ErrorDescriptor) -> string()`
  [page 132] Format an error descriptor

erl_tar

The following functions are exported:

- `add(TarDescriptor, Filename, Options) -> RetValue`
  [page 134] Add a file to an open tar file
- `add(TarDescriptor, FilenameOrBin, NameInArchive, Options) -> RetValue`
  [page 134] Add a file to an open tar file
- `close(TarDescriptor)`
  [page 134] Close an open tar file
- `create(Name, FileList) ->RetValue`
  [page 134] Create a tar archive
- `create(Name, FileList, OptionList)`
  [page 135] Create a tar archive with options
- `extract(Name) -> RetValue`
  [page 135] Extract all files from a tar file
- `extract(Name, OptionList)`
  [page 135] Extract files from a tar file
- `format_error(Reason) -> string()`
  [page 136] Convert error term to a readable string
- `open(Name, OpenModeList) -> RetValue`
  [page 136] Open a tar file.

- `table(Name) -> RetValue`
  [page 137] Retrieve the name of all files in a tar file
- `table(Name, Options)`
  [page 137] Retrieve name and information of all files in a tar file
- `t(Name)`
  [page 137] Print the name of each file in a tar file
- `tt(Name)`
  [page 138] Print name and information for each file in a tar file

ets

The following functions are exported:

- `all() -> [Tab]`
  [page 140] Return a list of all ETS tables.
- `delete(Tab) -> true`
  [page 140] Delete an entire ETS table.
- `delete(Tab, Key) -> true`
  [page 140] Delete all objects with a given key from an ETS table.
- `delete_all_objects(Tab) -> true`
  [page 140] Delete all objects in an ETS table.
- `delete_object(Tab,Object) -> true`
  [page 140] Deletes a specific from an ETS table.
- `file2tab(Filename) -> {ok,Tab} | {error,Reason}`
  [page 141] Read an ETS table from a file.
- `file2tab(Filename,Options) -> {ok,Tab} | {error,Reason}`
  [page 141] Read an ETS table from a file.
- `first(Tab) -> Key | '$end_of_table'`
  [page 141] Return the first key in an ETS table.
- `fixtable(Tab, true|false) -> true | false`
  [page 142] Fix an ETS table for safe traversal (obsolete).
- `foldl(Function, Acc0, Tab) -> Acc1`
  [page 142] Fold a function over an ETS table
- `foldr(Function, Acc0, Tab) -> Acc1`
  [page 142] Fold a function over an ETS table
- `from_dets(Tab, DetsTab) -> true`
  [page 142] Fill an ETS table with objects from a Dets table.
- `fun2ms(LiteralFun) -> MatchSpec`
  [page 143] Pseudo function that transforms fun syntax to a match_spec.
- `i() -> void()`
  [page 144] Display information about all ETS tables on tty.
- `i(Tab) -> void()`
  [page 144] Browse an ETS table on tty.
- `info(Tab) -> [{Item, Value}] | undefined`
  [page 144] Return information about an ETS table.
- `info(Tab, Item) -> Value | undefined`
  [page 145] Return the information associated with given item for an ETS table.

- `init_table(Name, InitFun) -> true`
  [page 145] Replace all objects of an ETS table.
- `insert(Tab, ObjectOrObjects) -> true`
  [page 146] Insert an object into an ETS table.
- `insert_new(Tab, ObjectOrObjects) -> bool()`
  [page 146] Insert an object into an ETS table if the key is not already present.
- `is_compiled_ms(Term) -> bool()`
  [page 146] Checks if an Erlang term is the result of ets:match_spec_compile
- `last(Tab) -> Key | '$end_of_table'`
  [page 147] Return the last key in an ETS table of type`ordered_set`.
- `lookup(Tab, Key) -> [Object]`
  [page 147] Return all objects with a given key in an ETS table.
- `lookup_element(Tab, Key, Pos) -> Elem`
  [page 147] Return the `Pos`:th element of all objects with a given key in an ETS table.
- `match(Tab, Pattern) -> [Match]`
  [page 148] Match the objects in an ETS table against a pattern.
- `match(Tab, Pattern, Limit) -> {[Match],Continuation} | '$end_of_table'`
  [page 148] Match the objects in an ETS table against a pattern and returns part of the answers.
- `match(Continuation) -> {[Match],Continuation} | '$end_of_table'`
  [page 149] Continues matching objects in an ETS table.
- `match_delete(Tab, Pattern) -> true`
  [page 149] Delete all objects which match a given pattern from an ETS table.
- `match_object(Tab, Pattern) -> [Object]`
  [page 149] Match the objects in an ETS table against a pattern.
- `match_object(Tab, Pattern, Limit) -> {[Match],Continuation} | '$end_of_table'`
  [page 149] Match the objects in an ETS table against a pattern and returns part of the answers.
- `match_object(Continuation) -> {[Match],Continuation} | '$end_of_table'`
  [page 150] Continues matching objects in an ETS table.
- `match_spec_compile(MatchSpec) -> CompiledMatchSpec`
  [page 150] Compiles a match specification into its internal representation
- `match_spec_run(List,CompiledMatchSpec) -> list()`
  [page 150] Performs matching, using a compiled match_spec, on a list of tuples
- `member(Tab, Key) -> true | false`
  [page 151] Tests for occurrence of a key in an ETS table
- `new(Name, Options) -> tid()`
  [page 151] Create a new ETS table.
- `next(Tab, Key1) -> Key2 | '$end_of_table'`
  [page 152] Return the next key in an ETS table.
- `prev(Tab, Key1) -> Key2 | '$end_of_table'`
  [page 152] Return the previous key in an ETS table of type`ordered_set`.

- `rename(Tab, Name) -> Name`
  [page 153] Rename a named ETS table.
- `repair_continuation(Continuation, MatchSpec) -> Continuation`
  [page 153] Repair a continuation from ets:select/1 or ets:select/3 that has passed through external representation
- `safe_fixtable(Tab, true|false) -> true`
  [page 154] Fix an ETS table for safe traversal.
- `select(Tab, MatchSpec) -> [Match]`
  [page 155] Match the objects in an ETS table against a match_spec.
- `select(Tab, MatchSpec, Limit) -> {[Match],Continuation} | '$end_of_table'`
  [page 156] Match the objects in an ETS table against a match_spec and returns part of the answers.
- `select(Continuation) -> {[Match],Continuation} | '$end_of_table'`
  [page 156] Continue matching objects in an ETS table.
- `select_delete(Tab, MatchSpec) -> NumDeleted`
  [page 157] Match the objects in an ETS table against a match_spec and deletes objects where the match_spec returns 'true'
- `select_count(Tab, MatchSpec) -> NumMatched`
  [page 157] Match the objects in an ETS table against a match_spec and returns the number of objects for which the match_spec returned 'true'
- `slot(Tab, I) -> [Object] | '$end_of_table'`
  [page 157] Return all objects in a given slot of an ETS table.
- `tab2file(Tab, Filename) -> ok | {error,Reason}`
  [page 158] Dump an ETS table to a file.
- `tab2file(Tab, Filename, Options) -> ok | {error,Reason}`
  [page 158] Dump an ETS table to a file.
- `tab2list(Tab) -> [Object]`
  [page 159] Return a list of all objects in an ETS table.
- `tabfile_info(Filename) -> {ok, TableInfo} | {error, Reason}`
  [page 159] Return a list of all objects in an ETS table.
- `table(Tab [, Options]) -> QueryHandle`
  [page 160] Return a QLC query handle.
- `test_ms(Tuple, MatchSpec) -> {ok, Result} | {error, Errors}`
  [page 161] Test a match_spec for use in ets:select/2.
- `to_dets(Tab, DetsTab) -> Tab`
  [page 161] Fill a Dets table with objects from an ETS table.
- `update_counter(Tab, Key, UpdateOp) -> Result`
  [page 161] Update a counter object in an ETS table.
- `update_counter(Tab, Key, [UpdateOp]) -> [Result]`
  [page 161] Update a counter object in an ETS table.
- `update_counter(Tab, Key, Incr) -> Result`
  [page 161] Update a counter object in an ETS table.
- `update_element(Tab, Key, {Pos,Value}) -> true | false`
  [page 162] Updates the Pos:th element of the object with a given key in an ETS table.
- `update_element(Tab, Key, [{Pos,Value}]) -> true | false`
  [page 162] Updates the Pos:th element of the object with a given key in an ETS table.

file_sorter

The following functions are exported:

- `sort(FileName) -> Reply`
  [page 166] Sort terms on files.
- `sort(Input, Output) -> Reply`
  [page 166] Sort terms on files.
- `sort(Input, Output, Options) -> Reply`
  [page 166] Sort terms on files.
- `keysort(KeyPos, FileName) -> Reply`
  [page 166] Sort terms on files by key.
- `keysort(KeyPos, Input, Output) -> Reply`
  [page 166] Sort terms on files by key.
- `keysort(KeyPos, Input, Output, Options) -> Reply`
  [page 166] Sort terms on files by key.
- `merge(FileNames, Output) -> Reply`
  [page 166] Merge terms on files.
- `merge(FileNames, Output, Options) -> Reply`
  [page 166] Merge terms on files.
- `keymerge(KeyPos, FileNames, Output) -> Reply`
  [page 167] Merge terms on files by key.
- `keymerge(KeyPos, FileNames, Output, Options) -> Reply`
  [page 167] Merge terms on files by key.
- `check(FileName) -> Reply`
  [page 167] Check whether terms on files are sorted.
- `check(FileNames, Options) -> Reply`
  [page 167] Check whether terms on files are sorted.
- `keycheck(KeyPos, FileName) -> CheckReply`
  [page 167] Check whether terms on files are sorted by key.
- `keycheck(KeyPos, FileNames, Options) -> Reply`
  [page 167] Check whether terms on files are sorted by key.

filelib

The following functions are exported:

- `ensure_dir(Name) -> ok | {error, Reason}`
  [page 168] Ensure that all parent directories for a file or directory exist.
- `file_size(Filename) -> integer()`
  [page 168] Return the size in bytes of the file.
- `fold_files(Dir, RegExp, Recursive, Fun, AccIn) -> AccOut`
  [page 168] Fold over all files matching a regular expression.
- `is_dir(Name) -> true | false`
  [page 169] Test whether Name refer to a directory or not
- `is_file(Name) -> true | false`
  [page 169] Test whether Name refer to a file or directory.

- `is_regular(Name) -> true | false`
  [page 169] Test whether Name refer to a (regular) file.
- `last_modified(Name) -> {{Year,Month,Day},{Hour,Min,Sec}}`
  [page 169] Return the local date and time when a file was last modified.
- `wildcard(Wildcard) -> list()`
  [page 169] Match filenames using Unix-style wildcards.
- `wildcard(Wildcard, Cwd) -> list()`
  [page 170] Match filenames using Unix-style wildcards starting at a specified directory.

filename

The following functions are exported:

- `absname(Filename) -> string()`
  [page 171] Convert a filename to an absolute name, relative the working directory
- `absname(Filename, Dir) -> string()`
  [page 172] Convert a filename to an absolute name, relative a specified directory
- `absname_join(Dir, Filename) -> string()`
  [page 172] Join an absolute directory with a relative filename
- `basename(Filename) -> string()`
  [page 172] Return the last component of a filename
- `basename(Filename, Ext) -> string()`
  [page 172] Return the last component of a filename, stripped of the specified extension
- `dirname(Filename) -> string()`
  [page 173] Return the directory part of a path name
- `extension(Filename) -> string()`
  [page 173] Return the file extension
- `flatten(Filename) -> string()`
  [page 173] Convert a filename to a flat string
- `join(Components) -> string()`
  [page 173] Join a list of filename components with directory separators
- `join(Name1, Name2) -> string()`
  [page 174] Join two filename components with directory separators
- `nativename(Path) -> string()`
  [page 174] Return the native form of a file path
- `pathtype(Path) -> absolute | relative | volumerelative`
  [page 174] Return the type of a path
- `rootname(Filename) -> string()`
  [page 174] Remove a filename extension
- `rootname(Filename, Ext) -> string()`
  [page 174] Remove a filename extension
- `split(Filename) -> Components`
  [page 175] Split a filename into its path components
- `find_src(Beam) -> {SourceFile, Options} | {error,{ErrorReason,Module}}`
  [page 175] Find the filename and compiler options for a module

- `find_src(Beam, Rules) -> {SourceFile, Options} | {error,{ErrorReason,Module}}`
  [page 175] Find the filename and compiler options for a module

gb_sets

The following functions are exported:

- `add(Element, Set1) -> Set2`
  [page 178] Add a (possibly existing) element to a gb_set
- `add_element(Element, Set1) -> Set2`
  [page 178] Add a (possibly existing) element to a gb_set
- `balance(Set1) -> Set2`
  [page 178] Rebalance tree representation of a gb_set
- `delete(Element, Set1) -> Set2`
  [page 178] Remove an element from a gb_set
- `delete_any(Element, Set1) -> Set2`
  [page 178] Remove a (possibly non-existing) element from a gb_set
- `del_element(Element, Set1) -> Set2`
  [page 178] Remove a (possibly non-existing) element from a gb_set
- `difference(Set1, Set2) -> Set3`
  [page 178] Return the difference of two gb_sets
- `subtract(Set1, Set2) -> Set3`
  [page 179] Return the difference of two gb_sets
- `empty() -> Set`
  [page 179] Return an empty gb_set
- `new() -> Set`
  [page 179] Return an empty gb_set
- `filter(Pred, Set1) -> Set2`
  [page 179] Filter gb_set elements
- `fold(Function, Acc0, Set) -> Acc1`
  [page 179] Fold over gb_set elements
- `from_list(List) -> Set`
  [page 179] Convert a list into a gb_set
- `from_ordset(List) -> Set`
  [page 179] Make a gb_set from an ordset list
- `insert(Element, Set1) -> Set2`
  [page 179] Add a new element to a gb_set
- `intersection(Set1, Set2) -> Set3`
  [page 180] Return the intersection of two gb_sets
- `intersection(SetList) -> Set`
  [page 180] Return the intersection of a list of gb_sets
- `is_empty(Set) -> bool()`
  [page 180] Test for empty gb_set
- `is_member(Element, Set) -> bool()`
  [page 180] Test for membership of a gb_set

- `is_element(Element, Set) -> bool()`
  [page 180] Test for membership of a gb_set
- `is_set(Set) -> bool()`
  [page 180] Test for a gb_set
- `is_subset(Set1, Set2) -> bool()`
  [page 180] Test for subset
- `iterator(Set) -> Iter`
  [page 180] Return an iterator for a gb_set
- `largest(Set) -> term()`
  [page 181] Return largest element
- `next(Iter1) -> {Element, Iter2 | none}`
  [page 181] Traverse a gb_set with an iterator
- `singleton(Element) -> gb_set()`
  [page 181] Return a gb_set with one element
- `size(Set) -> int()`
  [page 181] Return the number of elements in a gb_set
- `smallest(Set) -> term()`
  [page 181] Return smallest element
- `take_largest(Set1) -> {Element, Set2}`
  [page 181] Extract largest element
- `take_smallest(Set1) -> {Element, Set2}`
  [page 181] Extract smallest element
- `to_list(Set) -> List`
  [page 182] Convert a gb_set into a list
- `union(Set1, Set2) -> Set3`
  [page 182] Return the union of two gb_sets
- `union(SetList) -> Set`
  [page 182] Return the union of a list of gb_sets

gb_trees

The following functions are exported:

- `balance(Tree1) -> Tree2`
  [page 183] Rebalance a tree
- `delete(Key, Tree1) -> Tree2`
  [page 183] Remove a node from a tree
- `delete_any(Key, Tree1) -> Tree2`
  [page 184] Remove a (possibly non-existing) node from a tree
- `empty() -> Tree`
  [page 184] Return an empty tree
- `enter(Key, Val, Tree1) -> Tree2`
  [page 184] Insert or update key with value in a tree
- `from_orddict(List) -> Tree`
  [page 184] Make a tree from an orddict
- `get(Key, Tree) -> Val`
  [page 184] Look up a key in a tree, if present

- `lookup(Key, Tree) -> {value, Val} | none`
  [page 184] Look up a key in a tree
- `insert(Key, Val, Tree1) -> Tree2`
  [page 185] Insert a new key and value in a tree
- `is_defined(Key, Tree) -> bool()`
  [page 185] Test for membership of a tree
- `is_empty(Tree) -> bool()`
  [page 185] Test for empty tree
- `iterator(Tree) -> Iter`
  [page 185] Return an iterator for a tree
- `keys(Tree) -> [Key]`
  [page 185] Return a list of the keys in a tree
- `largest(Tree) -> {Key, Val}`
  [page 185] Return largest key and value
- `next(Iter1) -> {Key, Val, Iter2} | none`
  [page 185] Traverse a tree with an iterator
- `size(Tree) -> int()`
  [page 186] Return the number of nodes in a tree
- `smallest(Tree) -> {Key, Val}`
  [page 186] Return smallest key and value
- `take_largest(Tree1) -> {Key, Val, Tree2}`
  [page 186] Extract largest key and value
- `take_smallest(Tree1) -> {Key, Val, Tree2}`
  [page 186] Extract smallest key and value
- `to_list(Tree) -> [{Key, Val}]`
  [page 186] Convert a tree into a list
- `update(Key, Val, Tree1) -> Tree2`
  [page 186] Update a key to new value in a tree
- `values(Tree) -> [Val]`
  [page 187] Return a list of the values in a tree

gen_event

The following functions are exported:

- `start_link() -> Result`
  [page 189] Create a generic event manager process in a supervision tree.
- `start_link(EventMgrName) -> Result`
  [page 189] Create a generic event manager process in a supervision tree.
- `start() -> Result`
  [page 189] Create a stand-alone event manager process.
- `start(EventMgrName) -> Result`
  [page 189] Create a stand-alone event manager process.
- `add_handler(EventMgrRef, Handler, Args) -> Result`
  [page 190] Add an event handler to a generic event manager.
- `add_sup_handler(EventMgrRef, Handler, Args) -> Result`
  [page 190] Add a supervised event handler to a generic event manager.

- `notify(EventMgrRef, Event) -> ok`
  [page 191] Notify an event manager about an event.
- `sync_notify(EventMgrRef, Event) -> ok`
  [page 191] Notify an event manager about an event.
- `call(EventMgrRef, Handler, Request) -> Result`
  [page 191] Make a synchronous call to a generic event manager.
- `call(EventMgrRef, Handler, Request, Timeout) -> Result`
  [page 191] Make a synchronous call to a generic event manager.
- `delete_handler(EventMgrRef, Handler, Args) -> Result`
  [page 192] Delete an event handler from a generic event manager.
- `swap_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result`
  [page 192] Replace an event handler in a generic event manager.
- `swap_sup_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result`
  [page 193] Replace an event handler in a generic event manager.
- `which_handlers(EventMgrRef) -> [Handler]`
  [page 193] Return all event handlers installed in a generic event manager.
- `stop(EventMgrRef) -> ok`
  [page 194] Terminate a generic event manager.
- `Module:init(InitArgs) -> {ok,State} | {ok,State,hibernate}`
  [page 194] Initialize an event handler.
- `Module:handle_event(Event, State) -> Result`
  [page 194] Handle an event.
- `Module:handle_call(Request, State) -> Result`
  [page 195] Handle a synchronous request.
- `Module:handle_info(Info, State) -> Result`
  [page 195] Handle an incoming message.
- `Module:terminate(Arg, State) -> term()`
  [page 196] Clean up before deletion.
- `Module:code_change(OldVsn, State, Extra) -> {ok, NewState}`
  [page 196] Update the internal state during upgrade/downgrade.

gen_fsm

The following functions are exported:

- `start_link(Module, Args, Options) -> Result`
  [page 199] Create a gen_fsm process in a supervision tree.
- `start_link(FsmName, Module, Args, Options) -> Result`
  [page 199] Create a gen_fsm process in a supervision tree.
- `start(Module, Args, Options) -> Result`
  [page 200] Create a stand-alone gen_fsm process.
- `start(FsmName, Module, Args, Options) -> Result`
  [page 200] Create a stand-alone gen_fsm process.
- `send_event(FsmRef, Event) -> ok`
  [page 200] Send an event asynchronously to a generic FSM.

- `send_all_state_event(FsmRef, Event) -> ok`
  [page 201] Send an event asynchronously to a generic FSM.
- `sync_send_event(FsmRef, Event) -> Reply`
  [page 201] Send an event synchronously to a generic FSM.
- `sync_send_event(FsmRef, Event, Timeout) -> Reply`
  [page 201] Send an event synchronously to a generic FSM.
- `sync_send_all_state_event(FsmRef, Event) -> Reply`
  [page 201] Send an event synchronously to a generic FSM.
- `sync_send_all_state_event(FsmRef, Event, Timeout) -> Reply`
  [page 201] Send an event synchronously to a generic FSM.
- `reply(Caller, Reply) -> true`
  [page 202] Send a reply to a caller.
- `send_event_after(Time, Event) -> Ref`
  [page 202] Send a delayed event internally in a generic FSM.
- `start_timer(Time, Msg) -> Ref`
  [page 202] Send a timeout event internally in a generic FSM.
- `cancel_timer(Ref) -> RemainingTime | false`
  [page 203] Cancel an internal timer in a generic FSM.
- `enter_loop(Module, Options, StateName, StateData)`
  [page 203] Enter the gen_fsm receive loop
- `enter_loop(Module, Options, StateName, StateData, FsmName)`
  [page 203] Enter the gen_fsm receive loop
- `enter_loop(Module, Options, StateName, StateData, Timeout)`
  [page 203] Enter the gen_fsm receive loop
- `enter_loop(Module, Options, StateName, StateData, FsmName, Timeout)`
  [page 203] Enter the gen_fsm receive loop
- `Module:init(Args) -> Result`
  [page 204] Initialize process and internal state name and state data.
- `Module:StateName(Event, StateData) -> Result`
  [page 204] Handle an asynchronous event.
- `Module:handle_event(Event, StateName, StateData) -> Result`
  [page 205] Handle an asynchronous event.
- `Module:StateName(Event, From, StateData) -> Result`
  [page 205] Handle a synchronous event.
- `Module:handle_sync_event(Event, From, StateName, StateData) -> Result`
  [page 206] Handle a synchronous event.
- `Module:handle_info(Info, StateName, StateData) -> Result`
  [page 207] Handle an incoming message.
- `Module:terminate(Reason, StateName, StateData)`
  [page 207] Clean up before termination.
- `Module:code_change(OldVsn, StateName, StateData, Extra) -> {ok, NextStateName, NewStateData}`
  [page 208] Update the internal state data during upgrade/downgrade.

gen_server

The following functions are exported:

- `start_link(Module, Args, Options) -> Result`
  [page 210] Create a gen_server process in a supervision tree.
- `start_link(ServerName, Module, Args, Options) -> Result`
  [page 210] Create a gen_server process in a supervision tree.
- `start(Module, Args, Options) -> Result`
  [page 211] Create a stand-alone gen_server process.
- `start(ServerName, Module, Args, Options) -> Result`
  [page 211] Create a stand-alone gen_server process.
- `call(ServerRef, Request) -> Reply`
  [page 211] Make a synchronous call to a generic server.
- `call(ServerRef, Request, Timeout) -> Reply`
  [page 211] Make a synchronous call to a generic server.
- `multi_call(Name, Request) -> Result`
  [page 212] Make a synchronous call to several generic servers.
- `multi_call(Nodes, Name, Request) -> Result`
  [page 212] Make a synchronous call to several generic servers.
- `multi_call(Nodes, Name, Request, Timeout) -> Result`
  [page 212] Make a synchronous call to several generic servers.
- `cast(ServerRef, Request) -> ok`
  [page 213] Send an asynchronous request to a generic server.
- `abcast(Name, Request) -> abcast`
  [page 213] Send an asynchronous request to several generic servers.
- `abcast(Nodes, Name, Request) -> abcast`
  [page 213] Send an asynchronous request to several generic servers.
- `reply(Client, Reply) -> Result`
  [page 213] Send a reply to a client.
- `enter_loop(Module, Options, State)`
  [page 214] Enter the gen_server receive loop
- `enter_loop(Module, Options, State, ServerName)`
  [page 214] Enter the gen_server receive loop
- `enter_loop(Module, Options, State, Timeout)`
  [page 214] Enter the gen_server receive loop
- `enter_loop(Module, Options, State, ServerName, Timeout)`
  [page 214] Enter the gen_server receive loop
- `Module:init(Args) -> Result`
  [page 215] Initialize process and internal state.
- `Module:handle_call(Request, From, State) -> Result`
  [page 215] Handle a synchronous request.
- `Module:handle_cast(Request, State) -> Result`
  [page 216] Handle an asynchronous request.
- `Module:handle_info(Info, State) -> Result`
  [page 216] Handle an incoming message.

- `Module:terminate(Reason, State)`
  [page 217] Clean up before termination.
- `Module:code_change(OldVsn, State, Extra) -> {ok, NewState}`
  [page 217] Update the internal state during upgrade/downgrade.

io

The following functions are exported:

- `columns([IoDevice]) -> {ok,int()} | {error, enotsup}`
  [page 219] Get the number of columns of a device
- `put_chars([IoDevice,] IoData) -> ok`
  [page 219] Write a list of characters
- `nl([IoDevice]) -> ok`
  [page 219] Write a newline
- `get_chars([IoDevice,] Prompt, Count) -> string() | eof`
  [page 220] Read a specified number of characters
- `get_line([IoDevice,] Prompt) -> string() | eof`
  [page 220] Read a line
- `setopts([IoDevice,] Opts) -> ok | {error, Reason}`
  [page 220] Set options
- `write([IoDevice,] Term) -> ok`
  [page 221] Write a term
- `read([IoDevice,] Prompt) -> Result`
  [page 221] Read a term
- `read(IoDevice, Prompt, StartLine) -> Result`
  [page 221] Read a term
- `fwrite(Format) ->`
  [page 221] Write formatted output
- `fwrite([IoDevice,] Format, Data) -> ok`
  [page 221] Write formatted output
- `format(Format) ->`
  [page 222] Write formatted output
- `format([IoDevice,] Format, Data) -> ok`
  [page 222] Write formatted output
- `fread([IoDevice,] Prompt, Format) -> Result`
  [page 225] Read formatted input
- `rows([IoDevice]) -> {ok,int()} | {error, enotsup}`
  [page 227] Get the number of rows of a device
- `scan_erl_exprs(Prompt) ->`
  [page 227] Read and tokenize Erlang expressions
- `scan_erl_exprs([IoDevice,] Prompt, StartLine) -> Result`
  [page 227] Read and tokenize Erlang expressions
- `scan_erl_form(Prompt) ->`
  [page 227] Read and tokenize an Erlang form
- `scan_erl_form([IoDevice,] Prompt, StartLine) -> Result`
  [page 227] Read and tokenize an Erlang form

- `parse_erl_exprs(Prompt) ->`
  [page 228] Read, tokenize and parse Erlang expressions
- `parse_erl_exprs([IoDevice,] Prompt, StartLine) -> Result`
  [page 228] Read, tokenize and parse Erlang expressions
- `parse_erl_form(Prompt) ->`
  [page 228] Read, tokenize and parse an Erlang form
- `parse_erl_form([IoDevice,] Prompt, StartLine) -> Result`
  [page 228] Read, tokenize and parse an Erlang form

io_lib

The following functions are exported:

- `nl() -> chars()`
  [page 230] Write a newline
- `write(Term) ->`
  [page 230] Write a term
- `write(Term, Depth) -> chars()`
  [page 230] Write a term
- `print(Term) ->`
  [page 230] Pretty print a term
- `print(Term, Column, LineLength, Depth) -> chars()`
  [page 230] Pretty print a term
- `fwrite(Format, Data) ->`
  [page 231] Write formatted output
- `format(Format, Data) -> chars()`
  [page 231] Write formatted output
- `fread(Format, String) -> Result`
  [page 231] Read formatted input
- `fread(Continuation, String, Format) -> Return`
  [page 231] Re-entrant formatted reader
- `write_atom(Atom) -> chars()`
  [page 232] Write an atom
- `write_string(String) -> chars()`
  [page 232] Write a string
- `write_char(Integer) -> chars()`
  [page 232] Write a character
- `indentation(String, StartIndent) -> int()`
  [page 232] Indentation after printing string
- `char_list(Term) -> bool()`
  [page 233] Test for a list of characters
- `deep_char_list(Term) -> bool()`
  [page 233] Test for a deep list of characters
- `printable_list(Term) -> bool()`
  [page 233] Test for a list of printable characters

lib

The following functions are exported:

- `flush_receive() -> void()`
  [page 234] Flush messages
- `error_message(Format, Args) -> ok`
  [page 234] Print error message
- `progname() -> atom()`
  [page 234] Return name of Erlang start script
- `nonl(String1) -> String2`
  [page 234] Remove last newline
- `send(To, Msg)`
  [page 234] Send a message
- `sendw(To, Msg)`
  [page 235] Send a message and wait for an answer

lists

The following functions are exported:

- `all(Pred, List) -> bool()`
  [page 236] Return true if all elements in the list satisfy`Pred`
- `any(Pred, List) -> bool()`
  [page 236] Return true if any of the elements in the list satisfies`Pred`
- `append(ListOfLists) -> List1`
  [page 236] Append a list of lists
- `append(List1, List2) -> List3`
  [page 236] Append two lists
- `concat(Things) -> string()`
  [page 237] Concatenate a list of atoms
- `delete(Elem, List1) -> List2`
  [page 237] Delete an element from a list
- `dropwhile(Pred, List1) -> List2`
  [page 237] Drop elements from a list while a predicate is true
- `duplicate(N, Elem) -> List`
  [page 237] Make N copies of element
- `filter(Pred, List1) -> List2`
  [page 237] Choose elements which satisfy a predicate
- `flatlength(DeepList) -> int()`
  [page 238] Length of flattened deep list
- `flatmap(Fun, List1) -> List2`
  [page 238] Map and flatten in one pass
- `flatten(DeepList) -> List`
  [page 238] Flatten a deep list
- `flatten(DeepList, Tail) -> List`
  [page 238] Flatten a deep list

- `umerge3(List1, List2, List3) -> List4`
  [page 248] Merge three sorted lists, removing duplicates
- `unzip(List1) -> {List2, List3}`
  [page 248] Unzip a list of two-tuples into two lists
- `unzip3(List1) -> {List2, List3, List4}`
  [page 249] Unzip a list of three-tuples into three lists
- `usort(List1) -> List2`
  [page 249] Sort a list, removing duplicates
- `usort(Fun, List1) -> List2`
  [page 249] Sort a list, removing duplicates
- `zip(List1, List2) -> List3`
  [page 249] Zip two lists into a list of two-tuples
- `zip3(List1, List2, List3) -> List4`
  [page 249] Zip three lists into a list of three-tuples
- `zipwith(Combine, List1, List2) -> List3`
  [page 250] Zip two lists into one list according to a fun
- `zipwith3(Combine, List1, List2, List3) -> List4`
  [page 250] Zip three lists into one list according to a fun

log_mf_h

The following functions are exported:

- `init(Dir, MaxBytes, MaxFiles)`
  [page 251] Initiate the event handler
- `init(Dir, MaxBytes, MaxFiles, Pred) -> Args`
  [page 251] Initiate the event handler

math

The following functions are exported:

- `pi() -> float()`
  [page 252] A useful number
- `sin(X)`
  [page 252] Diverse math functions
- `cos(X)`
  [page 252] Diverse math functions
- `tan(X)`
  [page 252] Diverse math functions
- `asin(X)`
  [page 252] Diverse math functions
- `acos(X)`
  [page 252] Diverse math functions
- `atan(X)`
  [page 252] Diverse math functions
- `atan2(Y, X)`
  [page 252] Diverse math functions

- `sinh(X)`
  [page 252] Diverse math functions
- `cosh(X)`
  [page 252] Diverse math functions
- `tanh(X)`
  [page 252] Diverse math functions
- `asinh(X)`
  [page 252] Diverse math functions
- `acosh(X)`
  [page 252] Diverse math functions
- `atanh(X)`
  [page 252] Diverse math functions
- `exp(X)`
  [page 252] Diverse math functions
- `log(X)`
  [page 252] Diverse math functions
- `log10(X)`
  [page 252] Diverse math functions
- `pow(X, Y)`
  [page 252] Diverse math functions
- `sqrt(X)`
  [page 252] Diverse math functions
- `erf(X) -> float()`
  [page 253] Error function.
- `erfc(X) -> float()`
  [page 253] Another error function

ms_transform

The following functions are exported:

- `parse_transform(Forms,_Options) -> Forms`
  [page 263] Transforms Erlang abstract format containing calls to ets/dbg:fun2ms into literal match specifications.
- `transform_from_shell(Dialect,Clauses,BoundEnvironment) -> term()`
  [page 263] Used when transforming fun's created in the shell into match_specifications.
- `format_error(Errcode) -> ErrMessage`
  [page 264] Error formatting function as required by the parse_transform interface.

orddict

The following functions are exported:

- `append(Key, Value, Orddict1) -> Orddict2`
  [page 265] Append a value to keys in a dictionary
- `append_list(Key, ValList, Orddict1) -> Orddict2`
  [page 265] Append new values to keys in a dictionary

- `erase(Key, Orddict1) -> Orddict2`
  [page 265] Erase a key from a dictionary
- `fetch(Key, Orddict) -> Value`
  [page 266] Look-up values in a dictionary
- `fetch_keys(Orddict) -> Keys`
  [page 266] Return all keys in a dictionary
- `filter(Pred, Orddict1) -> Orddict2`
  [page 266] Choose elements which satisfy a predicate
- `find(Key, Orddict) -> {ok, Value} | error`
  [page 266] Search for a key in a dictionary
- `fold(Fun, Acc0, Orddict) -> Acc1`
  [page 266] Fold a function over a dictionary
- `from_list(List) -> Orddict`
  [page 266] Convert a list of pairs to a dictionary
- `is_key(Key, Orddict) -> bool()`
  [page 267] Test if a key is in a dictionary
- `map(Fun, Orddict1) -> Orddict2`
  [page 267] Map a function over a dictionary
- `merge(Fun, Orddict1, Orddict2) -> Orddict3`
  [page 267] Merge two dictionaries
- `new() -> ordered_dictionary()`
  [page 267] Create a dictionary
- `size(Orddict) -> int()`
  [page 267] Return the number of elements in an ordered dictionary
- `store(Key, Value, Orddict1) -> Orddict2`
  [page 267] Store a value in a dictionary
- `to_list(Orddict) -> List`
  [page 268] Convert a dictionary to a list of pairs
- `update(Key, Fun, Orddict1) -> Orddict2`
  [page 268] Update a value in a dictionary
- `update(Key, Fun, Initial, Orddict1) -> Orddict2`
  [page 268] Update a value in a dictionary
- `update_counter(Key, Increment, Orddict1) -> Orddict2`
  [page 268] Increment a value in a dictionary

ordsets

The following functions are exported:

- `new() -> Ordset`
  [page 270] Return an empty set
- `is_set(Ordset) -> bool()`
  [page 270] Test for an `Ordset`
- `size(Ordset) -> int()`
  [page 270] Return the number of elements in a set
- `to_list(Ordset) -> List`
  [page 270] Convert an `Ordset`into a list

- `from_list(List) -> Ordset`
  [page 271] Convert a list into an `Ordset`
- `is_element(Element, Ordset) -> bool()`
  [page 271] Test for membership of an `Ordset`
- `add_element(Element, Ordset1) -> Ordset2`
  [page 271] Add an element to an `Ordset`
- `del_element(Element, Ordset1) -> Ordset2`
  [page 271] Remove an element from an `Ordset`
- `union(Ordset1, Ordset2) -> Ordset3`
  [page 271] Return the union of two `Ordsets`
- `union(OrdsetList) -> Ordset`
  [page 271] Return the union of a list of `Ordsets`
- `intersection(Ordset1, Ordset2) -> Ordset3`
  [page 271] Return the intersection of two `Ordsets`
- `intersection(OrdsetList) -> Ordset`
  [page 272] Return the intersection of a list of `Ordsets`
- `subtract(Ordset1, Ordset2) -> Ordset3`
  [page 272] Return the difference of two `Ordsets`
- `is_subset(Ordset1, Ordset2) -> bool()`
  [page 272] Test for subset
- `fold(Function, Acc0, Ordset) -> Acc1`
  [page 272] Fold over set elements
- `filter(Pred, Ordset1) -> Set2`
  [page 272] Filter set elements

pg

The following functions are exported:

- `create(PgName) -> ok | {error, Reason}`
  [page 273] Create an empty group
- `create(PgName, Node) -> ok | {error, Reason}`
  [page 273] Create an empty group on another node
- `join(PgName, Pid) -> Members`
  [page 273] Join a pid to a process group
- `send(PgName, Msg) -> void()`
  [page 274] Send a message to all members of a process group
- `esend(PgName, Msg) -> void()`
  [page 274] Send a message to all members of a process group, except ourselves
- `members(PgName) -> Members`
  [page 274] Return a list of all members of a process group

pool

The following functions are exported:

- `start(Name) ->`
  [page 275] >Start a new pool
- `start(Name, Args) -> Nodes`
  [page 275] >Start a new pool
- `attach(Node) -> allready_attached | attached`
  [page 275] Ensure that a pool master is running
- `stop() -> stopped`
  [page 276] Stop the pool and kill all the slave nodes
- `get_nodes() -> Nodes`
  [page 276] Return a list of the current member nodes of the pool
- `pspawn(Mod, Fun, Args) -> pid()`
  [page 276] Spawn a process on the pool node with expected lowest future load
- `pspawn_link(Mod, Fun, Args) -> pid()`
  [page 276] Spawn and link to a process on the pool node with expected lowest future load
- `get_node() -> node()`
  [page 276] Return the node with the expected lowest future load

proc_lib

The following functions are exported:

- `spawn(Fun) -> pid()`
  [page 277] Spawn a new process.
- `spawn(Node, Fun) -> pid()`
  [page 277] Spawn a new process.
- `spawn(Module, Function, Args) -> pid()`
  [page 277] Spawn a new process.
- `spawn(Node, Module, Function, Args) -> pid()`
  [page 277] Spawn a new process.
- `spawn_link(Fun) -> pid()`
  [page 277] Spawn and link to a new process.
- `spawn_link(Node, Fun) -> pid()`
  [page 277] Spawn and link to a new process.
- `spawn_link(Module, Function, Args) -> pid()`
  [page 277] Spawn and link to a new process.
- `spawn_link(Node, Module, Function, Args) -> pid()`
  [page 278] Spawn and link to a new process.
- `spawn_opt(Fun, SpawnOpts) -> pid()`
  [page 278] Spawn a new process with given options.
- `spawn_opt(Node, Fun, SpawnOpts) -> pid()`
  [page 278] Spawn a new process with given options.
- `spawn_opt(Module, Function, Args, SpawnOpts) -> pid()`
  [page 278] Spawn a new process with given options.

- `spawn_opt(Node, Module, Func, Args, SpawnOpts) -> pid()`
  [page 278] Spawn a new process with given options.
- `start(Module, Function, Args) -> Ret`
  [page 278] Start a new process synchronously.
- `start(Module, Function, Args, Time) -> Ret`
  [page 278] Start a new process synchronously.
- `start(Module, Function, Args, Time, SpawnOpts) -> Ret`
  [page 278] Start a new process synchronously.
- `start_link(Module, Function, Args) -> Ret`
  [page 278] Start a new process synchronously.
- `start_link(Module, Function, Args, Time) -> Ret`
  [page 278] Start a new process synchronously.
- `start_link(Module, Function, Args, Time, SpawnOpts) -> Ret`
  [page 278] Start a new process synchronously.
- `init_ack(Parent, Ret) -> void()`
  [page 279] Used by a process when it has started.
- `init_ack(Ret) -> void()`
  [page 279] Used by a process when it has started.
- `format(CrashReport) -> string()`
  [page 280] Format a crash report.
- `initial_call(Process) -> {Module,Function,Args} | Fun | false`
  [page 280] Extract the initial call of a `proc_lib`spawned process.
- `translate_initial_call(Process) -> {Module,Function,Arity} | Fun`
  [page 280] Extract and translate the initial call of a `proc_lib`spawned process.
- `hibernate(Module, Function, Args)`
  [page 281] Hibernate a process until a message is sent to it

proplists

The following functions are exported:

- `append_values(Key, List) -> List`
  [page 282]
- `compact(List) -> List`
  [page 282]
- `delete(Key, List) -> List`
  [page 282]
- `expand(Expansions, List) -> List`
  [page 282]
- `get_all_values(Key, List) -> [term()]`
  [page 283]
- `get_bool(Key, List) -> bool()`
  [page 283]
- `get_keys(List) -> [term()]`
  [page 283]
- `get_value(Key, List) -> term()`
  [page 284]

- `get_value(Key, List, Default) -> term()`
  [page 284]
- `is_defined(Key, List) -> bool()`
  [page 284]
- `lookup(Key, List) -> none | tuple()`
  [page 284]
- `lookup_all(Key, List) -> [tuple()]`
  [page 284]
- `normalize(List, Stages) -> List`
  [page 284]
- `property(Property) -> Property`
  [page 285]
- `property(Key, Value) -> Property`
  [page 285]
- `split(List, Keys) -> {Lists, Rest}`
  [page 285]
- `substitute_aliases(Aliases, List) -> List`
  [page 286]
- `substitute_negations(Negations, List) -> List`
  [page 286]
- `unfold(List) -> List`
  [page 286]

qlc

The following functions are exported:

- `append(QHL) -> QH`
  [page 294] Return a query handle.
- `append(QH1, QH2) -> QH3`
  [page 294] Return a query handle.
- `cursor(QueryHandleOrList [, Options]) -> QueryCursor`
  [page 294] Create a query cursor.
- `delete_cursor(QueryCursor) -> ok`
  [page 295] Delete a query cursor.
- `eval(QueryHandleOrList [, Options]) -> Answers | Error`
  [page 295] Return all answers to a query.
- `e(QueryHandleOrList [, Options]) -> Answers`
  [page 295] Return all answers to a query.
- `fold(Function, Acc0, QueryHandleOrList [, Options]) -> Acc1 | Error`
  [page 295] Fold a function over the answers to a query.
- `format_error(Error) -> Chars`
  [page 296] Return an English description of a an error tuple.
- `info(QueryHandleOrList [, Options]) -> Info`
  [page 296] Return code describing a query handle.
- `keysort(KeyPos, QH1 [, SortOptions]) -> QH2`
  [page 297] Return a query handle.

- `next_answers(QueryCursor [, NumberOfAnswers]) -> Answers | Error`
  [page 297] Return some or all answers to a query.
- `q(QueryListComprehension [, Options]) -> QueryHandle`
  [page 298] Return a handle for a query list comprehension.
- `sort(QH1 [, SortOptions]) -> QH2`
  [page 301] Return a query handle.
- `string_to_handle(QueryString [, Options [, Bindings]]) -> QueryHandle | Error`
  [page 301] Return a handle for a query list comprehension.
- `table(TraverseFun, Options) -> QueryHandle`
  [page 301] Return a query handle for a table.

queue

The following functions are exported:

- `new() -> Q`
  [page 306] Create an empty queue
- `is_queue(Term) -> true | false`
  [page 306] Test if a term is a queue
- `is_empty(Q) -> true | false`
  [page 306] Test if a queue is empty
- `len(Q) -> N`
  [page 306] Get the length of a queue
- `in(Item, Q1) -> Q2`
  [page 306] Insert an item at the rear of a queue
- `in_r(Item, Q1) -> Q2`
  [page 306] Insert an item at the front of a queue
- `out(Q1) -> Result`
  [page 306] Remove the front item from a queue
- `out_r(Q1) -> Result`
  [page 307] Remove the rear item from a queue
- `from_list(L) -> queue()`
  [page 307] Convert a list to a queue
- `to_list(Q) -> list()`
  [page 307] Convert a queue to a list
- `reverse(Q1) -> Q2`
  [page 307] Reverse a queue
- `split(N, Q1) -> {Q2,Q3}`
  [page 307] Split a queue in two
- `join(Q1, Q2) -> Q3`
  [page 307] Join two queues
- `filter(Fun, Q1) -> Q2`
  [page 307] Filter a queue
- `get(Q) -> Item`
  [page 308] Return the front item of a queue

- `get_r(Q) -> Item`
  [page 308] Return the rear item of a queue
- `drop(Q1) -> Q2`
  [page 308] Remove the front item from a queue
- `drop_r(Q1) -> Q2`
  [page 308] Remove the rear item from a queue
- `peek(Q) -> {value,Item} | empty`
  [page 309] Return the front item of a queue
- `peek_r(Q) -> {value,Item} | empty`
  [page 309] Return the rear item of a queue
- `cons(Item, Q1) -> Q2`
  [page 309] Insert an item at the head of a queue
- `head(Q) -> Item`
  [page 309] Return the item at the head of a queue
- `tail(Q1) -> Q2`
  [page 309] Remove the head item from a queue
- `snoc(Q1, Item) -> Q2`
  [page 309] Insert an item at the tail of a queue
- `daeh(Q) -> Item`
  [page 310] Return the tail item of a queue
- `last(Q) -> Item`
  [page 310] Return the tail item of a queue
- `liat(Q1) -> Q2`
  [page 310] Remove the tail item from a queue
- `init(Q1) -> Q2`
  [page 310] Remove the tail item from a queue
- `lait(Q1) -> Q2`
  [page 310] Remove the tail item from a queue

random

The following functions are exported:

- `seed() -> ran()`
  [page 311] Seeds random number generation with default values
- `seed(A1, A2, A3) -> ran()`
  [page 311] Seeds random number generator
- `seed0() -> ran()`
  [page 311] Return default state for random number generation
- `uniform()-> float()`
  [page 311] Return a random float
- `uniform(N) -> int()`
  [page 311] Return a random integer
- `uniform_s(State0) -> {float(), State1}`
  [page 312] Return a random float
- `uniform_s(N, State0) -> {int(), State1}`
  [page 312] Return a random integer

re

The following functions are exported:

- `compile(Regexp) -> {ok, MP} | {error, ErrSpec}`
  [page 314] Compile a regular expression into a match program
- `compile(Regexp,Options) -> {ok, MP} | {error, ErrSpec}`
  [page 314] Compile a regular expression into a match program
- `run(Subject,RE) -> {match, Captured} | nomatch | {error, ErrSpec}`
  [page 315] Match a subject against regular expression and capture subpatterns
- `run(Subject,RE,Options) -> {match, Captured} | nomatch | {error, ErrSpec}`
  [page 316] Match a subject against regular expression and capture subpatterns

regexp

The following functions are exported:

- `match(String, RegExp) -> MatchRes`
  [page 349] Match a regular expression
- `first_match(String, RegExp) -> MatchRes`
  [page 349] Match a regular expression
- `matches(String, RegExp) -> MatchRes`
  [page 349] Match a regular expression
- `sub(String, RegExp, New) -> SubRes`
  [page 350] Substitute the first occurrence of a regular expression
- `gsub(String, RegExp, New) -> SubRes`
  [page 350] Substitute all occurrences of a regular expression
- `split(String, RegExp) -> SplitRes`
  [page 350] Split a string into fields
- `sh_to_awk(ShRegExp) -> AwkRegExp`
  [page 351] Convert an shregular expression into an `AWK`one
- `parse(RegExp) -> ParseRes`
  [page 351] Parse a regular expression
- `format_error(ErrorDescriptor) -> Chars`
  [page 351] Format an error descriptor

sets

The following functions are exported:

- `new() -> Set`
  [page 354] Return an empty set
- `is_set(Set) -> bool()`
  [page 354] Test for an `Set`
- `size(Set) -> int()`
  [page 354] Return the number of elements in a set
- `to_list(Set) -> List`
  [page 354] Convert an `Set`into a list

- `from_list(List) -> Set`
  [page 355] Convert a list into an `Set`
- `is_element(Element, Set) -> bool()`
  [page 355] Test for membership of an `Set`
- `add_element(Element, Set1) -> Set2`
  [page 355] Add an element to an `Set`
- `del_element(Element, Set1) -> Set2`
  [page 355] Remove an element from an `Set`
- `union(Set1, Set2) -> Set3`
  [page 355] Return the union of two `Sets`
- `union(SetList) -> Set`
  [page 355] Return the union of a list of `Sets`
- `intersection(Set1, Set2) -> Set3`
  [page 355] Return the intersection of two `Sets`
- `intersection(SetList) -> Set`
  [page 356] Return the intersection of a list of `Sets`
- `subtract(Set1, Set2) -> Set3`
  [page 356] Return the difference of two `Sets`
- `is_subset(Set1, Set2) -> bool()`
  [page 356] Test for subset
- `fold(Function, Acc0, Set) -> Acc1`
  [page 356] Fold over set elements
- `filter(Pred, Set1) -> Set2`
  [page 356] Filter set elements

shell

The following functions are exported:

- `history(N) -> integer()`
  [page 366] Sets the number of previous commands to keep
- `results(N) -> integer()`
  [page 366] Sets the number of previous results to keep
- `catch_exception(Bool) -> Bool`
  [page 366] Sets the exception handling of the shell
- `start_restricted(Module) -> ok | {error, Reason}`
  [page 366] Exits a normal shell and starts a restricted shell.
- `stop_restricted() -> ok`
  [page 367] Exits a restricted shell and starts a normal shell.

shell_default

No functions are exported.

slave

The following functions are exported:

- `start(Host) ->`
  [page 369] Start a slave node on a host
- `start(Host, Name) ->`
  [page 369] Start a slave node on a host
- `start(Host, Name, Args) -> {ok, Node} | {error, Reason}`
  [page 369] Start a slave node on a host
- `start_link(Host) ->`
  [page 370] Start and link to a slave node on a host
- `start_link(Host, Name) ->`
  [page 370] Start and link to a slave node on a host
- `start_link(Host, Name, Args) -> {ok, Node} | {error, Reason}`
  [page 370] Start and link to a slave node on a host
- `stop(Node) -> ok`
  [page 371] Stop (kill) a node
- `pseudo([Master | ServerList]) -> ok`
  [page 371] Start a number of pseudo servers
- `pseudo(Master, ServerList) -> ok`
  [page 371] Start a number of pseudo servers
- `relay(Pid)`
  [page 371] Run a pseudo server

sofs

The following functions are exported:

- `a_function(Tuples [, Type]) -> Function`
  [page 376] Create a function.
- `canonical_relation(SetOfSets) -> BinRel`
  [page 376] Return the canonical map.
- `composite(Function1, Function2) -> Function3`
  [page 376] Return the composite of two functions.
- `constant_function(Set, AnySet) -> Function`
  [page 376] Create the function that maps each element of a set onto another set.
- `converse(BinRel1) -> BinRel2`
  [page 377] Return the converse of a binary relation.
- `difference(Set1, Set2) -> Set3`
  [page 377] Return the difference of two sets.
- `digraph_to_family(Graph [, Type]) -> Family`
  [page 377] Create a family from a directed graph.
- `domain(BinRel) -> Set`
  [page 377] Return the domain of a binary relation.
- `drestriction(BinRel1, Set) -> BinRel2`
  [page 377] Return a restriction of a binary relation.

- `drestriction(SetFun, Set1, Set2) -> Set3`
  [page 378] Return a restriction of a relation.
- `empty_set() -> Set`
  [page 378] Return the untyped empty set.
- `extension(BinRel1, Set, AnySet) -> BinRel2`
  [page 378] Extend the domain of a binary relation.
- `family(Tuples [, Type]) -> Family`
  [page 379] Create a family of subsets.
- `family_difference(Family1, Family2) -> Family3`
  [page 379] Return the difference of two families.
- `family_domain(Family1) -> Family2`
  [page 379] Return a family of domains.
- `family_field(Family1) -> Family2`
  [page 379] Return a family of fields.
- `family_intersection(Family1) -> Family2`
  [page 380] Return the intersection of a family of sets of sets.
- `family_intersection(Family1, Family2) -> Family3`
  [page 380] Return the intersection of two families.
- `family_projection(SetFun, Family1) -> Family2`
  [page 380] Return a family of modified subsets.
- `family_range(Family1) -> Family2`
  [page 380] Return a family of ranges.
- `family_specification(Fun, Family1) -> Family2`
  [page 381] Select a subset of a family using a predicate.
- `family_to_digraph(Family [, GraphType]) -> Graph`
  [page 381] Create a directed graph from a family.
- `family_to_relation(Family) -> BinRel`
  [page 381] Create a binary relation from a family.
- `family_union(Family1) -> Family2`
  [page 382] Return the union of a family of sets of sets.
- `family_union(Family1, Family2) -> Family3`
  [page 382] Return the union of two families.
- `field(BinRel) -> Set`
  [page 382] Return the field of a binary relation.
- `from_external(ExternalSet, Type) -> AnySet`
  [page 382] Create a set.
- `from_sets(ListOfSets) -> Set`
  [page 382] Create a set out of a list of sets.
- `from_sets(TupleOfSets) -> Ordset`
  [page 382] Create an ordered set out of a tuple of sets.
- `from_term(Term [, Type]) -> AnySet`
  [page 383] Create a set.
- `image(BinRel, Set1) -> Set2`
  [page 384] Return the image of a set under a binary relation.
- `intersection(SetOfSets) -> Set`
  [page 384] Return the intersection of a set of sets.

- `intersection(Set1, Set2) -> Set3`
  [page 384] Return the intersection of two sets.
- `intersection_of_family(Family) -> Set`
  [page 384] Return the intersection of a family.
- `inverse(Function1) -> Function2`
  [page 384] Return the inverse of a function.
- `inverse_image(BinRel, Set1) -> Set2`
  [page 385] Return the inverse image of a set under a binary relation.
- `is_a_function(BinRel) -> Bool`
  [page 385] Test for a function.
- `is_disjoint(Set1, Set2) -> Bool`
  [page 385] Test for disjoint sets.
- `is_empty_set(AnySet) -> Bool`
  [page 385] Test for an empty set.
- `is_equal(AnySet1, AnySet2) -> Bool`
  [page 385] Test two sets for equality.
- `is_set(AnySet) -> Bool`
  [page 385] Test for an unordered set.
- `is_sofs_set(Term) -> Bool`
  [page 386] Test for an unordered set.
- `is_subset(Set1, Set2) -> Bool`
  [page 386] Test two sets for subset.
- `is_type(Term) -> Bool`
  [page 386] Test for a type.
- `join(Relation1, I, Relation2, J) -> Relation3`
  [page 386] Return the join of two relations.
- `multiple_relative_product(TupleOfBinRels, BinRel1) -> BinRel2`
  [page 386] Return the multiple relative product of a tuple of binary relations and a relation.
- `no_elements(ASet) -> NoElements`
  [page 387] Return the number of elements of a set.
- `partition(SetOfSets) -> Partition`
  [page 387] Return the coarsest partition given a set of sets.
- `partition(SetFun, Set) -> Partition`
  [page 387] Return a partition of a set.
- `partition(SetFun, Set1, Set2) -> {Set3, Set4}`
  [page 387] Return a partition of a set.
- `partition_family(SetFun, Set) -> Family`
  [page 388] Return a family indexing a partition.
- `product(TupleOfSets) -> Relation`
  [page 388] Return the Cartesian product of a tuple of sets.
- `product(Set1, Set2) -> BinRel`
  [page 388] Return the Cartesian product of two sets.
- `projection(SetFun, Set1) -> Set2`
  [page 389] Return a set of substituted elements.
- `range(BinRel) -> Set`
  [page 389] Return the range of a binary relation.

- `relation(Tuples [, Type]) -> Relation`
  [page 389] Create a relation.
- `relation_to_family(BinRel) -> Family`
  [page 389] Create a family from a binary relation.
- `relative_product(TupleOfBinRels [, BinRel1]) -> BinRel2`
  [page 390] Return the relative product of a tuple of binary relations and a binary relation.
- `relative_product(BinRel1, BinRel2) -> BinRel3`
  [page 390] Return the relative product of two binary relations.
- `relative_product1(BinRel1, BinRel2) -> BinRel3`
  [page 390] Return the relative_product of two binary relations.
- `restriction(BinRel1, Set) -> BinRel2`
  [page 390] Return a restriction of a binary relation.
- `restriction(SetFun, Set1, Set2) -> Set3`
  [page 391] Return a restriction of a set.
- `set(Terms [, Type]) -> Set`
  [page 391] Create a set of atoms or any type of sets.
- `specification(Fun, Set1) -> Set2`
  [page 391] Select a subset using a predicate.
- `strict_relation(BinRel1) -> BinRel2`
  [page 391] Return the strict relation corresponding to a given relation.
- `substitution(SetFun, Set1) -> Set2`
  [page 392] Return a function with a given set as domain.
- `symdiff(Set1, Set2) -> Set3`
  [page 392] Return the symmetric difference of two sets.
- `symmetric_partition(Set1, Set2) -> {Set3, Set4, Set5}`
  [page 393] Return a partition of two sets.
- `to_external(AnySet) -> ExternalSet`
  [page 393] Return the elements of a set.
- `to_sets(ASet) -> Sets`
  [page 393] Return a list or a tuple of the elements of set.
- `type(AnySet) -> Type`
  [page 393] Return the type of a set.
- `union(SetOfSets) -> Set`
  [page 393] Return the union of a set of sets.
- `union(Set1, Set2) -> Set3`
  [page 393] Return the union of two sets.
- `union_of_family(Family) -> Set`
  [page 394] Return the union of a family.
- `weak_relation(BinRel1) -> BinRel2`
  [page 394] Return the weak relation corresponding to a given relation.

string

The following functions are exported:

- `len(String) -> Length`
  [page 395] Return the length of a string
- `equal(String1, String2) -> bool()`
  [page 395] Test string equality
- `concat(String1, String2) -> String3`
  [page 395] Concatenate two strings
- `chr(String, Character) -> Index`
  [page 395] Return the index of the first/last occurrence of`Character`in `String`
- `rchr(String, Character) -> Index`
  [page 395] Return the index of the first/last occurrence of`Character`in `String`
- `str(String, SubString) -> Index`
  [page 395] Find the index of a substring
- `rstr(String, SubString) -> Index`
  [page 395] Find the index of a substring
- `span(String, Chars) -> Length`
  [page 396] Span characters at start of string
- `cspan(String, Chars) -> Length`
  [page 396] Span characters at start of string
- `substr(String, Start) -> SubString`
  [page 396] Return a substring of `String`
- `substr(String, Start, Length) -> Substring`
  [page 396] Return a substring of `String`
- `tokens(String, SeparatorList) -> Tokens`
  [page 396] Split string into tokens
- `join(StringList, Separator) -> String`
  [page 396] Join a list of strings with separator
- `chars(Character, Number) -> String`
  [page 397] Returns a string consisting of numbers of characters
- `chars(Character, Number, Tail) -> String`
  [page 397] Returns a string consisting of numbers of characters
- `copies(String, Number) -> Copies`
  [page 397] Copy a string
- `words(String) -> Count`
  [page 397] Count blank separated words
- `words(String, Character) -> Count`
  [page 397] Count blank separated words
- `sub_word(String, Number) -> Word`
  [page 397] Extract subword
- `sub_word(String, Number, Character) -> Word`
  [page 397] Extract subword
- `strip(String) -> Stripped`
  [page 398] Strip leading or trailing characters
- `strip(String, Direction) -> Stripped`
  [page 398] Strip leading or trailing characters
- `strip(String, Direction, Character) -> Stripped`
  [page 398] Strip leading or trailing characters

- `left(String, Number) -> Left`
  [page 398] Adjust left end of string
- `left(String, Number, Character) -> Left`
  [page 398] Adjust left end of string
- `right(String, Number) -> Right`
  [page 398] Adjust right end of string
- `right(String, Number, Character) -> Right`
  [page 398] Adjust right end of string
- `centre(String, Number) -> Centered`
  [page 398] Center a string
- `centre(String, Number, Character) -> Centered`
  [page 398] Center a string
- `sub_string(String, Start) -> SubString`
  [page 399] Extract a substring
- `sub_string(String, Start, Stop) -> SubString`
  [page 399] Extract a substring
- `to_float(String) -> {Float,Rest} | {error,Reason}`
  [page 399] Returns a float whose text representation is the integers (ASCII values) in String.
- `to_integer(String) -> {Int,Rest} | {error,Reason}`
  [page 399] Returns an integer whose text representation is the integers (ASCII values) in String.
- `to_lower(String) -> Result`
  [page 400] Convert case of string (ISO/IEC 8859-1)
- `to_lower(Char) -> CharResult`
  [page 400] Convert case of string (ISO/IEC 8859-1)
- `to_upper(String) -> Result`
  [page 400] Convert case of string (ISO/IEC 8859-1)
- `to_upper(Char) -> CharResult`
  [page 400] Convert case of string (ISO/IEC 8859-1)

supervisor

The following functions are exported:

- `start_link(Module, Args) -> Result`
  [page 403] Create a supervisor process.
- `start_link(SupName, Module, Args) -> Result`
  [page 403] Create a supervisor process.
- `start_child(SupRef, ChildSpec) -> Result`
  [page 404] Dynamically add a child process to a supervisor.
- `terminate_child(SupRef, Id) -> Result`
  [page 404] Terminate a child process belonging to a supervisor.
- `delete_child(SupRef, Id) -> Result`
  [page 405] Delete a child specification from a supervisor.
- `restart_child(SupRef, Id) -> Result`
  [page 405] Restart a terminated child process belonging to a supervisor.

- `which_children(SupRef) -> [{Id,Child,Type,Modules}]`
  [page 406] Return information about all children specifications and child processes belonging to a supervisor.
- `check_childspecs([ChildSpec]) -> Result`
  [page 406] Check if child specifications are syntactically correct.
- `Module:init(Args) -> Result`
  [page 407] Return a supervisor specification.

supervisor_bridge

The following functions are exported:

- `start_link(Module, Args) -> Result`
  [page 408] Create a supervisor bridge process.
- `start_link(SupBridgeName, Module, Args) -> Result`
  [page 408] Create a supervisor bridge process.
- `Module:init(Args) -> Result`
  [page 409] Initialize process and start subsystem.
- `Module:terminate(Reason, State)`
  [page 409] Clean up and stop subsystem.

sys

The following functions are exported:

- `log(Name,Flag)`
  [page 412] Log system events in memory
- `log(Name,Flag,Timeout) -> ok | {ok, [system_event()]}`
  [page 412] Log system events in memory
- `log_to_file(Name,Flag)`
  [page 412] Log system events to the specified file
- `log_to_file(Name,Flag,Timeout) -> ok | {error, open_file}`
  [page 412] Log system events to the specified file
- `statistics(Name,Flag)`
  [page 412] Enable or disable the collections of statistics
- `statistics(Name,Flag,Timeout) -> ok | {ok, Statistics}`
  [page 412] Enable or disable the collections of statistics
- `trace(Name,Flag)`
  [page 413] Print all system events on `standard_io`
- `trace(Name,Flag,Timeout) -> void()`
  [page 413] Print all system events on `standard_io`
- `no_debug(Name)`
  [page 413] Turn off debugging
- `no_debug(Name,Timeout) -> void()`
  [page 413] Turn off debugging
- `suspend(Name)`
  [page 413] Suspend the process

- `suspend(Name,Timeout) -> void()`
  [page 413] Suspend the process
- `resume(Name)`
  [page 413] Resume a suspended process
- `resume(Name,Timeout) -> void()`
  [page 413] Resume a suspended process
- `change_code(Name, Module, OldVsn, Extra)`
  [page 413] Send the code change system message to the process
- `change_code(Name, Module, OldVsn, Extra, Timeout) -> ok | {error, Reason}`
  [page 413] Send the code change system message to the process
- `get_status(Name)`
  [page 413] Get the status of the process
- `get_status(Name,Timeout) -> {status, Pid, {module, Mod}, [PDict, SysState, Parent, Dbg, Misc]}`
  [page 413] Get the status of the process
- `install(Name,{Func,FuncState})`
  [page 414] Install a debug function in the process
- `install(Name,{Func,FuncState},Timeout)`
  [page 414] Install a debug function in the process
- `remove(Name,Func)`
  [page 414] Remove a debug function from the process
- `remove(Name,Func,Timeout) -> void()`
  [page 414] Remove a debug function from the process
- `debug_options(Options) -> [dbg_opt()]`
  [page 415] Convert a list of options to a debug structure
- `get_debug(Item,Debug,Default) -> term()`
  [page 415] Get the data associated with a debug option
- `handle_debug([dbg_opt()],FormFunc,Extra,Event) -> [dbg_opt()]`
  [page 415] Generate a system event
- `handle_system_msg(Msg,From,Parent,Module,Debug,Misc)`
  [page 415] Take care of system messages
- `print_log(Debug) -> void()`
  [page 416] Print the logged events in the debug structure
- `Mod:system_continue(Parent, Debug, Misc)`
  [page 416] Called when the process should continue its execution
- `Mod:system_terminate(Reason, Parent, Debug, Misc)`
  [page 416] Called when the process should terminate
- `Mod:system_code_change(Misc, Module, OldVsn, Extra) -> {ok, NMisc}`
  [page 416] Called when the process should perform a code change

timer

The following functions are exported:

- `start() -> ok`
  [page 418] Start a global timer server (named `timer_server`).

- apply_after(Time, Module, Function, Arguments) -> {ok, Tref} | {error, Reason}
  [page 418] Apply Module:Function(Arguments) after a specified Time.
- send_after(Time, Pid, Message) -> {ok, TRef} | {error,Reason}
  [page 418] Send Message to Pid after a specified Time.
- send_after(Time, Message) -> {ok, TRef} | {error,Reason}
  [page 418] Send Message to Pid after a specified Time.
- exit_after(Time, Pid, Reason1) -> {ok, TRef} | {error,Reason2}
  [page 419] Send an exit signal with Reason after a specified Time.
- exit_after(Time, Reason1) -> {ok, TRef} | {error,Reason2}
  [page 419] Send an exit signal with Reason after a specified Time.
- kill_after(Time, Pid)-> {ok, TRef} | {error,Reason2}
  [page 419] Send an exit signal with Reason after a specified Time.
- kill_after(Time) -> {ok, TRef} | {error,Reason2}
  [page 419] Send an exit signal with Reason after a specified Time.
- apply_interval(Time, Module, Function, Arguments) -> {ok, TRef} | {error, Reason}
  [page 419] Evaluate Module:Function(Arguments) repeatedly at intervals of Time.
- send_interval(Time, Pid, Message) -> {ok, TRef} | {error, Reason}
  [page 419] Send Message repeatedly at intervals of Time.
- send_interval(Time, Message) -> {ok, TRef} | {error, Reason}
  [page 419] Send Message repeatedly at intervals of Time.
- cancel(TRef) -> {ok, cancel} | {error, Reason}
  [page 419] Cancel a previously requested timeout identified by TRef.
- sleep(Time) -> ok
  [page 419] Suspend the calling process for Time amount of milliseconds.
- tc(Module, Function, Arguments) -> {Time, Value}
  [page 420] Measure the real time it takes to evaluate apply(Module, Function, Arguments)
- now_diff(T2, T1) -> Tdiff
  [page 420] Calculate time difference between now/0 timestamps
- seconds(Seconds) -> Milliseconds
  [page 420] Convert Seconds to Milliseconds.
- minutes(Minutes) -> Milliseconds
  [page 420] Converts Minutes to Milliseconds.
- hours(Hours) -> Milliseconds
  [page 420] Convert Hours to Milliseconds.
- hms(Hours, Minutes, Seconds) -> Milliseconds
  [page 420] Convert Hours+Minutes+Seconds to Milliseconds.

win32reg

The following functions are exported:

- change_key(RegHandle, Key) -> ReturnValue
  [page 423] Move to a key in the registry

- `change_key_create(RegHandle, Key) -> ReturnValue`
  [page 423] Move to a key, create it if it is not there
- `close(RegHandle)-> ReturnValue`
  [page 423] Close the registry.
- `current_key(RegHandle) -> ReturnValue`
  [page 423] Return the path to the current key.
- `delete_key(RegHandle) -> ReturnValue`
  [page 423] Delete the current key
- `delete_value(RegHandle, Name) -> ReturnValue`
  [page 424] Delete the named value on the current key.
- `expand(String) -> ExpandedString`
  [page 424] Expand a string with environment variables
- `format_error(ErrorId) -> ErrorString`
  [page 424] Convert an POSIX errorcode to a string
- `open(OpenModeList)-> ReturnValue`
  [page 424] Open the registry for reading or writing
- `set_value(RegHandle, Name, Value) -> ReturnValue`
  [page 424] Set value at the current registry key with specified name.
- `sub_keys(RegHandle) -> ReturnValue`
  [page 425] Get subkeys to the current key.
- `value(RegHandle, Name) -> ReturnValue`
  [page 425] Get the named value on the current key.
- `values(RegHandle) -> ReturnValue`
  [page 425] Get all values on the current key.

zip

The following functions are exported:

- `zip(Name, FileList) -> RetValue`
  [page 427] Create a zip archive with options
- `zip(Name, FileList, Options) -> RetValue`
  [page 427] Create a zip archive with options
- `create(Name, FileList) -> RetValue`
  [page 427] Create a zip archive with options
- `create(Name, FileList, Options) -> RetValue`
  [page 427] Create a zip archive with options
- `unzip(Archive) -> RetValue`
  [page 428] Extract files from a zip archive
- `unzip(Archive, Options) -> RetValue`
  [page 428] Extract files from a zip archive
- `extract(Archive) -> RetValue`
  [page 428] Extract files from a zip archive
- `extract(Archive, Options) -> RetValue`
  [page 428] Extract files from a zip archive
- `list_dir(Archive) -> RetValue`
  [page 429] Retrieve the name of all files in a zip archive

- `list_dir(Archive, Options)`
  [page 429] Retrieve the name of all files in a zip archive
- `table(Archive) -> RetValue`
  [page 429] Retrieve the name of all files in a zip archive
- `table(Archive, Options)`
  [page 429] Retrieve the name of all files in a zip archive
- `t(Archive)`
  [page 429] Print the name of each file in a zip archive
- `tt(Archive)`
  [page 429] Print name and information for each file in a zip archive
- `zip_open(Archive) -> {ok, ZipHandle} | {error, Reason}`
  [page 430] Open an archive and return a handle to it
- `zip_open(Archive, Options) -> {ok, ZipHandle} | {error, Reason}`
  [page 430] Open an archive and return a handle to it
- `zip_list_dir(ZipHandle) -> Result | {error, Reason}`
  [page 430] Return a table of files in open zip archive
- `zip_get(ZipHandle) -> {ok, [Result]} | {error, Reason}`
  [page 430] Extract files from an open archive
- `zip_get(FileName, ZipHandle) -> {ok, Result} | {error, Reason}`
  [page 430] Extract files from an open archive
- `zip_close(ZipHandle) -> ok | {error, einval}`
  [page 430] Close an open archive

STDLIB

Application

The STDLIB is mandatory in the sense that the minimal system based on Erlang/OTP consists of Kernel and STDLIB. The STDLIB application contains no services.

Configuration

The following configuration parameters are defined for the STDLIB application. See `app(4)` for more information about configuration parameters.

`shell_esc = icl | abort` This parameter can be used to alter the behaviour of the Erlang shell when ^G is pressed.

`restricted_shell = module()` This parameter can be used to run the Erlang shell in restricted mode.

`shell_catch_exception = bool()` This parameter can be used to set the exception handling of the Erlang shell's evaluator process.

`shell_history_length = integer() >= 0` This parameter can be used to determine how many commands are saved by the Erlang shell.

`shell_saved_results = integer() >= 0` This parameter can be used to determine how many results are saved by the Erlang shell.

See Also

[app(4)], [application(3)], shell(3) [page 357],

array

Erlang Module

Functional, extendible arrays. Arrays can have fixed size, or can grow automatically as needed. A default value is used for entries that have not been explicitly set.

Arrays uses *zero* based indexing. This is a deliberate design choice and differs from other erlang datastructures, e.g. tuples.

Unless specified by the user when the array is created, the default value is the atom `undefined`. There is no difference between an unset entry and an entry which has been explicitly set to the same value as the default one (cf. reset/2 [page 61]). If you need to differentiate between unset and set entries, you must make sure that the default value cannot be confused with the values of set entries.

The array never shrinks automatically; if an index `I` has been used successfully to set an entry, all indices in the range [0,`I`] will stay accessible unless the array size is explicitly changed by calling resize/2 [page 61].

Examples:

```
%% Create a fixed-size array with entries 0-9 set to 'undefined'
A0 = array:new(10).
10 = array:size(A0).

%% Create an extendible array and set entry 17 to 'true',
%% causing the array to grow automatically
A1 = array:set(17, true, array:new()).
18 = array:size(A1).

%% Read back a stored value
true = array:get(17, A1).

%% Accessing an unset entry returns the default value
undefined = array:get(3, A1).

%% Accessing an entry beyond the last set entry also returns the
%% default value, if the array does not have fixed size
undefined = array:get(18, A1).

%% "sparse" functions ignore default-valued entries
A2 = array:set(4, false, A1).
[{4, false}, {17, true}] = array:sparse_to_orddict(A2).

%% An extendible array can be made fixed-size later
A3 = array:fix(A2).

%% A fixed-size array does not grow automatically and does not
%% allow accesses beyond the last set entry
```

```
{'EXIT',{badarg,_}} = (catch array:set(18, true, A3)).
{'EXIT',{badarg,_}} = (catch array:get(18, A3)).
```

DATA TYPES

`array()` A functional, extendible array. The representation is not documented and is subject to change without notice. Note that arrays cannot be directly compared for equality.

Exports

`default(Array::array()) -> term()`

Get the value used for uninitialized entries.

See also: new/2 [page 60].

`fix(Array::array()) -> array()`

Fix the size of the array. This prevents it from growing automatically upon insertion; see also set/3 [page 61].

See also: relax/1 [page 60].

`foldl(Function, InitialAcc::term(), Array::array()) -> term()`

Types:

- Function = (Index::integer(), Value::term(), Acc::term()) -> term()

Fold the elements of the array using the given function and initial accumulator value. The elements are visited in order from the lowest index to the highest. If `Function` is not a function, the call fails with reason `badarg`.

See also: foldr/3 [page 58], map/2 [page 59], sparse_foldl/3 [page 61].

`foldr(Function, InitialAcc::term(), Array::array()) -> term()`

Types:

- Function = (Index::integer(), Value::term(), Acc::term()) -> term()

Fold the elements of the array right-to-left using the given function and initial accumulator value. The elements are visited in order from the highest index to the lowest. If `Function` is not a function, the call fails with reason `badarg`.

See also: foldl/3 [page 58], map/2 [page 59].

`from_list(List::list()) -> array()`

Equivalent to from_list(List, undefined) [page 59].

`from_list(List::list(), Default::term()) -> array()`

Convert a list to an extendible array. `Default` is used as the value for uninitialized entries of the array. If `List` is not a proper list, the call fails with reason `badarg`.

See also: new/2 [page 60], to_list/1 [page 62].

`from_orddict(Orddict::list()) -> array()`

Equivalent to from_orddict(Orddict, undefined) [page 59].

`from_orddict(List::list(), Default::term()) -> array()`

Convert an ordered list of pairs {`Index, Value`} to a corresponding extendible array. `Default` is used as the value for uninitialized entries of the array. If `List` is not a proper, ordered list of pairs whose first elements are nonnegative integers, the call fails with reason `badarg`.

See also: new/2 [page 60], to_orddict/1 [page 62].

`get(I::integer(), Array::array()) -> term()`

Get the value of entry `I`. If `I` is not a nonnegative integer, or if the array has fixed size and `I` is larger than the maximum index, the call fails with reason `badarg`.

If the array does not have fixed size, this function will return the default value for any index `I` greater than `size(Array)-1`.

See also: set/3 [page 61].

`is_array(X::term()) -> bool()`

Returns `true` if `X` appears to be an array, otherwise `false`. Note that the check is only shallow; there is no guarantee that `X` is a well-formed array representation even if this function returns `true`.

`is_fix(Array::array()) -> bool()`

Check if the array has fixed size. Returns `true` if the array is fixed, otherwise `false`.

See also: fix/1 [page 58].

`map(Function, Array::array()) -> array()`

Types:

- Function = (Index::integer(), Value::term()) -> term()

Map the given function onto each element of the array. The elements are visited in order from the lowest index to the highest. If `Function` is not a function, the call fails with reason `badarg`.

See also: foldl/3 [page 58], foldr/3 [page 58], sparse_map/2 [page 62].

`new() -> array()`

Create a new, extendible array with initial size zero.

See also: new/1 [page 60], new/2 [page 60].

`new(Options::term()) -> array()`

Create a new array according to the given options. By default, the array is extendible and has initial size zero. Array indices start at 0.

`Options` is a single term or a list of terms, selected from the following:

`N::integer()` or `{size, N::integer()}` Specifies the initial size of the array; this also implies `{fixed, true}`. If `N` is not a nonnegative integer, the call fails with reason `badarg`.

`fixed` or `{fixed, true}` Creates a fixed-size array; see also fix/1 [page 58].

`{fixed, false}` Creates an extendible (non fixed-size) array.

`{default, Value}` Sets the default value for the array to `Value`.

Options are processed in the order they occur in the list, i.e., later options have higher precedence.

The default value is used as the value of uninitialized entries, and cannot be changed once the array has been created.

Examples:

```
array:new(100)
```

creates a fixed-size array of size 100.

```
array:new({default,0})
```

creates an empty, extendible array whose default value is 0.

```
array:new([{size,10},{fixed,false},{default,-1}])
```

creates an extendible array with initial size 10 whose default value is -1.

See also: fix/1 [page 58], from_list/2 [page 59], get/2 [page 59], new/0 [page 59], new/2 [page 60], set/3 [page 61].

`new(Size::integer(), Options::term()) -> array()`

Create a new array according to the given size and options. If `Size` is not a nonnegative integer, the call fails with reason `badarg`. By default, the array has fixed size. Note that any size specifications in `Options` will override the `Size` parameter.

If `Options` is a list, this is simply equivalent to `new([{size, Size} | Options]`, otherwise it is equivalent to `new([{size, Size} | [Options]]`. However, using this function directly is more efficient.

Example:

```
array:new(100, {default,0})
```

creates a fixed-size array of size 100, whose default value is 0.

See also: new/1 [page 60].

`relax(Array::array()) -> array()`

Make the array resizable. (Reverses the effects of fix/1 [page 58].)

See also: fix/1 [page 58].

`reset(I::integer(), Array::array()) -> array()`

Reset entry `I` to the default value for the array. This is equivalent to `set(I, default(Array), Array)`, and hence may cause the array to grow in size, but will not shrink it. Shrinking can be done explicitly by calling resize/2 [page 61].

If `I` is not a nonnegative integer, or if the array has fixed size and `I` is larger than the maximum index, the call fails with reason `badarg`; cf. set/3 [page 61]

See also: new/2 [page 60], set/3 [page 61].

`resize(Array::array()) -> array()`

Change the size of the array to that reported by sparse_size/1 [page 62]. If the given array has fixed size, the resulting array will also have fixed size.

See also: resize/2 [page 61], sparse_size/1 [page 62].

`resize(Size::integer(), Array::array()) -> array()`

Change the size of the array. If `Size` is not a nonnegative integer, the call fails with reason `badarg`. If the given array has fixed size, the resulting array will also have fixed size.

`set(I::integer(), Value::term(), Array::array()) -> array()`

Set entry `I` of the array to `Value`. If `I` is not a nonnegative integer, or if the array has fixed size and `I` is larger than the maximum index, the call fails with reason `badarg`.

If the array does not have fixed size, and `I` is greater than `size(Array)-1`, the array will grow to size `I+1`.

See also: get/2 [page 59], reset/2 [page 61].

`size(Array::array()) -> integer()`

Get the number of entries in the array. Entries are numbered from 0 to `size(Array)-1`; hence, this is also the index of the first entry that is guaranteed to not have been previously set.

See also: set/3 [page 61], sparse_size/1 [page 62].

`sparse_foldl(Function, InitialAcc::term(), Array::array()) -> term()`

Types:

- Function = (Index::integer(), Value::term(), Acc::term()) -> term()

Fold the elements of the array using the given function and initial accumulator value, skipping default-valued entries. The elements are visited in order from the lowest index to the highest. If `Function` is not a function, the call fails with reason `badarg`.

See also: foldl/3 [page 58], sparse_foldr/3 [page 62].

`sparse_foldr(Function, InitialAcc::term(), Array::array()) -> term()`

Types:

- Function = (Index::integer(), Value::term(), Acc::term()) -> term()

Fold the elements of the array right-to-left using the given function and initial accumulator value, skipping default-valued entries. The elements are visited in order from the highest index to the lowest. If `Function` is not a function, the call fails with reason `badarg`.

See also: foldr/3 [page 58], sparse_foldl/3 [page 61].

`sparse_map(Function, Array::array()) -> array()`

Types:

- Function = (Index::integer(), Value::term()) -> term()

Map the given function onto each element of the array, skipping default-valued entries. The elements are visited in order from the lowest index to the highest. If `Function` is not a function, the call fails with reason `badarg`.

See also: map/2 [page 59].

`sparse_size(A::array()) -> integer()`

Get the number of entries in the array up until the last non-default valued entry. In other words, returns `I+1` if `I` is the last non-default valued entry in the array, or zero if no such entry exists.

See also: resize/1 [page 61], size/1 [page 61].

`sparse_to_list(Array::array()) -> list()`

Converts the array to a list, skipping default-valued entries.

See also: to_list/1 [page 62].

`sparse_to_orddict(Array::array()) -> [{Index::integer(), Value::term()}]`

Convert the array to an ordered list of pairs {`Index, Value`}, skipping default-valued entries.

See also: to_orddict/1 [page 62].

`to_list(Array::array()) -> list()`

Converts the array to a list.

See also: from_list/2 [page 59], sparse_to_list/1 [page 62].

`to_orddict(Array::array()) -> [{Index::integer(), Value::term()}]`

Convert the array to an ordered list of pairs {`Index, Value`}.

See also: from_orddict/2 [page 59], sparse_to_orddict/1 [page 62].

base64

Erlang Module

Implements base 64 encode and decode, see RFC2045.

Exports

```
encode(Data) -> Base64
encode_to_string(Data) -> Base64String
```

Types:

- Data = string() | binary()
- Base64 = binary()
- Base64String = string()

Encodes a plain ASCII string into base64. The result will be 33% larger than the data.

```
decode(Base64) -> Data
decode_to_string(Base64) -> DataString
mime_decode(Base64) -> Data
mime_decode_to_string(Base64) -> DataString
```

Types:

- Base64 = string() | binary()
- Data = binary()
- DataString = string()

Decodes a base64 encoded string to plain ASCII. The string should only consist of characters in the base64 set, see RFC4648. `mime_decode/1` and `mime_decode_to_string/1` strips away illegal characters, while `decode/1` and `decode_to_string/1` fails if an illegal charactrer is found.

beam_lib

Erlang Module

beam_lib provides an interface to files created by the BEAM compiler ("BEAM files"). The format used, a variant of "EA IFF 1985" Standard for Interchange Format Files, divides data into chunks.

Chunk data can be returned as binaries or as compound terms. Compound terms are returned when chunks are referenced by names (atoms) rather than identifiers (strings). The names recognized and the corresponding identifiers are:

- abstract_code ("Abst")
- attributes ("Attr")
- compile_info ("CInf")
- exports ("ExpT")
- labeled_exports ("ExpT")
- imports ("ImpT")
- indexed_imports ("ImpT")
- locals ("LocT")
- labeled_locals ("LocT")
- atoms ("Atom")

Debug Information/Abstract Code

The option debug_info can be given to the compiler (see [compile(3)]) in order to have debug information in the form of abstract code (see [The Abstract Format] in ERTS User's Guide) stored in the abstract_code chunk. Tools such as Debugger and Xref require the debug information to be included.

> **Warning:**
> Source code can be reconstructed from the debug information. Use encrypted debug information (see below) to prevent this.

The debug information can also be removed from BEAM files using strip/1 [page 69], strip_files/1 [page 69] and/or strip_release/1 [page 70].

Reconstructing source code

Here is an example of how to reconstruct source code from the debug information in a BEAM file Beam:

```
{ok,{_,[{abstract_code,{_,AC}}]}} = beam_lib:chunks(Beam,[abstract_code]).
io:fwrite("~s~n", [erl_prettypr:format(erl_syntax:form_list(AC))]).
```

Encrypted debug information

The debug information can be encrypted in order to keep the source code secret, but still being able to use tools such as Xref or Debugger.

To use encrypted debug information, a key must be provided to the compiler and `beam_lib`. The key is given as a string and it is recommended that it contains at least 32 characters and that both upper and lower case letters as well as digits and special characters are used.

The default type – and currently the only type – of crypto algorithm is `des3_cbc`, three rounds of DES. The key string will be scrambled using `erlang:md5/1` to generate the actual keys used for `des3_cbc`.

> **Note:**
> As far as we know when by the time of writing, it is infeasible to break `des3_cbc` encryption without any knowledge of the key. Therefore, as long as the key is kept safe and is unguessable, the encrypted debug information *should* be safe from intruders.

There are two ways to provide the key:

1. Use the compiler option {`debug_info,Key`}, see [compile(3)], and the function crypto_key_fun/1 [page 70] to register a fun which returns the key whenever `beam_lib` needs to decrypt the debug information.
   If no such fun is registered, `beam_lib` will instead search for a `.erlang.crypt` file, see below.
2. Store the key in a text file named `.erlang.crypt`.
   In this case, the compiler option `encrypt_debug_info` can be used, see [compile(3)].

.erlang.crypt

`beam_lib` searches for `.erlang.crypt` in the current directory and then the home directory for the current user. If the file is found and contains a key, `beam_lib` will implicitly create a crypto key fun and register it.

The `.erlang.crypt` file should contain a single list of tuples:

```
{debug_info, Mode, Module, Key}
```

`Mode` is the type of crypto algorithm; currently, the only allowed value thus is `des3_cbc`. `Module` is either an atom, in which case `Key` will only be used for the module `Module`, or `[]`, in which case `Key` will be used for all modules. `Key` is the non-empty key string.

The `Key` in the first tuple where both `Mode` and `Module` matches will be used.

Here is an example of an `.erlang.crypt` file that returns the same key for all modules:

```
[{debug_info, des3_cbc, [], "%>7}|pc/DM6Cga*68$Mw]L#&_Gejr]G^"}].
```

And here is a slightly more complicated example of an `.erlang.crypt` which provides one key for the module `t`, and another key for all other modules:

```
[{debug_info, des3_cbc, t, "My KEY"},
 {debug_info, des3_cbc, [], "%>7}|pc/DM6Cga*68$Mw]L#&_Gejr]G^"}].
```

> Note:
> Do not use any of the keys in these examples. Use your own keys.

DATA TYPES

```
beam() -> Module | Filename | binary()
  Module = atom()
  Filename = string() | atom()
```

Each of the functions described below accept either the module name, the filename, or a binary containing the beam module.

```
chunkdata() = {ChunkId, DataB} | {ChunkName, DataT}
  ChunkId = chunkid()
  DataB = binary()
  {ChunkName, DataT} =
        {abstract_code, AbstractCode}
      | {attributes, [{Attribute, [AttributeValue]}]}
      | {compile_info, [{InfoKey, [InfoValue]}]}
      | {exports, [{Function, Arity}]}
      | {labeled_exports, [{Function, Arity, Label}]}
      | {imports, [{Module, Function, Arity}]}
      | {indexed_imports, [{Index, Module, Function, Arity}]}
      | {locals, [{Function, Arity}]}]}
      | {labeled_locals, [{Function, Arity, Label}]}]}
      | {atoms, [{integer(), atom()}]}
  AbstractCode = {AbstVersion, Forms} | no_abstract_code
    AbstVersion = atom()
  Attribute = atom()
  AttributeValue = term()
  Module = Function = atom()
  Arity = int()
  Label = int()
```

It is not checked that the forms conform to the abstract format indicated by `AbstVersion`. `no_abstract_code` means that the `"Abst"` chunk is present, but empty.

The list of attributes is sorted on `Attribute`, and each attribute name occurs once in the list. The attribute values occur in the same order as in the file. The lists of functions are also sorted.

```
chunkid() = "Abst" | "Attr" | "CInf"
            | "ExpT" | "ImpT" | "LocT"
            | "Atom"

chunkname() = abstract_code | attributes | compile_info
            | exports | labeled_exports
            | imports | indexed_imports
```

```
                | locals | labeled_locals
                | atoms

chunkref() = chunkname() | chunkid()
```

Exports

`chunks(Beam, [ChunkRef]) -> {ok, {Module, [ChunkData]}} | {error, beam_lib, Reason}`

Types:

- Beam = beam()
- ChunkRef = chunkref()
- Module = atom()
- ChunkData = chunkdata()
- Reason = {unknown_chunk, Filename, atom()}
- | {key_missing_or_invalid, Filename, abstract_code}
- | Reason1 – see info/1
- Filename = string()

Reads chunk data for selected chunks refs. The order of the returned list of chunk data is determined by the order of the list of chunks references.

`chunks(Beam, [ChunkRef], [Option]) -> {ok, {Module, [ChunkResult]}} | {error, beam_lib, Reason}`

Types:

- Beam = beam()
- ChunkRef = chunkref()
- Module = atom()
- Option = allow_missing_chunks
- ChunkResult = {chunkref(), ChunkContents} | {chunkref(), missing_chunk}
- Reason = {missing_chunk, Filename, atom()}
- | {key_missing_or_invalid, Filename, abstract_code}
- | Reason1 – see info/1
- Filename = string()

Reads chunk data for selected chunks refs. The order of the returned list of chunk data is determined by the order of the list of chunks references.

By default, if any requested chunk is missing in `Beam`, an `error` tuple is returned. However, if the option `allow_missing_chunks` has been given, a result will be returned even if chunks are missing. In the result list, any missing chunks will be represented as {`ChunkRef`,`missing_chunk`}. Note, however, that if the `"Atom"` chunk if missing, that is considered a fatal error and the return value will be an `error` tuple.

`version(Beam) -> {ok, {Module, [Version]}} | {error, beam_lib, Reason}`

Types:

- Beam = beam()
- Module = atom()

- Version = term()
- Reason – see chunks/2

Returns the module version(s). A version is defined by the module attribute `-vsn(Vsn)`. If this attribute is not specified, the version defaults to the checksum of the module. Note that if the version `Vsn` is not a list, it is made into one, that is `{ok,{Module,[Vsn]}}` is returned. If there are several `-vsn` module attributes, the result is the concatenated list of versions. Examples:

```
1> beam_lib:version(a). % -vsn(1).
{ok,{a,[1]}}
2> beam_lib:version(b). % -vsn([1]).
{ok,{b,[1]}}
3> beam_lib:version(c). % -vsn([1]). -vsn(2).
{ok,{c,[1,2]}}
4> beam_lib:version(d). % no -vsn attribute
{ok,{d,[275613208176997377698094100858909383631]}}
```

`md5(Beam) -> {ok, {Module, MD5}} | {error, beam_lib, Reason}`

Types:

- Beam = beam()
- Module = atom()
- MD5 = binary()
- Reason – see chunks/2

Calculates an MD5 redundancy check for the code of the module (compilation date and other attributes are not included).

`info(Beam) -> [{Item, Info}] | {error, beam_lib, Reason1}`

Types:

- Beam = beam()
- Item, Info – see below
- Reason1 = {chunk_too_big, Filename, ChunkId, ChunkSize, FileSize}
- | {invalid_beam_file, Filename, Pos}
- | {invalid_chunk, Filename, ChunkId}
- | {missing_chunk, Filename, ChunkId}
- | {not_a_beam_file, Filename}
- | {file_error, Filename, Posix}
- Filename = string()
- ChunkId = chunkid()
- ChunkSize = FileSize = int()
- Pos = int()
- Posix = posix() – see file(3)

Returns a list containing some information about a BEAM file as tuples `{Item, Info}`:

`{file, Filename} | {binary, Binary}` The name (string) of the BEAM file, or the binary from which the information was extracted.

`{module, Module}` The name (atom) of the module.

{chunks, [{ChunkId, Pos, Size}]} For each chunk, the identifier (string) and the position and size of the chunk data, in bytes.

`cmp(Beam1, Beam2) -> ok | {error, beam_lib, Reason}`

Types:

- Beam1 = Beam2 = beam()
- Reason = {modules_different, Module1, Module2}
- | {chunks_different, ChunkId}
- | Reason1 – see info/1
- Module1 = Module2 = atom()
- ChunkId = chunkid()

Compares the contents of two BEAM files. If the module names are the same, and the chunks with the identifiers `"Code"`, `"ExpT"`, `"ImpT"`, `"StrT"`, and `"Atom"` have the same contents in both files, `ok` is returned. Otherwise an error message is returned.

`cmp_dirs(Dir1, Dir2) -> {Only1, Only2, Different} | {error, beam_lib, Reason1}`

Types:

- Dir1 = Dir2 = string() | atom()
- Different = [{Filename1, Filename2}]
- Only1 = Only2 = [Filename]
- Filename = Filename1 = Filename2 = string()
- Reason1 = {not_a_directory, term()} | – see info/1

The `cmp_dirs/2` function compares the BEAM files in two directories. Only files with extension `".beam"` are compared. BEAM files that exist in directory `Dir1` (`Dir2`) only are returned in `Only1` (`Only2`). BEAM files that exist on both directories but are considered different by `cmp/2` are returned as pairs {`Filename1`, `Filename2`} where `Filename1` (`Filename2`) exists in directory `Dir1` (`Dir2`).

`diff_dirs(Dir1, Dir2) -> ok | {error, beam_lib, Reason1}`

Types:

- Dir1 = Dir2 = string() | atom()
- Reason1 = {not_a_directory, term()} | – see info/1

The `diff_dirs/2` function compares the BEAM files in two directories the way `cmp_dirs/2` does, but names of files that exist in only one directory or are different are presented on standard output.

`strip(Beam1) -> {ok, {Module, Beam2}} | {error, beam_lib, Reason1}`

Types:

- Beam1 = Beam2 = beam()
- Module = atom()
- Reason1 – see info/1

The `strip/1` function removes all chunks from a BEAM file except those needed by the loader. In particular, the debug information (`abstract_code` chunk) is removed.

`strip_files(Files) -> {ok, [{Module, Beam2}]} | {error, beam_lib, Reason1}`

Types:

- Files = [Beam1]
- Beam1 = beam()
- Module = atom()
- Beam2 = beam()
- Reason1 – see info/1

The `strip_files/1` function removes all chunks except those needed by the loader from BEAM files. In particular, the debug information (`abstract_code` chunk) is removed. The returned list contains one element for each given file name, in the same order as in `Files`.

`strip_release(Dir) -> {ok, [{Module, Filename]}} | {error, beam_lib, Reason1}`

Types:

- Dir = string() | atom()
- Module = atom()
- Filename = string()
- Reason1 = {not_a_directory, term()} | – see info/1

The `strip_release/1` function removes all chunks except those needed by the loader from the BEAM files of a release. `Dir` should be the installation root directory. For example, the current OTP release can be stripped with the call `beam_lib:strip_release(code:root_dir())`.

`format_error(Reason) -> Chars`

Types:

- Reason – see other functions
- Chars = [char() | Chars]

Given the error returned by any function in this module, the function `format_error` returns a descriptive string of the error in English. For file errors, the function `file:format_error(Posix)` should be called.

`crypto_key_fun(CryptoKeyFun) -> ok | {error, Reason}`

Types:

- CryptoKeyFun = fun() – see below
- Reason = badfun | exists | term()

The `crypto_key_fun/1` function registers a unary fun that will be called if `beam_lib` needs to read an `abstract_code` chunk that has been encrypted. The fun is held in a process that is started by the function.

If there already is a fun registered when attempting to register a fun, `{error, exists}` is returned.

The fun must handle the following arguments:

```
CryptoKeyFun(init) -> ok | {ok, NewCryptoKeyFun} | {error, Term}
```

Called when the fun is registered, in the process that holds the fun. Here the crypto key fun can do any necessary initializations. If {ok, NewCryptoKeyFun} is returned then NewCryptoKeyFun will be registered instead of CryptoKeyFun. If {error, Term} is returned, the registration is aborted and crypto_key_fun/1 returns {error, Term} as well.

```
CryptoKeyFun({debug_info, Mode, Module, Filename}) -> Key
```

Called when the key is needed for the module Module in the file named Filename. Mode is the type of crypto algorithm; currently, the only possible value thus is des3_cbc. The call should fail (raise an exception) if there is no key available.

```
CryptoKeyFun(clear) -> term()
```

Called before the fun is unregistered. Here any cleaning up can be done. The return value is not important, but is passed back to the caller of clear_crypto_key_fun/0 as part of its return value.

clear_crypto_key_fun() -> {ok, Result}

Types:

- Result = undefined | term()

Unregisters the crypto key fun and terminates the process holding it, started by crypto_key_fun/1.

The clear_crypto_key_fun/1 either returns {ok, undefined} if there was no crypto key fun registered, or {ok, Term}, where Term is the return value from CryptoKeyFun(clear), see crypto_key_fun/1.

C

Erlang Module

The c module enables users to enter the short form of some commonly used commands.

> **Note:**
> These functions are are intended for interactive use in the Erlang shell only. The module prefix may be omitted.

Exports

`bt(Pid) -> void()`

Types:

- Pid = pid()

Stack backtrace for a process. Equivalent to `erlang:process_display(Pid, backtrace)`.

`c(File) -> {ok, Module} | error`
`c(File, Options) -> {ok, Module} | error`

Types:

- File = name() – see filename(3)
- Options = [Opt] – see compile:file/2

`c/1,2` compiles and then purges and loads the code for a file. `Options` defaults to []. Compilation is equivalent to:

```
compile:file(File, Options ++ [report_errors, report_warnings])
```

Note that purging the code means that any processes lingering in old code for the module are killed without warning. See `code/3` for more information.

`cd(Dir) -> void()`

Types:

- Dir = name() – see filename(3)

Changes working directory to `Dir`, which may be a relative name, and then prints the name of the new working directory.

```
2> cd("../erlang").
/home/ron/erlang
```

`flush() -> void()`

Flushes any messages sent to the shell.

`help() -> void()`

Displays help information: all valid shell internal commands, and commands in this module.

`i() -> void()`
`ni() -> void()`

`i/0` displays information about the system, listing information about all processes. `ni/0` does the same, but for all nodes the network.

`i(X, Y, Z) -> void()`

Types:

- X = Y = Z = int()

Displays information about a process, Equivalent to `process_info(pid(X, Y, Z))`, but location transparent.

`l(Module) -> void()`

Types:

- Module = atom()

Purges and loads, or reloads, a module by calling `code:purge(Module)` followed by `code:load_file(Module)`.

Note that purging the code means that any processes lingering in old code for the module are killed without warning. See `code/3` for more information.

`lc(Files) -> ok`

Types:

- Files = [File]
- File = name() – see filename(3)

Compiles a list of files by calling `compile:file(File, [report_errors, report_warnings])` for each `File` in `Files`.

`ls() -> void()`

Lists files in the current directory.

`ls(Dir) -> void()`

Types:

- Dir = name() – see filename(3)

Lists files in directory `Dir`.

`m() -> void()`

Displays information about the loaded modules, including the files from which they have been loaded.

```
m(Module) -> void()
```

Types:

- Module = atom()

Displays information about `Module`.

```
memory() -> [{Type, Size}]
```

Types:

- Type, Size – see erlang:memory/0

Memory allocation information. Equivalent to `erlang:memory/0`.

```
memory(Type) -> Size
memory([Type]) -> [{Type, Size}]
```

Types:

- Type, Size – see erlang:memory/0

Memory allocation information. Equivalent to `erlang:memory/1`.

```
nc(File) -> {ok, Module} | error
nc(File, Options) -> {ok, Module} | error
```

Types:

- File = name() – see filename(3)
- Options = [Opt] – see compile:file/2

Compiles and then loads the code for a file on all nodes. `Options` defaults to []. Compilation is equivalent to:

```
compile:file(File, Opts ++ [report_errors, report_warnings])
```

```
nl(Module) -> void()
```

Types:

- Module = atom()

Loads `Module` on all nodes.

```
pid(X, Y, Z) -> pid()
```

Types:

- X = Y = Z = int()

Converts `X`, `Y`, `Z` to the pid <`X.Y.Z`>. This function should only be used when debugging.

```
pwd() -> void()
```

Prints the name of the working directory.

```
q() -> void()
```

This function is shorthand for `init:stop()`, that is, it causes the node to stop in a controlled fashion.

```
regs() -> void()
nregs() -> void()
```

`regs/0` displays information about all registered processes. `nregs/0` does the same, but for all nodes in the network.

```
xm(ModSpec) -> void()
```

Types:

- ModSpec = Module | Filename
- Module = atom()
- Filename = string()

This function finds undefined functions, unused functions, and calls to deprecated functions in a module by calling `xref:m/1`.

```
y(File) -> YeccRet
```

Types:

- File = name() – see filename(3)
- YeccRet = – see yecc:file/2

Generates an LALR-1 parser. Equivalent to:

```
yecc:file(File)
```

```
y(File, Options) -> YeccRet
```

Types:

- File = name() – see filename(3)
- Options, YeccRet = – see yecc:file/2

Generates an LALR-1 parser. Equivalent to:

```
yecc:file(File, Options)
```

See Also

[compile(3)], filename(3) [page 171], [erlang(3)], [yecc(3)], [xref(3)]

calendar

Erlang Module

This module provides computation of local and universal time, day-of-the-week, and several time conversion functions.

Time is local when it is adjusted in accordance with the current time zone and daylight saving. Time is universal when it reflects the time at longitude zero, without any adjustment for daylight saving. Universal Coordinated Time (UTC) time is also called Greenwich Mean Time (GMT).

The time functions `local_time/0` and `universal_time/0` provided in this module both return date and time. The reason for this is that separate functions for date and time may result in a date/time combination which is displaced by 24 hours. This happens if one of the functions is called before midnight, and the other after midnight. This problem also applies to the Erlang BIFs `date/0` and `time/0`, and their use is strongly discouraged if a reliable date/time stamp is required.

All dates conform to the Gregorian calendar. This calendar was introduced by Pope Gregory XIII in 1582 and was used in all Catholic countries from this year. Protestant parts of Germany and the Netherlands adopted it in 1698, England followed in 1752, and Russia in 1918 (the October revolution of 1917 took place in November according to the Gregorian calendar).

The Gregorian calendar in this module is extended back to year 0. For a given date, the *gregorian days* is the number of days up to and including the date specified. Similarly, the *gregorian seconds* for a given date and time, is the the number of seconds up to and including the specified date and time.

For computing differences between epochs in time, use the functions counting gregorian days or seconds. If epochs are given as local time, they must be converted to universal time, in order to get the correct value of the elapsed time between epochs. Use of the function `time_difference/2` is discouraged.

DATA TYPES

```
date() = {Year, Month, Day}
  Year = int()
  Month = 1..12
  Day = 1..31
Year cannot be abbreviated. Example: 93 denotes year 93, not 1993.
Valid range depends on the underlying OS.
The date tuple must denote a valid date.

time() = {Hour, Minute, Second}
  Hour = 0..23
  Minute = Second = 0..59
```

Exports

`date_to_gregorian_days(Date) -> Days`

`date_to_gregorian_days(Year, Month, Day) -> Days`

Types:

- Date = date()
- Days = int()

This function computes the number of gregorian days starting with year 0 and ending at the given date.

`datetime_to_gregorian_seconds({Date, Time}) -> Seconds`

Types:

- Date = date()
- Time = time()
- Seconds = int()

This function computes the number of gregorian seconds starting with year 0 and ending at the given date and time.

`day_of_the_week(Date) -> DayNumber`

`day_of_the_week(Year, Month, Day) -> DayNumber`

Types:

- Date = date()
- DayNumber = 1..7

This function computes the day of the week given `Year`, `Month` and `Day`. The return value denotes the day of the week as 1: Monday, 2: Tuesday, and so on.

`gregorian_days_to_date(Days) -> Date`

Types:

- Days = int()
- Date = date()

This function computes the date given the number of gregorian days.

`gregorian_seconds_to_datetime(Seconds) -> {Date, Time}`

Types:

- Seconds = int()
- Date = date()
- Time = time()

This function computes the date and time from the given number of gregorian seconds.

`is_leap_year(Year) -> bool()`

This function checks if a year is a leap year.

`last_day_of_the_month(Year, Month) -> int()`

This function computes the number of days in a month.

`local_time() -> {Date, Time}`

Types:

- Date = date()
- Time = time()

This function returns the local time reported by the underlying operating system.

`local_time_to_universal_time({Date1, Time1}) -> {Date2, Time2}`

This function converts from local time to Universal Coordinated Time (UTC). `Date1` must refer to a local date after Jan 1, 1970.

> **Warning:**
> This function is deprecated. Use `local_time_to_universal_time_dst/1` instead, as it gives a more correct and complete result. Especially for the period that does not exist since it gets skipped during the switch *to* daylight saving time, this function still returns a result.

`local_time_to_universal_time_dst({Date1, Time1}) -> [{Date, Time}]`

Types:

- Date1 = Date = date()
- Time1 = Time = time()

This function converts from local time to Universal Coordinated Time (UTC). `Date1` must refer to a local date after Jan 1, 1970.

The return value is a list of 0, 1 or 2 possible UTC times:

`[]` For a local `{Date1, Time1}` during the period that is skipped when switching *to* daylight saving time, there is no corresponding UTC since the local time is illegal - it has never happened.

`[DstDateTimeUTC, DateTimeUTC]` For a local `{Date1, Time1}` during the period that is repeated when switching *from* daylight saving time, there are two corresponding UTCs. One for the first instance of the period when daylight saving time is still active, and one for the second instance.

`[DateTimeUTC]` For all other local times there is only one corresponding UTC.

`now_to_local_time(Now) -> {Date, Time}`

Types:

- Now – see erlang:now/0
- Date = date()
- Time = time()

This function returns local date and time converted from the return value from `erlang:now()`.

`now_to_universal_time(Now) -> {Date, Time}`
`now_to_datetime(Now) -> {Date, Time}`

Types:

- Now – see erlang:now/0
- Date = date()
- Time = time()

This function returns Universal Coordinated Time (UTC) converted from the return value from `erlang:now()`.

`seconds_to_daystime(Seconds) -> {Days, Time}`

Types:

- Seconds = Days = int()
- Time = time()

This function transforms a given number of seconds into days, hours, minutes, and seconds. The `Time` part is always non-negative, but `Days` is negative if the argument `Seconds` is.

`seconds_to_time(Seconds) -> Time`

Types:

- Seconds = int() < 86400
- Time = time()

This function computes the time from the given number of seconds. `Seconds` must be less than the number of seconds per day (86400).

`time_difference(T1, T2) -> {Days, Time}`

This function returns the difference between two `{Date, Time}` tuples. `T2` should refer to an epoch later than `T1`.

> **Warning:**
> This function is obsolete. Use the conversion functions for gregorian days and seconds instead.

`time_to_seconds(Time) -> Seconds`

Types:

- Time = time()
- Seconds = int()

This function computes the number of seconds since midnight up to the specified time.

`universal_time() -> {Date, Time}`

Types:

- Date = date()
- Time = time()

This function returns the Universal Coordinated Time (UTC) reported by the underlying operating system. Local time is returned if universal time is not available.

`universal_time_to_local_time({Date1, Time1}) -> {Date2, Time2}`

Types:

- Date1 = Date2 = date()
- Time1 = Time2 = time()

This function converts from Universal Coordinated Time (UTC) to local time. `Date1` must refer to a date after Jan 1, 1970.

`valid_date(Date) -> bool()`
`valid_date(Year, Month, Day) -> bool()`

Types:

- Date = date()

This function checks if a date is a valid.

Leap Years

The notion that every fourth year is a leap year is not completely true. By the Gregorian rule, a year Y is a leap year if either of the following rules is valid:

- Y is divisible by 4, but not by 100; or
- Y is divisible by 400.

Accordingly, 1996 is a leap year, 1900 is not, but 2000 is.

Date and Time Source

Local time is obtained from the Erlang BIF `localtime/0`. Universal time is computed from the BIF `universaltime/0`.

The following facts apply:

- there are 86400 seconds in a day
- there are 365 days in an ordinary year
- there are 366 days in a leap year
- there are 1461 days in a 4 year period
- there are 36524 days in a 100 year period
- there are 146097 days in a 400 year period
- there are 719528 days between Jan 1, 0 and Jan 1, 1970.

dets

Erlang Module

The module `dets` provides a term storage on file. The stored terms, in this module called *objects*, are tuples such that one element is defined to be the key. A Dets *table* is a collection of objects with the key at the same position stored on a file.

Dets is used by the Mnesia application, and is provided as is for users who are interested in an efficient storage of Erlang terms on disk only. Many applications just need to store some terms in a file. Mnesia adds transactions, queries, and distribution. The size of Dets files cannot exceed 2 GB. If larger tables are needed, Mnesia's table fragmentation can be used.

There are three types of Dets tables: set, bag and duplicate_bag. A table of type *set* has at most one object with a given key. If an object with a key already present in the table is inserted, the existing object is overwritten by the new object. A table of type *bag* has zero or more different objects with a given key. A table of type *duplicate_bag* has zero or more possibly matching objects with a given key.

Dets tables must be opened before they can be updated or read, and when finished they must be properly closed. If a table has not been properly closed, Dets will automatically repair the table. This can take a substantial time if the table is large. A Dets table is closed when the process which opened the table terminates. If several Erlang processes (users) open the same Dets table, they will share the table. The table is properly closed when all users have either terminated or closed the table. Dets tables are not properly closed if the Erlang runtime system is terminated abnormally.

> **Note:**
> A `^C` command abnormally terminates an Erlang runtime system in a Unix environment with a break-handler.

Since all operations performed by Dets are disk operations, it is important to realize that a single look-up operation involves a series of disk seek and read operations. For this reason, the Dets functions are much slower than the corresponding Ets functions, although Dets exports a similar interface.

Dets organizes data as a linear hash list and the hash list grows gracefully as more data is inserted into the table. Space management on the file is performed by what is called a buddy system. The current implementation keeps the entire buddy system in RAM, which implies that if the table gets heavily fragmented, quite some memory can be used up. The only way to defragment a table is to close it and then open it again with the `repair` option set to `force`.

It is worth noting that the ordered_set type present in Ets is not yet implemented by Dets, neither is the limited support for concurrent updates which makes a sequence of `first` and `next` calls safe to use on fixed Ets tables. Both these features will be implemented by Dets in a future release of Erlang/OTP. Until then, the Mnesia

application (or some user implemented method for locking) has to be used to implement safe concurrency. Currently, no library of Erlang/OTP has support for ordered disk based term storage.

Two versions of the format used for storing objects on file are supported by Dets. The first version, 8, is the format always used for tables created by OTP R7 and earlier. The second version, 9, is the default version of tables created by OTP R8 (and later OTP releases). OTP R8 can create version 8 tables, and convert version 8 tables to version 9, and vice versa, upon request.

All Dets functions return `{error, Reason}` if an error occurs (`first/1` and `next/2` are exceptions, they exit the process with the error tuple). If given badly formed arguments, all functions exit the process with a `badarg` message.

Types

```
access() = read | read_write
auto_save() = infinity | int()
bindings_cont() = tuple()
bool() = true | false
file() = string()
int() = integer() >= 0
keypos() = integer() >= 1
name() = atom() | ref()
no_slots() = integer() >= 0 | default
object() = tuple()
object_cont() = tuple()
select_cont() = tuple()
type() = bag | duplicate_bag | set
version() = 8 | 9 | default
```

Exports

`all() -> [Name]`

Types:

- Name = name()

Returns a list of the names of all open tables on this node.

`bchunk(Name, Continuation) -> {Continuation2, Data} | '$end_of_table' | {error, Reason}`

Types:

- Name = name()
- Continuation = start | cont()
- Continuation2 = cont()
- Data = binary() | tuple()

Returns a list of objects stored in a table. The exact representation of the returned objects is not public. The lists of data can be used for initializing a table by giving the value `bchunk` to the `format` option of the `init_table/3` function. The Mnesia application uses this function for copying open tables.

Unless the table is protected using `safe_fixtable/2`, calls to `bchunk/2` may not work as expected if concurrent updates are made to the table.

The first time `bchunk/2` is called, an initial continuation, the atom `start`, must be provided.

The `bchunk/2` function returns a tuple {`Continuation2, Data`}, where `Data` is a list of objects. `Continuation2` is another continuation which is to be passed on to a subsequent call to `bchunk/2`. With a series of calls to `bchunk/2` it is possible to extract all objects of the table.

`bchunk/2` returns `'$end_of_table'` when all objects have been returned, or {`error, Reason`} if an error occurs.

`close(Name) -> ok | {error, Reason}`

Types:

- Name = name()

Closes a table. Only processes that have opened a table are allowed to close it.

All open tables must be closed before the system is stopped. If an attempt is made to open a table which has not been properly closed, Dets automatically tries to repair the table.

`delete(Name, Key) -> ok | {error, Reason}`

Types:

- Name = name()

Deletes all objects with the key `Key` from the table `Name`.

`delete_all_objects(Name) -> ok | {error, Reason}`

Types:

- Name = name()

Deletes all objects from a table in almost constant time. However, if the table if fixed, `delete_all_objects(T)` is equivalent to `match_delete(T, '_')`.

`delete_object(Name, Object) -> ok | {error, Reason}`

Types:

- Name = name()
- Object = object()

Deletes all instances of a given object from a table. If a table is of type `bag` or `duplicate_bag`, the `delete/2` function cannot be used to delete only some of the objects with a given key. This function makes this possible.

`first(Name) -> Key | '$end_of_table'`

Types:

- Key = term()
- Name = name()

Returns the first key stored in the table `Name` according to the table's internal order, or `'$end_of_table'` if the table is empty.

Unless the table is protected using `safe_fixtable/2`, subsequent calls to `next/2` may not work as expected if concurrent updates are made to the table.

Should an error occur, the process is exited with an error tuple `{error, Reason}`. The reason for not returning the error tuple is that it cannot be distinguished from a key.

There are two reasons why `first/1` and `next/2` should not be used: they are not very efficient, and they prevent the use of the key `'$end_of_table'` since this atom is used to indicate the end of the table. If possible, the `match`, `match_object`, and `select` functions should be used for traversing tables.

`foldl(Function, Acc0, Name) -> Acc1 | {error, Reason}`

Types:

- Function = fun(Object, AccIn) -> AccOut
- Acc0 = Acc1 = AccIn = AccOut = term()
- Name = name()
- Object = object()

Calls `Function` on successive elements of the table `Name` together with an extra argument `AccIn`. The order in which the elements of the table are traversed is unspecified. `Function` must return a new accumulator which is passed to the next call. `Acc0` is returned if the table is empty.

`foldr(Function, Acc0, Name) -> Acc1 | {error, Reason}`

Types:

- Function = fun(Object, AccIn) -> AccOut
- Acc0 = Acc1 = AccIn = AccOut = term()
- Name = name()
- Object = object()

Calls `Function` on successive elements of the table `Name` together with an extra argument `AccIn`. The order in which the elements of the table are traversed is unspecified. `Function` must return a new accumulator which is passed to the next call. `Acc0` is returned if the table is empty.

`from_ets(Name, EtsTab) -> ok | {error, Reason}`

Types:

- Name = name()
- EtsTab = -see ets(3)-

Deletes all objects of the table `Name` and then inserts all the objects of the Ets table `EtsTab`. The order in which the objects are inserted is not specified. Since `ets:safe_fixtable/2` is called the Ets table must be public or owned by the calling process.

`info(Name) -> InfoList | undefined`

Types:

- Name = name()
- InfoList = [{Item, Value}]

Returns information about the table `Name` as a list of `{Item, Value}` tuples:

- `{file_size, int()}`, the size of the file in bytes.
- `{filename, file()}`, the name of the file where objects are stored.
- `{keypos, keypos()}`, the position of the key.
- `{size, int()}`, the number of objects stored in the table.
- `{type, type()}`, the type of the table.

```
info(Name, Item) -> Value | undefined
```

Types:

- Name = name()

Returns the information associated with `Item` for the table `Name`. In addition to the `{Item, Value}` pairs defined for `info/1`, the following items are allowed:

- `{access, access()}`, the access mode.
- `{auto_save, auto_save()}`, the auto save interval.
- `{bchunk_format, binary()}`, an opaque binary describing the format of the objects returned by `bchunk/2`. The binary can be used as argument to `is_compatible_chunk_format/2`. Only available for version 9 tables.
- `{hash, Hash}`. Describes which BIF is used to calculate the hash values of the objects stored in the Dets table. Possible values of Hash are `hash`, which implies that the `erlang:hash/2` BIF is used, `phash`, which implies that the `erlang:phash/2` BIF is used, and `phash2`, which implies that the `erlang:phash2/1` BIF is used.
- `{memory, int()}`, the size of the file in bytes. The same value is associated with the item `file_size`.
- `{no_keys, int()}`, the number of different keys stored in the table. Only available for version 9 tables.
- `{no_objects, int()}`, the number of objects stored in the table.
- `{no_slots, {Min, Used, Max}}`, the number of slots of the table. `Min` is the minimum number of slots, `Used` is the number of currently used slots, and `Max` is the maximum number of slots. Only available for version 9 tables.
- `{owner, pid()}`, the pid of the process that handles requests to the Dets table.
- `{ram_file, bool()}`, whether the table is kept in RAM.
- `{safe_fixed, SafeFixed}`. If the table is fixed, SafeFixed is a tuple `{FixedAtTime, [{Pid,RefCount}]}`. `FixedAtTime` is the time when the table was first fixed, and `Pid` is the pid of the process that fixes the table `RefCount` times. There may be any number of processes in the list. If the table is not fixed, SafeFixed is the atom `false`.
- `{version, int()}`, the version of the format of the table.

```
init_table(Name, InitFun [, Options]) -> ok | {error, Reason}
```

Types:

- Name = atom()
- InitFun = fun(Arg) -> Res
- Arg = read | close
- Res = end_of_input | {[object()], InitFun} | {Data, InitFun} | term()
- Data = binary() | tuple()

Replaces the existing objects of the table `Name` with objects created by calling the input function `InitFun`, see below. The reason for using this function rather than calling `insert/2` is that of efficiency. It should be noted that the input functions are called by the process that handles requests to the Dets table, not by the calling process.

When called with the argument `read` the function `InitFun` is assumed to return `end_of_input` when there is no more input, or `{Objects, Fun}`, where `Objects` is a list of objects and `Fun` is a new input function. Any other value Value is returned as an error `{error, {init_fun, Value}}`. Each input function will be called exactly once, and should an error occur, the last function is called with the argument `close`, the reply of which is ignored.

If the type of the table is `set` and there is more than one object with a given key, one of the objects is chosen. This is not necessarily the last object with the given key in the sequence of objects returned by the input functions. Extra objects should be avoided, or the file will be unnecessarily fragmented. This holds also for duplicated objects stored in tables of type `duplicate_bag`.

It is important that the table has a sufficient number of slots for the objects. If not, the hash list will start to grow when `init_table/2` returns which will significantly slow down access to the table for a period of time. The minimum number of slots is set by the `open_file/2` option `min_no_slots` and returned by the `info/2` item `no_slots`. See also the `min_no_slots` option below.

The `Options` argument is a list of `{Key, Val}` tuples where the following values are allowed:

- `{min_no_slots, no_slots()}`. Specifies the estimated number of different keys that will be stored in the table. The `open_file` option with the same name is ignored unless the table is created, and in that case performance can be enhanced by supplying an estimate when initializing the table.
- `{format, Format}`. Specifies the format of the objects returned by the function `InitFun`. If `Format` is `term` (the default), `InitFun` is assumed to return a list of tuples. If `Format` is `bchunk`, `InitFun` is assumed to return `Data` as returned by `bchunk/2`. This option overrides the `min_no_slots` option.

`insert(Name, Objects) -> ok | {error, Reason}`

Types:

- Name = name()
- Objects = object() | [object()]

Inserts one or more objects into the table `Name`. If there already exists an object with a key matching the key of some of the given objects and the table type is `set`, the old object will be replaced.

`insert_new(Name, Objects) -> Bool`

Types:

- Name = name()
- Objects = object() | [object()]
- Bool = bool()

Inserts one or more objects into the table `Name`. If there already exists some object with a key matching the key of any of the given objects the table is not updated and `false` is returned, otherwise the objects are inserted and `true` returned.

`is_compatible_bchunk_format(Name, BchunkFormat) -> Bool`

Types:

- Name = name()
- BchunkFormat = binary()
- Bool = bool()

Returns `true` if it would be possible to initialize the table `Name`, using `init_table/3` with the option {`format,bchunk`}, with objects read with `bchunk/2` from some table `T` such that calling `info(T,bchunk_format)` returns `BchunkFormat`.

`is_dets_file(FileName) -> Bool | {error, Reason}`

Types:

- FileName = file()
- Bool = bool()

Returns `true` if the file `FileName` is a Dets table, `false` otherwise.

`lookup(Name, Key) -> [Object] | {error, Reason}`

Types:

- Key = term()
- Name = name()
- Object = object()

Returns a list of all objects with the key `Key` stored in the table `Name`. For example:

```
2> dets:open_file(abc, [{type, bag}]).
{ok,abc}
3> dets:insert(abc, {1,2,3}).
ok
4> dets:insert(abc, {1,3,4}).
ok
5> dets:lookup(abc, 1).
[{1,2,3},{1,3,4}]
```

If the table is of type `set`, the function returns either the empty list or a list with one object, as there cannot be more than one object with a given key. If the table is of type `bag` or `duplicate_bag`, the function returns a list of arbitrary length.

Note that the order of objects returned is unspecified. In particular, the order in which objects were inserted is not reflected.

`match(Continuation) -> {[Match], Continuation2} | '$end_of_table' | {error, Reason}`

Types:

- Continuation = Continuation2 = bindings_cont()
- Match = [term()]

Matches some objects stored in a table and returns a non-empty list of the bindings that match a given pattern in some unspecified order. The table, the pattern, and the number of objects that are matched are all defined by `Continuation`, which has been returned by a prior call to `match/1` or `match/3`.

When all objects of the table have been matched, `'$end_of_table'` is returned.

`match(Name, Pattern) -> [Match] | {error, Reason}`

Types:

- Name = name()
- Pattern = tuple()
- Match = [term()]

Returns for each object of the table `Name` that matches `Pattern` a list of bindings in some unspecified order. See ets(3) [page 139] for a description of patterns. If the keypos'th element of `Pattern` is unbound, all objects of the table are matched. If the keypos'th element is bound, only the objects with the right key are matched.

`match(Name, Pattern, N) -> {[Match], Continuation} | '$end_of_table' | {error, Reason}`

Types:

- Name = name()
- Pattern = tuple()
- N = default | int()
- Match = [term()]
- Continuation = bindings_cont()

Matches some or all objects of the table `Name` and returns a non-empty list of the bindings that match `Pattern` in some unspecified order. See ets(3) [page 139] for a description of patterns.

A tuple of the bindings and a continuation is returned, unless the table is empty, in which case `'$end_of_table'` is returned. The continuation is to be used when matching further objects by calling `match/1`.

If the keypos'th element of `Pattern` is bound, all objects of the table are matched. If the keypos'th element is unbound, all objects of the table are matched, `N` objects at a time, until at least one object matches or the end of the table has been reached. The default, indicated by giving `N` the value `default`, is to let the number of objects vary depending on the sizes of the objects. If `Name` is a version 9 table, all objects with the same key are always matched at the same time which implies that more than N objects may sometimes be matched.

The table should always be protected using `safe_fixtable/2` before calling `match/3`, or errors may occur when calling `match/1`.

`match_delete(Name, Pattern) -> N | {error, Reason}`

Types:

- Name = name()
- N = int()

- Pattern = tuple()

Deletes all objects that match `Pattern` from the table `Name`, and returns the number of deleted objects. See ets(3) [page 139] for a description of patterns.

If the keypos'th element of `Pattern` is bound, only the objects with the right key are matched.

`match_object(Continuation) -> {[Object], Continuation2} | '$end_of_table' | {error, Reason}`

Types:

- Continuation = Continuation2 = object_cont()
- Object = object()

Returns a non-empty list of some objects stored in a table that match a given pattern in some unspecified order. The table, the pattern, and the number of objects that are matched are all defined by `Continuation`, which has been returned by a prior call to `match_object/1` or `match_object/3`.

When all objects of the table have been matched, `'$end_of_table'` is returned.

`match_object(Name, Pattern) -> [Object] | {error, Reason}`

Types:

- Name = name()
- Pattern = tuple()
- Object = object()

Returns a list of all objects of the table `Name` that match `Pattern` in some unspecified order. See ets(3) [page 139] for a description of patterns.

If the keypos'th element of `Pattern` is unbound, all objects of the table are matched. If the keypos'th element of `Pattern` is bound, only the objects with the right key are matched.

Using the `match_object` functions for traversing all objects of a table is more efficient than calling `first/1` and `next/2` or `slot/2`.

`match_object(Name, Pattern, N) -> {[Object], Continuation} | '$end_of_table' | {error, Reason}`

Types:

- Name = name()
- Pattern = tuple()
- N = default | int()
- Object = object()
- Continuation = object_cont()

Matches some or all objects stored in the table `Name` and returns a non-empty list of the objects that match `Pattern` in some unspecified order. See ets(3) [page 139] for a description of patterns.

A list of objects and a continuation is returned, unless the table is empty, in which case `'$end_of_table'` is returned. The continuation is to be used when matching further objects by calling `match_object/1`.

If the keypos'th element of `Pattern` is bound, all objects of the table are matched. If the keypos'th element is unbound, all objects of the table are matched, `N` objects at a time, until at least one object matches or the end of the table has been reached. The default, indicated by giving `N` the value `default`, is to let the number of objects vary depending on the sizes of the objects. If `Name` is a version 9 table, all matching objects with the same key are always returned in the same reply which implies that more than N objects may sometimes be returned.

The table should always be protected using `safe_fixtable/2` before calling `match_object/3`, or errors may occur when calling `match_object/1`.

`member(Name, Key) -> Bool | {error, Reason}`

Types:

- Name = name()
- Key = term()
- Bool = bool()

Works like `lookup/2`, but does not return the objects. The function returns `true` if one or more elements of the table has the key `Key`, `false` otherwise.

`next(Name, Key1) -> Key2 | '$end_of_table'`

Types:

- Name = name()
- Key1 = Key2 = term()

Returns the key following `Key1` in the table `Name` according to the table's internal order, or `'$end_of_table'` if there is no next key.

Should an error occur, the process is exited with an error tuple `{error, Reason}`.

Use `first/1` to find the first key in the table.

`open_file(Filename) -> {ok, Reference} | {error, Reason}`

Types:

- FileName = file()
- Reference = ref()

Opens an existing table. If the table has not been properly closed, the error `{error, need_repair}` is returned. The returned reference is to be used as the name of the table. This function is most useful for debugging purposes.

`open_file(Name, Args) -> {ok, Name} | {error, Reason}`

Types:

- Name = atom()

Opens a table. An empty Dets table is created if no file exists.

The atom `Name` is the name of the table. The table name must be provided in all subsequent operations on the table. The name can be used by other processes as well, and several process can share one table.

If two processes open the same table by giving the same name and arguments, then the table will have two users. If one user closes the table, it still remains open until the second user closes the table.

The `Args` argument is a list of `{Key, Val}` tuples where the following values are allowed:

- `{access, access()}`. It is possible to open existing tables in read-only mode. A table which is opened in read-only mode is not subjected to the automatic file reparation algorithm if it is later opened after a crash. The default value is `read_write`.
- `{auto_save, auto_save()}`, the auto save interval. If the interval is an integer `Time`, the table is flushed to disk whenever it is not accessed for `Time` milliseconds. A table that has been flushed will require no reparation when reopened after an uncontrolled emulator halt. If the interval is the atom `infinity`, auto save is disabled. The default value is 180000 (3 minutes).
- `{estimated_no_objects, int()}`. Equivalent to the `min_no_slots` option.
- `{file, file()}`, the name of the file to be opened. The default value is the name of the table.
- `{max_no_slots, no_slots()}`, the maximum number of slots that will be used. The default value is 2 M, and the maximal value is 32 M. Note that a higher value may increase the fragmentation of the table, and conversely, that a smaller value may decrease the fragmentation, at the expense of execution time. Only available for version 9 tables.
- `{min_no_slots, no_slots()}`. Application performance can be enhanced with this flag by specifying, when the table is created, the estimated number of different keys that will be stored in the table. The default value as well as the minimum value is 256.
- `{keypos, keypos()}`, the position of the element of each object to be used as key. The default value is 1. The ability to explicitly state the key position is most convenient when we want to store Erlang records in which the first position of the record is the name of the record type.
- `{ram_file, bool()}`, whether the table is to be kept in RAM. Keeping the table in RAM may sound like an anomaly, but can enhance the performance of applications which open a table, insert a set of objects, and then close the table. When the table is closed, its contents are written to the disk file. The default value is `false`.
- `{repair, Value}`. `Value` can be either a `bool()` or the atom `force`. The flag specifies whether the Dets server should invoke the automatic file reparation algorithm. The default is `true`. If `false` is specified, there is no attempt to repair the file and `{error, need_repair}` is returned if the table needs to be repaired.

  The value `force` means that a reparation will take place even if the table has been properly closed. This is how to convert tables created by older versions of STDLIB. An example is tables hashed with the deprecated `erlang:hash/2` BIF. Tables created with Dets from a STDLIB version of 1.8.2 and later use the `erlang:phash/2` function or the `erlang:phash2/1` function, which is preferred.

  The `repair` option is ignored if the table is already open.

- {type, type()}, the type of the table. The default value is set.
- {version, version()}, the version of the format used for the table. The default value is 9. Tables on the format used before OTP R8 can be created by giving the value 8. A version 8 table can be converted to a version 9 table by giving the options {version,9} and {repair,force}.

pid2name(Pid) -> {ok, Name} | undefined

Types:

- Name = name()
- Pid = pid()

Returns the name of the table given the pid of a process that handles requests to a table, or undefined if there is no such table.

This function is meant to be used for debugging only.

repair_continuation(Continuation, MatchSpec) -> Continuation2

Types:

- Continuation = Continuation2 = select_cont()
- MatchSpec = match_spec()

This function can be used to restore an opaque continuation returned by select/3 or select/1 if the continuation has passed through external term format (been sent between nodes or stored on disk).

The reason for this function is that continuation terms contain compiled match specifications and therefore will be invalidated if converted to external term format. Given that the original match specification is kept intact, the continuation can be restored, meaning it can once again be used in subsequent select/1 calls even though it has been stored on disk or on another node.

See also ets(3) for further explanations and examples.

> **Note:**
> This function is very rarely needed in application code. It is used by Mnesia to implement distributed select/3 and select/1 sequences. A normal application would either use Mnesia or keep the continuation from being converted to external format.
>
> The reason for not having an external representation of compiled match specifications is performance. It may be subject to change in future releases, while this interface will remain for backward compatibility.

safe_fixtable(Name, Fix)

Types:

- Name = name()
- Fix = bool()

If Fix is true, the table Name is fixed (once more) by the calling process, otherwise the table is released. The table is also released when a fixing process terminates.

If several processes fix a table, the table will remain fixed until all processes have released it or terminated. A reference counter is kept on a per process basis, and N consecutive fixes require N releases to release the table.

It is not guaranteed that calls to first/1, next/2, select and match functions work as expected even if the table has been fixed; the limited support for concurrency implemented in Ets has not yet been implementeded in Dets. Fixing a table currently only disables resizing of the hash list of the table.

If objects have been added while the table was fixed, the hash list will start to grow when the table is released which will significantly slow down access to the table for a period of time.

`select(Continuation) -> {Selection, Continuation2} | '$end_of_table' | {error, Reason}`

Types:

- Continuation = Continuation2 = select_cont()
- Selection = [term()]

Applies a match specification to some objects stored in a table and returns a non-empty list of the results. The table, the match specification, and the number of objects that are matched are all defined by Continuation, which has been returned by a prior call to select/1 or select/3.

When all objects of the table have been matched, '$end_of_table' is returned.

`select(Name, MatchSpec) -> Selection | {error, Reason}`

Types:

- Name = name()
- MatchSpec = match_spec()
- Selection = [term()]

Returns the results of applying the match specification MatchSpec to all or some objects stored in the table Name. The order of the objects is not specified. See the ERTS User's Guide for a description of match specifications.

If the keypos'th element of MatchSpec is unbound, the match specification is applied to all objects of the table. If the keypos'th element is bound, the match specification is applied to the objects with the right key(s) only.

Using the select functions for traversing all objects of a table is more efficient than calling first/1 and next/2 or slot/2.

`select(Name, MatchSpec, N) -> {Selection, Continuation} | '$end_of_table' | {error, Reason}`

Types:

- Name = name()
- MatchSpec = match_spec()
- N = default | int()
- Selection = [term()]
- Continuation = select_cont()

Returns the results of applying the match specification MatchSpec to some or all objects stored in the table Name. The order of the objects is not specified. See the ERTS User's Guide for a description of match specifications.

A tuple of the results of applying the match specification and a continuation is returned, unless the table is empty, in which case '$end_of_table' is returned. The continuation is to be used when matching further objects by calling select/1.

If the keypos'th element of MatchSpec is bound, the match specification is applied to all objects of the table with the right key(s). If the keypos'th element of MatchSpec is unbound, the match specification is applied to all objects of the table, N objects at a time, until at least one object matches or the end of the table has been reached. The default, indicated by giving N the value default, is to let the number of objects vary depending on the sizes of the objects. If Name is a version 9 table, all objects with the same key are always handled at the same time which implies that the match specification may be applied to more than N objects.

The table should always be protected using safe_fixtable/2 before calling select/3, or errors may occur when calling select/1.

```
select_delete(Name, MatchSpec) -> N | {error, Reason}
```

Types:

- Name = name()
- MatchSpec = match_spec()
- N = int()

Deletes each object from the table Name such that applying the match specification MatchSpec to the object returns the value true. See the ERTS User's Guide for a description of match specifications. Returns the number of deleted objects.

If the keypos'th element of MatchSpec is bound, the match specification is applied to the objects with the right key(s) only.

```
slot(Name, I) -> '$end_of_table' | [Object] | {error, Reason}
```

Types:

- Name = name()
- I = int()
- Object = object()

The objects of a table are distributed among slots, starting with slot 0 and ending with slot n. This function returns the list of objects associated with slot I. If I is greater than n '$end_of_table' is returned.

```
sync(Name) -> ok | {error, Reason}
```

Types:

- Name = name()

Ensures that all updates made to the table Name are written to disk. This also applies to tables which have been opened with the ram_file flag set to true. In this case, the contents of the RAM file are flushed to disk.

Note that the space management data structures kept in RAM, the buddy system, is also written to the disk. This may take some time if the table is fragmented.

`table(Name [, Options]) -> QueryHandle`

Types:

- Name = name()
- QueryHandle = -a query handle, see qlc(3)-
- Options = [Option] | Option
- Option = {n_objects, Limit} | {traverse, TraverseMethod}
- Limit = default | integer() >= 1
- TraverseMethod = first_next | select | {select, MatchSpec}
- MatchSpec = match_spec()

Returns a QLC (Query List Comprehension) query handle. The module `qlc` implements a query language aimed mainly at Mnesia but Ets tables, Dets tables, and lists are also recognized by QLC as sources of data. Calling `dets:table/1,2` is the means to make the Dets table `Name` usable to QLC.

When there are only simple restrictions on the key position QLC uses `dets:lookup/2` to look up the keys, but when that is not possible the whole table is traversed. The option `traverse` determines how this is done:

- `first_next`. The table is traversed one key at a time by calling `dets:first/1` and `dets:next/2`.
- `select`. The table is traversed by calling `dets:select/3` and `dets:select/1`. The option `n_objects` determines the number of objects returned (the third argument of `select/3`). The match specification (the second argument of `select/3`) is assembled by QLC: simple filters are translated into equivalent match specifications while more complicated filters have to be applied to all objects returned by `select/3` given a match specification that matches all objects.
- {`select, MatchSpec`}. As for `select` the table is traversed by calling `dets:select/3` and `dets:select/1`. The difference is that the match specification is explicitly given. This is how to state match specifications that cannot easily be expressed within the syntax provided by QLC.

The following example uses an explicit match specification to traverse the table:

```
1> dets:open_file(t, []),
dets:insert(t, [{1,a},{2,b},{3,c},{4,d}]),
MS = ets:fun2ms(fun({X,Y}) when (X > 1) or (X < 5) -> {Y} end),
QH1 = dets:table(t, [{traverse, {select, MS}}]).
```

An example with implicit match specification:

```
2> QH2 = qlc:q([{Y} || {X,Y} <- dets:table(t), (X > 1) or (X < 5)]).
```

The latter example is in fact equivalent to the former which can be verified using the function `qlc:info/1`:

```
3> qlc:info(QH1) =:= qlc:info(QH2).
true
```

`qlc:info/1` returns information about a query handle, and in this case identical information is returned for the two query handles.

`to_ets(Name, EtsTab) -> EtsTab | {error, Reason}`

Types:

- Name = name()
- EtsTab = -see ets(3)-

Inserts the objects of the Dets table `Name` into the Ets table `EtsTab`. The order in which the objects are inserted is not specified. The existing objects of the Ets table are kept unless overwritten.

`traverse(Name, Fun) -> Return | {error, Reason}`

Types:

- Fun = fun(Object) -> FunReturn
- FunReturn = continue | {continue, Val} | {done, Value}
- Val = Value = term()
- Name = name()
- Object = object()
- Return = [term()]

Applies `Fun` to each object stored in the table `Name` in some unspecified order. Different actions are taken depending on the return value of `Fun`. The following `Fun` return values are allowed:

`continue` Continue to perform the traversal. For example, the following function can be used to print out the contents of a table:

```
fun(X) -> io:format("~p~n", [X]), continue end.
```

`{continue, Val}` Continue the traversal and accumulate `Val`. The following function is supplied in order to collect all objects of a table in a list:

```
fun(X) -> {continue, X} end.
```

`{done, Value}` Terminate the traversal and return `[Value | Acc]`.

Any other value returned by `Fun` terminates the traversal and is immediately returned.

`update_counter(Name, Key, Increment) -> Result`

Types:

- Name = name()
- Key = term()
- Increment = {Pos, Incr} | Incr
- Pos = Incr = Result = integer()

Updates the object with key `Key` stored in the table `Name` of type `set` by adding `Incr` to the element at the `Pos`:th position. The new counter value is returned. If no position is specified, the element directly following the key is updated.

This functions provides a way of updating a counter, without having to look up an object, update the object by incrementing an element and insert the resulting object into the table again.

See Also

ets(3) [page 139], mnesia(3), qlc(3) [page 287]

dict

Erlang Module

`Dict` implements a `Key` - `Value` dictionary. The representation of a dictionary is not defined.

This module provides exactly the same interface as the module `orddict`. One difference is that while this module considers two keys as different if they do not match (`=:=`), `orddict` considers two keys as different if and only if they do not compare equal (`==`).

DATA TYPES

```
dictionary()
  as returned by new/0
```

Exports

`append(Key, Value, Dict1) -> Dict2`

Types:

- Key = Value = term()
- Dict1 = Dict2 = dictionary()

This function appends a new `Value` to the current list of values associated with `Key`. An exception is generated if the initial value associated with `Key` is not a list of values.

`append_list(Key, ValList, Dict1) -> Dict2`

Types:

- ValList = [Value]
- Key = Value = term()
- Dict1 = Dict2 = dictionary()

This function appends a list of values `ValList` to the current list of values associated with `Key`. An exception is generated if the initial value associated with `Key` is not a list of values.

`erase(Key, Dict1) -> Dict2`

Types:

- Key = term()
- Dict1 = Dict2 = dictionary()

This function erases all items with a given key from a dictionary.

`fetch(Key, Dict) -> Value`

Types:

- Key = Value = term()
- Dict = dictionary()

This function returns the value associated with `Key` in the dictionary `Dict`. `fetch` assumes that the `Key` is present in the dictionary and an exception is generated if `Key` is not in the dictionary.

`fetch_keys(Dict) -> Keys`

Types:

- Dict = dictionary()
- Keys = [term()]

This function returns a list of all keys in the dictionary.

`filter(Pred, Dict1) -> Dict2`

Types:

- Pred = fun(Key, Value) -> bool()
- Key = Value = term()
- Dict1 = Dict2 = dictionary()

`Dict2` is a dictionary of all keys and values in `Dict1` for which `Pred(Key, Value)` is `true`.

`find(Key, Dict) -> {ok, Value} | error`

Types:

- Key = Value = term()
- Dict = dictionary()

This function searches for a key in a dictionary. Returns `{ok, Value}` where `Value` is the value associated with `Key`, or `error` if the key is not present in the dictionary.

`fold(Fun, Acc0, Dict) -> Acc1`

Types:

- Fun = fun(Key, Value, AccIn) -> AccOut
- Key = Value = term()
- Acc0 = Acc1 = AccIn = AccOut = term()
- Dict = dictionary()

Calls `Fun` on successive keys and values of `Dict` together with an extra argument `Acc` (short for accumulator). `Fun` must return a new accumulator which is passed to the next call. `Acc0` is returned if the list is empty. The evaluation order is undefined.

`from_list(List) -> Dict`

Types:

- List = [{Key, Value}]

- Dict = dictionary()

This function converts the key/value list `List` to a dictionary.

`is_key(Key, Dict) -> bool()`

Types:

- Key = term()
- Dict = dictionary()

This function tests if `Key` is contained in the dictionary `Dict`.

`map(Fun, Dict1) -> Dict2`

Types:

- Fun = fun(Key, Value1) -> Value2
- Key = Value1 = Value2 = term()
- Dict1 = Dict2 = dictionary()

`map` calls `Func` on successive keys and values of `Dict` to return a new value for each key. The evaluation order is undefined.

`merge(Fun, Dict1, Dict2) -> Dict3`

Types:

- Fun = fun(Key, Value1, Value2) -> Value
- Key = Value1 = Value2 = Value3 = term()
- Dict1 = Dict2 = Dict3 = dictionary()

`merge` merges two dictionaries, `Dict1` and `Dict2`, to create a new dictionary. All the `Key - Value` pairs from both dictionaries are included in the new dictionary. If a key occurs in both dictionaries then `Fun` is called with the key and both values to return a new value. `merge` could be defined as:

```
merge(Fun, D1, D2) ->
    fold(fun (K, V1, D) ->
                 update(K, fun (V2) -> Fun(K, V1, V2) end, V1, D)
         end, D2, D1).
```

but is faster.

`new() -> dictionary()`

This function creates a new dictionary.

`size(Dict) -> int()`

Types:

- Dict = dictionary()

Returns the number of elements in a `Dict`.

`store(Key, Value, Dict1) -> Dict2`

Types:

- Key = Value = term()

- Dict1 = Dict2 = dictionary()

This function stores a `Key` - `Value` pair in a dictionary. If the `Key` already exists in `Dict1`, the associated value is replaced by `Value`.

`to_list(Dict) -> List`

Types:

- Dict = dictionary()
- List = [{Key, Value}]

This function converts the dictionary to a list representation.

`update(Key, Fun, Dict1) -> Dict2`

Types:

- Key = term()
- Fun = fun(Value1) -> Value2
- Value1 = Value2 = term()
- Dict1 = Dict2 = dictionary()

Update the a value in a dictionary by calling `Fun` on the value to get a new value. An exception is generated if `Key` is not present in the dictionary.

`update(Key, Fun, Initial, Dict1) -> Dict2`

Types:

- Key = Initial = term()
- Fun = fun(Value1) -> Value2
- Value1 = Value2 = term()
- Dict1 = Dict2 = dictionary()

Update the a value in a dictionary by calling `Fun` on the value to get a new value. If `Key` is not present in the dictionary then `Initial` will be stored as the first value. For example `append/3` could be defined as:

```
append(Key, Val, D) ->
    update(Key, fun (Old) -> Old ++ [Val] end, [Val], D).
```

`update_counter(Key, Increment, Dict1) -> Dict2`

Types:

- Key = term()
- Increment = number()
- Dict1 = Dict2 = dictionary()

Add `Increment` to the value associated with `Key` and store this value. If `Key` is not present in the dictionary then `Increment` will be stored as the first value.

This could be defined as:

```
update_counter(Key, Incr, D) ->
    update(Key, fun (Old) -> Old + Incr end, Incr, D).
```

but is faster.

Notes

The functions `append` and `append_list` are included so we can store keyed values in a list *accumulator*. For example:

```
> D0 = dict:new(),
  D1 = dict:store(files, [], D0),
  D2 = dict:append(files, f1, D1),
  D3 = dict:append(files, f2, D2),
  D4 = dict:append(files, f3, D3),
  dict:fetch(files, D4).
[f1,f2,f3]
```

This saves the trouble of first fetching a keyed value, appending a new value to the list of stored values, and storing the result.

The function `fetch` should be used if the key is known to be in the dictionary, otherwise `find`.

See Also

gb_trees(3) [page 183], orddict(3) [page 265]

digraph

Erlang Module

The `digraph` module implements a version of labeled directed graphs. What makes the graphs implemented here non-proper directed graphs is that multiple edges between vertices are allowed. However, the customary definition of directed graphs will be used in the text that follows.

A *directed graph* (or just "digraph") is a pair (V,E) of a finite set V of *vertices* and a finite set E of *directed edges* (or just "edges"). The set of edges E is a subset of VV (the Cartesian product of V with itself). In this module, V is allowed to be empty; the so obtained unique digraph is called the *empty digraph*. Both vertices and edges are represented by unique Erlang terms.

Digraphs can be annotated with additional information. Such information may be attached to the vertices and to the edges of the digraph. A digraph which has been annotated is called a *labeled digraph*, and the information attached to a vertex or an edge is called a *label*. Labels are Erlang terms.

An edge e=(v,w) is said to *emanate* from vertex v and to be *incident* on vertex w. The *out-degree* of a vertex is the number of edges emanating from that vertex. The *in-degree* of a vertex is the number of edges incident on that vertex. If there is an edge emanating from v and incident on w, then w is said to be an *out-neighbour* of v, and v is said to be an *in-neighbour* of w. A *path* P from v[1] to v[k] in a digraph (V,E) is a non-empty sequence v[1],v[2],...,v[k] of vertices in V such that there is an edge (v[i],v[i+1]) in E for 1<=i<k. The *length* of the path P is k-1. P is *simple* if all vertices are distinct, except that the first and the last vertices may be the same. P is a *cycle* if the length of P is not zero and v[1] = v[k]. A *loop* is a cycle of length one. A *simple cycle* is a path that is both a cycle and simple. An *acyclic digraph* is a digraph that has no cycles.

Exports

```
add_edge(G, E, V1, V2, Label) -> edge() | {error, Reason}
add_edge(G, V1, V2, Label) -> edge() | {error, Reason}
add_edge(G, V1, V2) -> edge() | {error, Reason}
```

Types:

- G = digraph()
- E = edge()
- V1 = V2 = vertex()
- Label = label()
- Reason = {bad_edge, Path} | {bad_vertex, V}
- Path = [vertex()]

add_edge/5 creates (or modifies) the edge E of the digraph G, using Label as the (new) label [page 102] of the edge. The edge is emanating [page 102] from V1 and incident [page 102] on V2. Returns E.

add_edge(G,V1,V2,Label) is equivalent to add_edge(G,E,V1,V2,Label), where E is a created edge. Tuples on the form ['$e'|N], where N is an integer>=1, are used for representing the created edges.

add_edge(G,V1,V2) is equivalent to add_edge(G,V1,V2,[]).

If the edge would create a cycle in an acyclic digraph [page 102], then {error,{bad_edge,Path}} is returned. If either of V1 or V2 is not a vertex of the digraph G, then {error,{bad_vertex,V}} is returned, V=V1 or V=V2.

```
add_vertex(G, V, Label) -> vertex()
add_vertex(G, V) -> vertex()
add_vertex(G) -> vertex()
```

Types:

- G = digraph()
- V = vertex()
- Label = label()

add_vertex/3 creates (or modifies) the vertex V of the digraph G, using Label as the (new) label [page 102] of the vertex. Returns V.

add_vertex(G,V) is equivalent to add_vertex(G,V,[]).

add_vertex/1 creates a vertex using the empty list as label, and returns the created vertex. Tuples on the form ['$v'|N], where N is an integer>=1, are used for representing the created vertices.

```
del_edge(G, E) -> true
```

Types:

- G = digraph()
- E = edge()

Deletes the edge E from the digraph G.

```
del_edges(G, Edges) -> true
```

Types:

- G = digraph()
- Edges = [edge()]

Deletes the edges in the list Edges from the digraph G.

```
del_path(G, V1, V2) -> true
```

Types:

- G = digraph()
- V1 = V2 = vertex()

Deletes edges from the digraph `G` until there are no paths [page 102] from the vertex `V1` to the vertex `V2`.

A sketch of the procedure employed: Find an arbitrary simple path [page 102] v[1],v[2],...,v[k] from `V1` to `V2` in `G`. Remove all edges of `G` emanating [page 102] from v[i] and incident [page 102] to v[i+1] for 1<=i<k (including multiple edges). Repeat until there is no path between `V1` and `V2`.

```
del_vertex(G, V) -> true
```

Types:

- G = digraph()
- V = vertex()

Deletes the vertex `V` from the digraph `G`. Any edges emanating [page 102] from `V` or incident [page 102] on `V` are also deleted.

```
del_vertices(G, Vertices) -> true
```

Types:

- G = digraph()
- Vertices = [vertex()]

Deletes the vertices in the list `Vertices` from the digraph `G`.

```
delete(G) -> true
```

Types:

- G = digraph()

Deletes the digraph `G`. This call is important because digraphs are implemented with `Ets`. There is no garbage collection of `Ets` tables. The digraph will, however, be deleted if the process that created the digraph terminates.

```
edge(G, E) -> {E, V1, V2, Label} | false
```

Types:

- G = digraph()
- E = edge()
- V1 = V2 = vertex()
- Label = label()

Returns {`E`,`V1`,`V2`,`Label`} where `Label` is the label [page 102] of the edge `E` emanating [page 102] from `V1` and incident [page 102] on `V2` of the digraph `G`. If there is no edge `E` of the digraph `G`, then `false` is returned.

```
edges(G) -> Edges
```

Types:

- G = digraph()
- Edges = [edge()]

Returns a list of all edges of the digraph `G`, in some unspecified order.

```
edges(G, V) -> Edges
```

Types:

- G = digraph()
- V = vertex()
- Edges = [edge()]

Returns a list of all edges emanating [page 102] from or incident [page 102] on `V` of the digraph `G`, in some unspecified order.

`get_cycle(G, V) -> Vertices | false`

Types:

- G = digraph()
- V1 = V2 = vertex()
- Vertices = [vertex()]

If there is a simple cycle [page 102] of length two or more through the vertex `V`, then the cycle is returned as a list `[V,...,V]` of vertices, otherwise if there is a loop [page 102] through `V`, then the loop is returned as a list `[V]`. If there are no cycles through `V`, then `false` is returned.

`get_path/3` is used for finding a simple cycle through `V`.

`get_path(G, V1, V2) -> Vertices | false`

Types:

- G = digraph()
- V1 = V2 = vertex()
- Vertices = [vertex()]

Tries to find a simple path [page 102] from the vertex `V1` to the vertex `V2` of the digraph `G`. Returns the path as a list `[V1,...,V2]` of vertices, or `false` if no simple path from `V1` to `V2` of length one or more exists.

The digraph `G` is traversed in a depth-first manner, and the first path found is returned.

`get_short_cycle(G, V) -> Vertices | false`

Types:

- G = digraph()
- V1 = V2 = vertex()
- Vertices = [vertex()]

Tries to find an as short as possible simple cycle [page 102] through the vertex `V` of the digraph `G`. Returns the cycle as a list `[V,...,V]` of vertices, or `false` if no simple cycle through `V` exists. Note that a loop [page 102] through `V` is returned as the list `[V,V]`.

`get_short_path/3` is used for finding a simple cycle through `V`.

`get_short_path(G, V1, V2) -> Vertices | false`

Types:

- G = digraph()
- V1 = V2 = vertex()
- Vertices = [vertex()]

Tries to find an as short as possible simple path [page 102] from the vertex `V1` to the vertex `V2` of the digraph `G`. Returns the path as a list `[V1,...,V2]` of vertices, or `false` if no simple path from `V1` to `V2` of length one or more exists.

The digraph `G` is traversed in a breadth-first manner, and the first path found is returned.

`in_degree(G, V) -> integer()`

Types:

- G= digraph()
- V = vertex()

Returns the in-degree [page 102] of the vertex `V` of the digraph `G`.

`in_edges(G, V) -> Edges`

Types:

- G = digraph()
- V = vertex()
- Edges = [edge()]

Returns a list of all edges incident [page 102] on `V` of the digraph `G`, in some unspecified order.

`in_neighbours(G, V) -> Vertices`

Types:

- G = digraph()
- V = vertex()
- Vertices = [vertex()]

Returns a list of all in-neighbours [page 102] of `V` of the digraph `G`, in some unspecified order.

`info(G) -> InfoList`

Types:

- G = digraph()
- InfoList = [{cyclicity, Cyclicity}, {memory, NoWords}, {protection, Protection}]
- Cyclicity = cyclic | acyclic
- Protection = protected | private
- NoWords = integer() >= 0

Returns a list of {`Tag, Value`} pairs describing the digraph `G`. The following pairs are returned:

- {`cyclicity, Cyclicity`}, where `Cyclicity` is `cyclic` or `acyclic`, according to the options given to `new`.
- {`memory, NoWords`}, where `NoWords` is the number of words allocated to the `ets` tables.
- {`protection, Protection`}, where `Protection` is `protected` or `private`, according to the options given to `new`.

`new() -> digraph()`

Equivalent to `new([])`.

`new(Type) -> digraph() | {error, Reason}`

Types:

- Type = [cyclic | acyclic | private | protected]
- Reason = {unknown_type, term()}

Returns an empty digraph [page 102] with properties according to the options in `Type`:

`cyclic` Allow cycles [page 102] in the digraph (default).

`acyclic` The digraph is to be kept acyclic [page 102].

`protected` Other processes can read the digraph (default).

`private` The digraph can be read and modified by the creating process only.

If an unrecognized type option T is given, then {`error`,{`unknown_type`,T}} is returned.

`no_edges(G) -> integer() >= 0`

Types:

- G = digraph()

Returns the number of edges of the digraph `G`.

`no_vertices(G) -> integer() >= 0`

Types:

- G = digraph()

Returns the number of vertices of the digraph `G`.

`out_degree(G, V) -> integer()`

Types:

- G = digraph()
- V = vertex()

Returns the out-degree [page 102] of the vertex `V` of the digraph `G`.

`out_edges(G, V) -> Edges`

Types:

- G = digraph()
- V = vertex()
- Edges = [edge()]

Returns a list of all edges emanating [page 102] from `V` of the digraph `G`, in some unspecified order.

`out_neighbours(G, V) -> Vertices`

Types:

- G = digraph()

- V = vertex()
- Vertices = [vertex()]

Returns a list of all out-neighbours [page 102] of `V` of the digraph `G`, in some unspecified order.

`vertex(G, V) -> {V, Label} | false`

Types:

- G = digraph()
- V = vertex()
- Label = label()

Returns {`V`,`Label`} where `Label` is the label [page 102] of the vertex `V` of the digraph `G`, or `false` if there is no vertex `V` of the digraph `G`.

`vertices(G) -> Vertices`

Types:

- G = digraph()
- Vertices = [vertex()]

Returns a list of all vertices of the digraph `G`, in some unspecified order.

See Also

digraph_utils(3) [page 109], ets(3) [page 139]

digraph_utils

Erlang Module

The `digraph_utils` module implements some algorithms based on depth-first traversal of directed graphs. See the `digraph` module for basic functions on directed graphs.

A *directed graph* (or just "digraph") is a pair (V,E) of a finite set V of *vertices* and a finite set E of *directed edges* (or just "edges"). The set of edges E is a subset of VV (the Cartesian product of V with itself).

Digraphs can be annotated with additional information. Such information may be attached to the vertices and to the edges of the digraph. A digraph which has been annotated is called a *labeled digraph*, and the information attached to a vertex or an edge is called a *label*.

An edge e=(v,w) is said to *emanate* from vertex v and to be *incident* on vertex w. If there is an edge emanating from v and incident on w, then w is said to be an *out-neighbour* of v, and v is said to be an *in-neighbour* of w. A *path* P from v[1] to v[k] in a digraph (V,E) is a non-empty sequence v[1],v[2],...,v[k] of vertices in V such that there is an edge (v[i],v[i+1]) in E for 1<=i<k. The *length* of the path P is k-1. P is a *cycle* if the length of P is not zero and v[1] = v[k]. A *loop* is a cycle of length one. An *acyclic digraph* is a digraph that has no cycles.

A *depth-first traversal* of a directed digraph can be viewed as a process that visits all vertices of the digraph. Initially, all vertices are marked as unvisited. The traversal starts with an arbitrarily chosen vertex, which is marked as visited, and follows an edge to an unmarked vertex, marking that vertex. The search then proceeds from that vertex in the same fashion, until there is no edge leading to an unvisited vertex. At that point the process backtracks, and the traversal continues as long as there are unexamined edges. If there remain unvisited vertices when all edges from the first vertex have been examined, some hitherto unvisited vertex is chosen, and the process is repeated.

A *partial ordering* of a set S is a transitive, antisymmetric and reflexive relation between the objects of S. The problem of *topological sorting* is to find a total ordering of S that is a superset of the partial ordering. A digraph G=(V,E) is equivalent to a relation E on V (we neglect the fact that the version of directed graphs implemented in the `digraph` module allows multiple edges between vertices). If the digraph has no cycles of length two or more, then the reflexive and transitive closure of E is a partial ordering.

A *subgraph* G' of G is a digraph whose vertices and edges form subsets of the vertices and edges of G. G' is *maximal* with respect to a property P if all other subgraphs that include the vertices of G' do not have the property P. A *strongly connected component* is a maximal subgraph such that there is a path between each pair of vertices. A *connected component* is a maximal subgraph such that there is a path between each pair of vertices, considering all edges undirected. An *arborescence* is an acyclic digraph with a vertex V, the *root*, such that there is a unique path from V to every other vertex of G. A *tree* is an acyclic non-empty digraph such that there is a unique path between every pair of vertices, considering all edges undirected.

Exports

`arborescence_root(Digraph) -> no | {yes, Root}`

Types:

- Digraph = digraph()
- Root = vertex()

Returns `{yes, Root}` if `Root` is the root [page 109] of the arborescence `Digraph`, `no` otherwise.

`components(Digraph) -> [Component]`

Types:

- Digraph = digraph()
- Component = [vertex()]

Returns a list of connected components [page 109]. Each component is represented by its vertices. The order of the vertices and the order of the components are arbitrary. Each vertex of the digraph `Digraph` occurs in exactly one component.

`condensation(Digraph) -> CondensedDigraph`

Types:

- Digraph = CondensedDigraph = digraph()

Creates a digraph where the vertices are the strongly connected components [page 109] of `Digraph` as returned by `strong_components/1`. If X and Y are strongly connected components, and there exist vertices x and y in X and Y respectively such that there is an edge emanating [page 109] from x and incident [page 109] on y, then an edge emanating from X and incident on Y is created.

The created digraph has the same type as `Digraph`. All vertices and edges have the default label [page 109] `[]`.

Each and every cycle [page 109] is included in some strongly connected component, which implies that there always exists a topological ordering [page 109] of the created digraph.

`cyclic_strong_components(Digraph) -> [StrongComponent]`

Types:

- Digraph = digraph()
- StrongComponent = [vertex()]

Returns a list of strongly connected components [page 109]. Each strongly component is represented by its vertices. The order of the vertices and the order of the components are arbitrary. Only vertices that are included in some cycle [page 109] in `Digraph` are returned, otherwise the returned list is equal to that returned by `strong_components/1`.

`is_acyclic(Digraph) -> bool()`

Types:

- Digraph = digraph()

Returns `true` if and only if the digraph `Digraph` is acyclic [page 109].

`is_arborescence(Digraph) -> bool()`

Types:

- Digraph = digraph()

Returns `true` if and only if the digraph `Digraph` is an arborescence [page 109].

`is_tree(Digraph) -> bool()`

Types:

- Digraph = digraph()

Returns `true` if and only if the digraph `Digraph` is a tree [page 109].

`loop_vertices(Digraph) -> Vertices`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns a list of all vertices of `Digraph` that are included in some loop [page 109].

`postorder(Digraph) -> Vertices`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns all vertices of the digraph `Digraph`. The order is given by a depth-first traversal [page 109] of the digraph, collecting visited vertices in postorder. More precisely, the vertices visited while searching from an arbitrarily chosen vertex are collected in postorder, and all those collected vertices are placed before the subsequently visited vertices.

`preorder(Digraph) -> Vertices`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns all vertices of the digraph `Digraph`. The order is given by a depth-first traversal [page 109] of the digraph, collecting visited vertices in pre-order.

`reachable(Vertices, Digraph) -> Vertices`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path [page 109] in `Digraph` from some vertex of `Vertices` to the vertex. In particular, since paths may have length zero, the vertices of `Vertices` are included in the returned list.

`reachable_neighbours(Vertices, Digraph) -> Vertices`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path [page 109] in `Digraph` of length one or more from some vertex of `Vertices` to the vertex. As a consequence, only those vertices of `Vertices` that are included in some cycle [page 109] are returned.

`reaching(Vertices, Digraph) -> Vertices`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path [page 109] from the vertex to some vertex of `Vertices`. In particular, since paths may have length zero, the vertices of `Vertices` are included in the returned list.

`reaching_neighbours(Vertices, Digraph) -> Vertices`

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path [page 109] of length one or more from the vertex to some vertex of `Vertices`. As a consequence, only those vertices of `Vertices` that are included in some cycle [page 109] are returned.

`strong_components(Digraph) -> [StrongComponent]`

Types:

- Digraph = digraph()
- StrongComponent = [vertex()]

Returns a list of strongly connected components [page 109]. Each strongly component is represented by its vertices. The order of the vertices and the order of the components are arbitrary. Each vertex of the digraph `Digraph` occurs in exactly one strong component.

`subgraph(Digraph, Vertices [, Options]) -> Subgraph | {error, Reason}`

Types:

- Digraph = Subgraph = digraph()
- Options = [{type, SubgraphType}, {keep_labels, bool()}]
- Reason = {invalid_option, term()} | {unknown_type, term()}
- SubgraphType = inherit | type()
- Vertices = [vertex()]

Creates a maximal subgraph [page 109] of `Digraph` having as vertices those vertices of `Digraph` that are mentioned in `Vertices`.

If the value of the option `type` is `inherit`, which is the default, then the type of `Digraph` is used for the subgraph as well. Otherwise the option value of `type` is used as argument to `digraph:new/1`.

If the value of the option `keep_labels` is `true`, which is the default, then the labels [page 109] of vertices and edges of `Digraph` are used for the subgraph as well. If the value is `false`, then the default label, `[]`, is used for the subgraph's vertices and edges.

`subgraph(Digraph, Vertices)` is equivalent to `subgraph(Digraph, Vertices, [])`.

```
topsort(Digraph) -> Vertices | false
```

Types:

- Digraph = digraph()
- Vertices = [vertex()]

Returns a topological ordering [page 109] of the vertices of the digraph `Digraph` if such an ordering exists, `false` otherwise. For each vertex in the returned list, there are no out-neighbours [page 109] that occur earlier in the list.

See Also

digraph(3) [page 102]

epp

Erlang Module

The Erlang code preprocessor includes functions which are used by `compile` to preprocess macros and include files before the actual parsing takes place.

Exports

`open(FileName, IncludePath) -> {ok,Epp} | {error, ErrorDescriptor}`
`open(FileName, IncludePath, PredefMacros) -> {ok,Epp} | {error, ErrorDescriptor}`

Types:

- FileName = atom() | string()
- IncludePath = [DirectoryName]
- DirectoryName = atom() | string()
- PredefMacros = [{atom(),term()}]
- Epp = pid() – handle to the epp server
- ErrorDescriptor = term()

Opens a file for preprocessing.

`close(Epp) -> ok`

Types:

- Epp = pid() – handle to the epp server

Closes the preprocessing of a file.

`parse_erl_form(Epp) -> {ok, AbsForm} | {eof, Line} | {error, ErrorInfo}`

Types:

- Epp = pid()
- AbsForm = term()
- Line = integer()
- ErrorInfo = see separate description below.

Returns the next Erlang form from the opened Erlang source file. The tuple {`eof`, `Line`} is returned at end-of-file. The first form corresponds to an implicit attribute `-file(File,1).`, where `File` is the name of the file.

`parse_file(FileName,IncludePath,PredefMacro) -> {ok,[Form]} | {error,OpenError}`

Types:

- FileName = atom() | string()

- IncludePath = [DirectoryName]
- DirectoryName = atom() | string()
- PredefMacros = [{atom(),term()}]
- Form = term() – same as returned by erl_parse:parse_form

Preprocesses and parses an Erlang source file. Note that the tuple {`eof, Line`} returned at end-of-file is included as a “form”.

Error Information

The `ErrorInfo` mentioned above is the standard `ErrorInfo` structure which is returned from all IO modules. It has the following format:

```
{ErrorLine, Module, ErrorDescriptor}
```

A string which describes the error is obtained with the following call:

```
apply(Module, format_error, ErrorDescriptor)
```

See Also

erl_parse(3) [page 125]

erl_eval

Erlang Module

This module provides an interpreter for Erlang expressions. The expressions are in the abstract syntax as returned by `erl_parse`, the Erlang parser, or a call to `io:parse_erl_exprs/2`.

Exports

`exprs(Expressions, Bindings) -> {value, Value, NewBindings}`

`exprs(Expressions, Bindings, LocalFunctionHandler) -> {value, Value, NewBindings}`

`exprs(Expressions, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {value, Value, NewBindings}`

Types:

- Expressions = as returned by erl_parse or io:parse_erl_exprs/2
- Bindings = as returned by bindings/1
- LocalFunctionHandler = {value, Func} | {eval, Func} | none
- NonlocalFunctionHandler = {value, Func} | none

Evaluates `Expressions` with the set of bindings `Bindings`, where `Expressions` is a sequence of expressions (in abstract syntax) of a type which may be returned by `io:parse_erl_exprs/2`. See below for an explanation of how and when to use the arguments `LocalFunctionHandler` and `NonlocalFunctionHandler`.

Returns `{value, Value, NewBindings}`

`expr(Expression, Bindings) -> { value, Value, NewBindings }`

`expr(Expression, Bindings, LocalFunctionHandler) -> { value, Value, NewBindings }`

`expr(Expression, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> { value, Value, NewBindings }`

Types:

- Expression = as returned by io:parse_erl_form/2, for example
- Bindings = as returned by bindings/1
- LocalFunctionHandler = {value, Func} | {eval, Func} | none
- NonlocalFunctionHandler = {value, Func} | none

Evaluates `Expression` with the set of bindings `Bindings`. `Expression` is an expression (in abstract syntax) of a type which may be returned by `io:parse_erl_form/2`. See below for an explanation of how and when to use the arguments `LocalFunctionHandler` and `NonlocalFunctionHandler`.

Returns `{value, Value, NewBindings}`.

```
expr_list(ExpressionList, Bindings) -> {ValueList, NewBindings}
expr_list(ExpressionList, Bindings, LocalFunctionHandler) -> {ValueList, NewBindings}
expr_list(ExpressionList, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) ->
        {ValueList, NewBindings}
```

Evaluates a list of expressions in parallel, using the same initial bindings for each expression. Attempts are made to merge the bindings returned from each evaluation. This function is useful in the `LocalFunctionHandler`. See below.

Returns `{ValueList, NewBindings}`.

```
new_bindings() -> BindingStruct
```

Returns an empty binding structure.

```
bindings(BindingStruct) -> Bindings
```

Returns the list of bindings contained in the binding structure.

```
binding(Name, BindingStruct) -> Binding
```

Returns the binding of `Name` in `BindingStruct`.

```
add_binding(Name, Value, Bindings) -> BindingStruct
```

Adds the binding `Name = Value` to `Bindings`. Returns an updated binding structure.

```
del_binding(Name, Bindings) -> BindingStruct
```

Removes the binding of `Name` in `Bindings`. Returns an updated binding structure.

Local Function Handler

During evaluation of a function, no calls can be made to local functions. An undefined function error would be generated. However, the optional argument `LocalFunctionHandler` may be used to define a function which is called when there is a call to a local function. The argument can have the following formats:

`{value,Func}` This defines a local function handler which is called with:

```
Func(Name, Arguments)
```

`Name` is the name of the local function (an atom) and `Arguments` is a list of the *evaluated* arguments. The function handler returns the value of the local function. In this case, it is not possible to access the current bindings. To signal an error, the function handler just calls `exit/1` with a suitable exit value.

`{eval,Func}` This defines a local function handler which is called with:

```
Func(Name, Arguments, Bindings)
```

`Name` is the name of the local function (an atom), `Arguments` is a list of the *unevaluated* arguments, and `Bindings` are the current variable bindings. The function handler returns:

```
{value,Value,NewBindings}
```

`Value` is the value of the local function and `NewBindings` are the updated variable bindings. In this case, the function handler must itself evaluate all the function arguments and manage the bindings. To signal an error, the function handler just calls `exit/1` with a suitable exit value.

`none` There is no local function handler.

Non-local Function Handler

The optional argument `NonlocalFunctionHandler` may be used to define a function which is called in the following cases: a functional object (fun) is called; a built-in function is called; a function is called using the M:F syntax, where M and F are atoms or expressions; an operator Op/A is called (this is handled as a call to the function `erlang:Op/A`). Exceptions are calls to `erlang:apply/2,3`; neither of the function handlers will be called for such calls. The argument can have the following formats:

`{value,Func}` This defines an nonlocal function handler which is called with:

```
Func(FuncSpec, Arguments)
```

`FuncSpec` is the name of the function on the form `{Module,Function}` or a fun, and `Arguments` is a list of the *evaluated* arguments. The function handler returns the value of the function. To signal an error, the function handler just calls `exit/1` with a suitable exit value.

`none` There is no nonlocal function handler.

> **Note:**
> For calls such as `erlang:apply(Fun, Args)` or `erlang:apply(Module, Function, Args)` the call of the non-local function handler corresponding to the call to `erlang:apply/2,3` itself–`Func({erlang, apply}, [Fun, Args])` or `Func({erlang, apply}, [Module, Function, Args])`–will never take place. The non-local function handler *will* however be called with the evaluated arguments of the call to `erlang:apply/2,3`: `Func(Fun, Args)` or `Func({Module, Function}, Args)` (assuming that `{Module, Function}` is not `{erlang, apply}`).

The nonlocal function handler argument is probably not used as frequently as the local function handler argument. A possible use is to call `exit/1` on calls to functions that for some reason are not allowed to be called.

Bugs

The evaluator is not complete. `receive` cannot be handled properly.

Any undocumented functions in `erl_eval` should not be used.

erl_expand_records

Erlang Module

Exports

```
module(AbsForms, CompileOptions) -> AbsForms
```

Types:

- AbsForms = [term()]
- CompileOptions = [term()]

Expands all records in a module. The returned module has no references to records, neither attributes nor code.

See Also

The [abstract format] documentation in ERTS User's Guide

erl_id_trans

Erlang Module

This module performs an identity parse transformation of Erlang code. It is included as an example for users who may wish to write their own parse transformers. If the option {parse_transform,Module} is passed to the compiler, a user written function parse_transform/2 is called by the compiler before the code is checked for errors.

Exports

```
parse_transform(Forms, Options) -> Forms
```

Types:

- Forms = [erlang_form()]
- Options = [compiler_options()]

Performs an identity transformation on Erlang forms, as an example.

Parse Transformations

Parse transformations are used if a programmer wants to use Erlang syntax, but with different semantics. The original Erlang code is then transformed into other Erlang code.

> **Note:**
> Programmers are strongly advised not to engage in parse transformations and no support is offered for problems encountered.

See Also

erl_parse(3) [page 125], compile(3).

erl_internal

Erlang Module

This module defines Erlang BIFs, guard tests and operators. This module is only of interest to programmers who manipulate Erlang code.

Exports

`bif(Name, Arity) -> bool()`

Types:

- Name = atom()
- Arity = integer()

Returns `true` if `Name/Arity` is an Erlang BIF which is automatically recognized by the compiler, otherwise `false`.

`guard_bif(Name, Arity) -> bool()`

Types:

- Name = atom()
- Arity = integer()

Returns `true` if `Name/Arity` is an Erlang BIF which is allowed in guards, otherwise `false`.

`type_test(Name, Arity) -> bool()`

Types:

- Name = atom()
- Arity = integer()

Returns `true` if `Name/Arity` is a valid Erlang type test, otherwise `false`.

`arith_op(OpName, Arity) -> bool()`

Types:

- OpName = atom()
- Arity = integer()

Returns `true` if `OpName/Arity` is an arithmetic operator, otherwise `false`.

`bool_op(OpName, Arity) -> bool()`

Types:

- OpName = atom()

- Arity = integer()

Returns `true` if `OpName/Arity` is a Boolean operator, otherwise `false`.

`comp_op(OpName, Arity) -> bool()`

Types:

- OpName = atom()
- Arity = integer()

Returns `true` if `OpName/Arity` is a comparison operator, otherwise `false`.

`list_op(OpName, Arity) -> bool()`

Types:

- OpName = atom()
- Arity = integer()

Returns `true` if `OpName/Arity` is a list operator, otherwise `false`.

`send_op(OpName, Arity) -> bool()`

Types:

- OpName = atom()
- Arity = integer()

Returns `true` if `OpName/Arity` is a send operator, otherwise `false`.

`op_type(OpName, Arity) -> Type`

Types:

- OpName = atom()
- Arity = integer()
- Type = arith | bool | comp | list | send

Returns the `Type` of operator that `OpName/Arity` belongs to, or generates a `function_clause` error if it is not an operator at all.

erl_lint

Erlang Module

This module is used to check Erlang code for illegal syntax and other bugs. It also warns against coding practices which are not recommended.

The errors detected include:

- redefined and undefined functions
- unbound and unsafe variables
- illegal record usage.

Warnings include:

- unused functions and imports
- unused variables
- variables imported into matches
- variables exported from `if`/`case`/`receive`
- variables shadowed in lambdas and list comprehensions.

Some of the warnings are optional, and can be turned on by giving the appropriate option, described below.

The functions in this module are invoked automatically by the Erlang compiler and there is no reason to invoke these functions separately unless you have written your own Erlang compiler.

Exports

```
module(AbsForms) -> {ok,Warnings} | {error,Errors,Warnings}
module(AbsForms, FileName) -> {ok,Warnings} | {error,Errors,Warnings}
module(AbsForms, FileName, CompileOptions) -> {ok,Warnings} | {error,Errors,Warnings}
```

Types:

- AbsForms = [term()]
- FileName = FileName2 = atom() | string()
- Warnings = Errors = [{Filename2,[ErrorInfo]}]
- ErrorInfo = see separate description below.
- CompileOptions = [term()]

This function checks all the forms in a module for errors. It returns:

`{ok,Warnings}` There were no errors in the module.

`{error,Errors,Warnings}` There were errors in the module.

Since this module is of interest only to the maintainers of the compiler, and to avoid having the same description in two places to avoid the usual maintainence nightmare, the elements of `Options` that control the warnings are only described in [compile(3)].

The `AbsForms` of a module which comes from a file that is read through `epp`, the Erlang pre-processor, can come from many files. This means that any references to errors must include the file name (see epp(3) [page 114], or parser erl_parse(3) [page 125] The warnings and errors returned have the following format:

```
[{FileName2,[ErrorInfo]}]
```

The errors and warnings are listed in the order in which they are encountered in the forms. This means that the errors from one file may be split into different entries in the list of errors.

`is_guard_test(Expr) -> bool()`

Types:

- Expr = term()

This function tests if `Expr` is a legal guard test. `Expr` is an Erlang term representing the abstract form for the expression. `erl_parse:parse_exprs(Tokens)` can be used to generate a list of `Expr`.

`format_error(ErrorDescriptor) -> Chars`

Types:

- ErrorDescriptor = errordesc()
- Chars = [char() | Chars]

Takes an `ErrorDescriptor` and returns a string which describes the error or warning. This function is usually called implicitly when processing an `ErrorInfo` structure (see below).

Error Information

The `ErrorInfo` mentioned above is the standard `ErrorInfo` structure which is returned from all IO modules. It has the following format:

```
{ErrorLine, Module, ErrorDescriptor}
```

A string which describes the error is obtained with the following call:

```
apply(Module, format_error, ErrorDescriptor)
```

See Also

erl_parse(3) [page 125], epp(3) [page 114]

erl_parse

Erlang Module

This module is the basic Erlang parser which converts tokens into the abstract form of either forms (i.e., top-level constructs), expressions, or terms. The Abstract Format is described in the ERTS User's Guide. Note that a token list must end with the *dot* token in order to be acceptable to the parse functions (see erl_scan).

Exports

`parse_form(Tokens) -> {ok, AbsForm} | {error, ErrorInfo}`

Types:

- Tokens = [Token]
- Token = {Tag,Line} | {Tag,Line,term()}
- Tag = atom()
- AbsForm = term()
- ErrorInfo = see section Error Information below.

This function parses `Tokens` as if it were a form. It returns:

`{ok, AbsForm}` The parsing was successful. `AbsForm` is the abstract form of the parsed form.

`{error, ErrorInfo}` An error occurred.

`parse_exprs(Tokens) -> {ok, Expr_list} | {error, ErrorInfo}`

Types:

- Tokens = [Token]
- Token = {Tag,Line} | {Tag,Line,term()}
- Tag = atom()
- Expr_list = [AbsExpr]
- AbsExpr = term()
- ErrorInfo = see section Error Information below.

This function parses `Tokens` as if it were a list of expressions. It returns:

`{ok, Expr_list}` The parsing was successful. `Expr_list` is a list of the abstract forms of the parsed expressions.

`{error, ErrorInfo}` An error occurred.

`parse_term(Tokens) -> {ok, Term} | {error, ErrorInfo}`

Types:

- Tokens = [Token]
- Token = {Tag,Line} | {Tag,Line,term()}
- Tag = atom()
- Term = term()
- ErrorInfo = see section Error Information below.

This function parses `Tokens` as if it were a term. It returns:

{`ok, Term`} The parsing was successful. `Term` is the Erlang term corresponding to the token list.

{`error, ErrorInfo`} An error occurred.

`format_error(ErrorDescriptor) -> Chars`

Types:

- ErrorDescriptor = errordesc()
- Chars = [char() | Chars]

Uses an `ErrorDescriptor` and returns a string which describes the error. This function is usually called implicitly when an `ErrorInfo` structure is processed (see below).

`tokens(AbsTerm) -> Tokens`
`tokens(AbsTerm, MoreTokens) -> Tokens`

Types:

- Tokens = MoreTokens = [Token]
- Token = {Tag,Line} | {Tag,Line,term()}
- Tag = atom()
- AbsTerm = term()
- ErrorInfo = see section Error Information below.

This function generates a list of tokens representing the abstract form `AbsTerm` of an expression. Optionally, it appends `Moretokens`.

`normalise(AbsTerm) -> Data`

Types:

- AbsTerm = Data = term()

Converts the abstract form `AbsTerm` of a term into a conventional Erlang data structure (i.e., the term itself). This is the inverse of `abstract/1`.

`abstract(Data) -> AbsTerm`

Types:

- Data = AbsTerm = term()

Converts the Erlang data structure `Data` into an abstract form of type `AbsTerm`. This is the inverse of `normalise/1`.

Error Information

The `ErrorInfo` mentioned above is the standard `ErrorInfo` structure which is returned from all IO modules. It has the format:

```
{ErrorLine, Module, ErrorDescriptor}
```

A string which describes the error is obtained with the following call:

```
apply(Module, format_error, ErrorDescriptor)
```

See Also

io(3) [page 219], erl_scan(3) [page 131], ERTS User's Guide

erl_pp

Erlang Module

The functions in this module are used to generate aesthetically attractive representations of abstract forms, which are suitable for printing. All functions return (possibly deep) lists of characters and generate an error if the form is wrong.

All functions can have an optional argument which specifies a hook that is called if an attempt is made to print an unknown form.

Exports

```
form(Form) -> DeepCharList
form(Form, HookFunction) -> DeepCharList
```

Types:

- Form = term()
- HookFunction = see separate description below.
- DeepCharList = [char()|DeepCharList]

Pretty prints a `Form` which is an abstract form of a type which is returned by `erl_parse:parse_form`.

```
attribute(Attribute) -> DeepCharList
attribute(Attribute, HookFunction) -> DeepCharList
```

Types:

- Attribute = term()
- HookFunction = see separate description below.
- DeepCharList = [char()|DeepCharList]

The same as `form`, but only for the attribute `Attribute`.

```
function(Function) -> DeepCharList
function(Function, HookFunction) -> DeepCharList
```

Types:

- Function = term()
- HookFunction = see separate description below.
- DeepCharList = [char()|DeepCharList]

The same as `form`, but only for the function `Function`.

```
guard(Guard) -> DeepCharList
guard(Guard, HookFunction) -> DeepCharList
```

Types:

- Form = term()
- HookFunction = see separate description below.
- DeepCharList = [char() | DeepCharList]

The same as `form`, but only for the guard test `Guard`.

```
exprs(Expressions) -> DeepCharList
exprs(Expressions, HookFunction) -> DeepCharList
exprs(Expressions, Indent, HookFunction) -> DeepCharList
```

Types:

- Expressions = term()
- HookFunction = see separate description below.
- Indent = integer()
- DeepCharList = [char() | DeepCharList]

The same as `form`, but only for the sequence of expressions in `Expressions`.

```
expr(Expression) -> DeepCharList
expr(Expression, HookFunction) -> DeepCharList
expr(Expression, Indent, HookFunction) -> DeepCharList
expr(Expression, Indent, Precedence, HookFunction) ->-> DeepCharList
```

Types:

- Expression = term()
- HookFunction = see separate description below.
- Indent = integer()
- Precedence =
- DeepCharList = [char() | DeepCharList]

This function prints one expression. It is useful for implementing hooks (see below).

Unknown Expression Hooks

The optional argument `HookFunction`, shown in the functions described above, defines a function which is called when an unknown form occurs where there should be a valid expression. It can have the following formats:

`Function` The hook function is called by:

```
Function(Expr,
         CurrentIndentation,
         CurrentPrecedence,
         HookFunction)
```

`none` There is no hook function

The called hook function should return a (possibly deep) list of characters. `expr/4` is useful in a hook.

If `CurrentIndentation` is negative, there will be no line breaks and only a space is used as a separator.

Bugs

It should be possible to have hook functions for unknown forms at places other than expressions.

See Also

io(3) [page 219], erl_parse(3) [page 125], erl_eval(3) [page 116]

erl_scan

Erlang Module

This module contains functions for tokenizing characters into Erlang tokens.

Exports

`string(CharList,StartLine]) -> {ok, Tokens, EndLine} | Error`
`string(CharList) -> {ok, Tokens, EndLine} | Error`

Types:

- CharList = string()
- StartLine = EndLine = Line = integer()
- Tokens = [{atom(),Line}|{atom(),Line,term()}]
- Error = {error, ErrorInfo, EndLine}

Takes the list of characters `CharList` and tries to scan (tokenize) them. Returns {`ok, Tokens, EndLine`}, where `Tokens` are the Erlang tokens from `CharList`. `EndLine` is the last line where a token was found.

`StartLine` indicates the initial line when scanning starts. `string/1` is equivalent to `string(CharList,1)`.

{`error, ErrorInfo, EndLine`} is returned if an error occurs. `EndLine` indicates where the error occurred.

`tokens(Continuation, CharList, StartLine) ->Return`

Types:

- Return = {done, Result, LeftOverChars} | {more, Continuation}
- Continuation = [] | string()
- CharList = string()
- StartLine = EndLine = integer()
- Result = {ok, Tokens, EndLine} | {eof, EndLine}
- Tokens = [{atom(),Line}|{atom(),Line,term()}]

This is the re-entrant scanner which scans characters until a *dot* ('.' whitespace) has been reached. It returns:

{`done, Result, LeftOverChars`} This return indicates that there is sufficient input data to get an input. `Result` is:

{`ok, Tokens, EndLine`} The scanning was successful. `Tokens` is the list of tokens including *dot*.

{`eof, EndLine`} End of file was encountered before any more tokens.

{error, ErrorInfo, EndLine} An error occurred.

{more, Continuation} More data is required for building a term. Continuation must be passed in a new call to tokens/3 when more data is available.

reserved_word(Atom) -> bool()

Returns true if Atom is an Erlang reserved word, otherwise false.

format_error(ErrorDescriptor) -> string()

Types:

- ErrorDescriptor = errordesc()

Takes an ErrorDescriptor and returns a string which describes the error or warning. This function is usually called implicitly when processing an ErrorInfo structure (see below).

Error Information

The ErrorInfo mentioned above is the standard ErrorInfo structure which is returned from all IO modules. It has the following format:

```
{ErrorLine, Module, ErrorDescriptor}
```

A string which describes the error is obtained with the following call:

```
apply(Module, format_error, ErrorDescriptor)
```

Notes

The continuation of the first call to the re-entrant input functions must be []. Refer to Armstrong, Virding and Williams, 'Concurrent Programming in Erlang', Chapter 13, for a complete description of how the re-entrant input scheme works.

See Also

io(3) [page 219], erl_parse(3) [page 125]

erl_tar

Erlang Module

The `erl_tar` module archives and extract files to and from a tar file. The tar file format is the POSIX extended tar file format specified in IEEE Std 1003.1 and ISO/IEC9945-1. That is the same format as used by `tar` program on Solaris, but is not the same as used by the GNU tar program.

By convention, the name of a tar file should end in "`.tar`". To abide to the convention, you'll need to add "`.tar`" yourself to the name.

Tar files can be created in one operation using the create/2 [page 135] or create/3 [page 135] function.

Alternatively, for more control, the open [page 137], add/3,4 [page 134], and close/1 [page 134] functions can be used.

To extract all files from a tar file, use the extract/1 [page 135] function. To extract only some files or to be able to specify some more options, use the extract/2 [page 136] function.

To return a list of the files in a tar file, use either the table/1 [page 137] or table/2 [page 137] function. To print a list of files to the Erlang shell, use either the t/1 [page 138] or tt/1 [page 138] function.

To convert an error term returned from one of the functions above to a readable message, use the format_error/1 [page 136] function.

LIMITATIONS

For maximum compatibility, it is safe to archive files with names up to 100 characters in length. Such tar files can generally be extracted by any `tar` program.

If filenames exceed 100 characters in length, the resulting tar file can only be correctly extracted by a POSIX-compatible `tar` program (such as Solaris `tar`), not by GNU tar.

File have longer names than 256 bytes cannot be stored at all.

The filename of the file a symbolic link points is always limited to 100 characters.

Exports

`add(TarDescriptor, Filename, Options) -> RetValue`

Types:

- TarDescriptor = term()
- Filename = filename()
- Options = [Option]
- Option = dereference|verbose
- RetValue = ok|{error,{Filename,Reason}}
- Reason = term()

The `add/3` function adds a file to a tar file that has been opened for writing by open/1 [page 137].

`dereference` By default, symbolic links will be stored as symbolic links in the tar file. Use the `dereference` option to override the default and store the file that the symbolic link points to into the tar file.

`verbose` Print an informational message about the file being added.

`add(TarDescriptor, FilenameOrBin, NameInArchive, Options) -> RetValue`

Types:

- TarDescriptor = term()
- FilenameOrBin = Filename()|binary()
- Filename = filename()()
- NameInArchive = filename()
- Options = [Option]
- Option = dereference|verbose
- RetValue = ok|{error,{Filename,Reason}}
- Reason = term()

The `add/4` function adds a file to a tar file that has been opened for writing by open/1 [page 137]. It accepts the same options as add/3 [page 134].

`NameInArchive` is the name under which the file will be stored in the tar file. That is the name that the file will get when it will be extracted from the tar file.

`close(TarDescriptor)`

Types:

- TarDescriptor = term()

The `close/1` function closes a tar file opened by open/1 [page 137].

`create(Name, FileList) ->RetValue`

Types:

- Name = filename()
- FileList = [Filename|{NameInArchive, binary()},{NameInArchive, Filename}]
- Filename = filename()
- NameInArchive = filename()

- RetValue = ok|{error,{Name,Reason}} <V>Reason = term()

The `create/2` function creates a tar file and archives the files whose names are given in `FileList` into it. The files may either be read from disk or given as binaries.

`create(Name, FileList, OptionList)`

Types:

- Name = filename()
- FileList = [Filename|{NameInArchive, binary()},{NameInArchive, Filename}]
- Filename = filename()
- NameInArchive = filename()
- OptionList = [Option]
- Option = compressed|cooked|dereference|verbose
- RetValue = ok|{error,{Name,Reason}} <V>Reason = term()

The `create/3` function creates a tar file and archives the files whose names are given in `FileList` into it. The files may either be read from disk or given as binaries.

The options in `OptionList` modify the defaults as follows.

`compressed` The entire tar file will be compressed, as if it has been run through the `gzip` program. To abide to the convention that a compressed tar file should end in "`.tar.gz`" or "`.tgz`", you'll need to add the appropriate extension yourself.

`cooked` By default, the `open/2` function will open the tar file in `raw` mode, which is faster but does not allow a remote (erlang) file server to be used. Adding `cooked` to the mode list will override the default and open the tar file without the `raw` option.

`dereference` By default, symbolic links will be stored as symbolic links in the tar file. Use the `dereference` option to override the default and store the file that the symbolic link points to into the tar file.

`verbose` Print an informational message about each file being added.

`extract(Name) -> RetValue`

Types:

- Name = filename()
- RetValue = ok|{error,{Name,Reason}}
- Reason = term()

The `extract/1` function extracts all files from a tar archive.

If the `Name` argument is given as "`{binary,Binary}`", the contents of the binary is assumed to be a tar archive.

If the `Name` argument is given as "`{file,Fd}`", `Fd` is assumed to be a file descriptor returned from the `file:open/2` function.

Otherwise, `Name` should be a filename.

`extract(Name, OptionList)`

Types:

- Name = filename() | {binary,Binary} | {file,Fd}
- Binary = binary()
- Fd = file_descriptor()

- OptionList = [Option]
- Option = {cwd,Cwd} | {files,FileList} | keep_old_files | verbose | memory
- Cwd = [dirname()]
- FileList = [filename()]
- RetValue = ok | MemoryRetValue | {error,{Name,Reason}}
- MemoryRetValue = {ok, [{NameInArchive,binary()}]}
- NameInArchive = filename()
- Reason = term()

The `extract/2` function extracts files from a tar archive.

If the `Name` argument is given as "`{binary,Binary}`", the contents of the binary is assumed to be a tar archive.

If the `Name` argument is given as "`{file,Fd}`", `Fd` is assumed to be a file descriptor returned from the `file:open/2` function.

Otherwise, `Name` should be a filename.

The following options modify the defaults for the extraction as follows.

`{cwd,Cwd}` Files with relative filenames will by default be extracted to the current working directory. Given the `{cwd,Cwd}` option, the `extract/2` function will extract into the directory `Cwd` instead of to the current working directory.

`{files,FileList}` By default, all files will be extracted from the tar file. Given the `{files,Files}` option, the `extract/2` function will only extract the files whose names are included in `FileList`.

`compressed` Given the `compressed` option, the `extract/2` function will uncompress the file while extracting If the tar file is not actually compressed, the `compressed` will effectively be ignored.

`cooked` By default, the `open/2` function will open the tar file in `raw` mode, which is faster but does not allow a remote (erlang) file server to be used. Adding `cooked` to the mode list will override the default and open the tar file without the `raw` option.

`memory` Instead of extracting to a directory, the memory option will give the result as a list of tuples {Filename, Binary}, where Binary is a binary containing the extracted data of the file named Filename in the tar file.

`keep_old_files` By default, all existing files with the same name as file in the tar file will be overwritten Given the `keep_old_files` option, the `extract/2` function will not overwrite any existing files.

`verbose` Print an informational message as each file is being extracted.

`format_error(Reason) -> string()`

Types:

- Reason = term()

The `format_error/1` converts an error reason term to a human-readable error message string.

`open(Name, OpenModeList) -> RetValue`

Types:

- Name = filename()

- OpenModeList = [OpenMode]
- Mode = read|write|compressed|cooked
- RetValue = {ok,TarDescriptor}|{error,{Name,Reason}} <V>TarDescriptor = term()
- Reason = term()

The open/2 function opens a tar file.

By convention, the name of a tar file should end in ".tar". To abide to the convention, you'll need to add ".tar" yourself to the name.

Note that there is currently no function for reading from an opened tar file, meaning that opening a tar file for reading is not very useful.

Except for read and write (which are mutually exclusive), the following atoms may be added to OpenModeList:

compressed The entire tar file will be compressed, as if it has been run through the gzip program. To abide to the convention that a compressed tar file should end in ".tar.gz" or ".tgz", you'll need to add the appropriate extension yourself.

cooked By default, the open/2 function will open the tar file in raw mode, which is faster but does not allow a remote (erlang) file server to be used. Adding cooked to the mode list will override the default and open the tar file without the raw option.

Use the add/3,4 [page 134] functions to add one file at the time into an opened tar file. When you are finished adding files, use the close [page 134] function to close the tar file.

Warning:
The TarDescriptor term is not a file descriptor. You should not rely on the specific contents of the TarDescriptor term, as it may change in future versions as more features are added to the erl_tar module.

```
table(Name) -> RetValue
```

Types:

- Name = filename()
- RetValue = {ok,[string()]}|{error,{Name,Reason}}
- Reason = term()

The table/1 function retrieves the names of all files in the tar file Name.

```
table(Name, Options)
```

Types:

- Name = filename()

The table/2 function retrieves the names of all files in the tar file Name.

```
t(Name)
```

Types:

- Name = filename()

The `t/1` function prints the names of all files in the tar file `Name` to the Erlang shell. (Similar to "`tart`".)

`tt(Name)`

Types:

- Name = filename()

The `tt/1` function prints names and information about all files in the tar file `Name` to the Erlang shell. (Similar to "`tartv`".)

ets

Erlang Module

This module is an interface to the Erlang built-in term storage BIFs. These provide the ability to store very large quantities of data in an Erlang runtime system, and to have constant access time to the data. (In the case of `ordered_set`, see below, access time is proportional to the logarithm of the number of objects stored).

Data is organized as a set of dynamic tables, which can store tuples. Each table is created by a process. When the process terminates, the table is automatically destroyed. Every table has access rights set at creation.

Tables are divided into four different types, `set`, `ordered_set`, `bag` and `duplicate_bag`. A `set` or `ordered_set` table can only have one object associated with each key. A `bag` or `duplicate_bag` can have many objects associated with each key.

The number of tables stored at one Erlang node is limited. The current default limit is approximately 1400 tables. The upper limit can be increased by setting the environment variable `ERL_MAX_ETS_TABLES` before starting the Erlang runtime system (i.e. with the `-env` option to `erl`/`werl`). The actual limit may be slightly higher than the one specified, but never lower.

Note that there is no automatic garbage collection for tables. Even if there are no references to a table from any process, it will not automatically be destroyed unless the owner process terminates. It can be destroyed explicitly by using `delete/1`.

Some implementation details:

- In the current implementation, every object insert and look-up operation results in a copy of the object.
- This module provides very limited support for concurrent updates. No locking is available, but the `safe_fixtable/2` function can be used to guarantee that a sequence of `first/1` and `next/2` calls will traverse the table without errors and that each object in the table is visited exactly once, even if another process (or the same process) simultaneously deletes or inserts objects into the table. Nothing more is guaranteed; in particular any object inserted during a traversal *may* be visited in the traversal.
- `'$end_of_table'` should not be used as a key since this atom is used to mark the end of the table when using `first`/`next`.

Also worth noting is the subtle difference between *matching* and *comparing equal*, which is demonstrated by the different table types `set` and `ordered_set`. Two Erlang terms `match` if they are of the same type and have the same value, so that `1` matches `1`, but not `1.0` (as `1.0` is a `float()` and not an `integer()`). Two Erlang terms *compare equal* if they either are of the same type and value, or if both are numeric types and extend to the same value, so that `1` compares equal to both `1` and `1.0`. The `ordered_set` works on the *Erlang term order* and there is no defined order between an `integer()` and a `float()` that extends to the same value, hence the key `1` and the key `1.0` are regarded as equal in an `ordered_set` table.

In general, the functions below will exit with reason badarg if any argument is of the wrong format, or if the table identifier is invalid.

Match Specifications

Some of the functions uses a *match specification*, match_spec. A brief explanation is given in select/2 [page 155]. For a detailed description, see the chapter “Match specifications in Erlang” in *ERTS User’s Guide*.

DATA TYPES

```
match_spec()
  a match specification, see above

tid()
  a table identifier, as returned by new/2
```

Exports

`all() -> [Tab]`

Types:

- Tab = tid() | atom()

Returns a list of all tables at the node. Named tables are given by their names, unnamed tables are given by their table identifiers.

`delete(Tab) -> true`

Types:

- Tab = tid() | atom()

Deletes the entire table Tab.

`delete(Tab, Key) -> true`

Types:

- Tab = tid() | atom()
- Key = term()

Deletes all objects with the key Key from the table Tab.

`delete_all_objects(Tab) -> true`

Types:

- Tab = tid() | atom()

Delete all objects in the ETS table Tab. The deletion is atomic.

`delete_object(Tab,Object) -> true`

Types:

- Tab = tid() | atom()
- Object = tuple()

Delete the exact object `Object` from the ETS table, leaving objects with the same key but other differences (useful for type `bag`).

`file2tab(Filename) -> {ok,Tab} | {error,Reason}`

Types:

- Filename = string() | atom()
- Tab = tid() | atom()
- Reason = term()

Reads a file produced by tab2file/2 [page 158] or tab2file/3 [page 158] and creates the corresponding table `Tab`.

Equivalent to `file2tab(Filename,[])`.

`file2tab(Filename,Options) -> {ok,Tab} | {error,Reason}`

Types:

- Filename = string() | atom()
- Tab = tid() | atom()
- Options = [Option]
- Option = {verify, bool()}
- Reason = term()

Reads a file produced by tab2file/2 [page 158] or tab2file/3 [page 158] and creates the corresponding table `Tab`.

The currently only supported option is `{verify,bool()}`. If verification is turned on (by means of specifying `{verify,true}`), the function utilizes whatever information is present in the file to assert that the information is not damaged. How this is done depends on which `extended_info` was written using tab2file/3 [page 158].

If no `extended_info` is present in the file and `{verify,true}` is specified, the number of objects written is compared to the size of the original table when the dump was started. This might make verification fail if the table was `public` and objects were added or removed while the table was dumped to file. To avoid this type of problems, either do not verify files dumped while updated simultaneously or use the `{extended_info, [object_count]}` option to tab2file/3 [page 158], which extends the information in the file with the number of objects actually written.

If verification is turned on and the file was written with the option `{extended_info, [md5sum]}`, reading the file is slower and consumes radically more CPU time than otherwise.

`{verify,false}` is the default.

`first(Tab) -> Key | '$end_of_table'`

Types:

- Tab = tid() | atom()
- Key = term()

Returns the first key Key in the table Tab. If the table is of the ordered_set type, the first key in Erlang term order will be returned. If the table is of any other type, the first key according to the table's internal order will be returned. If the table is empty, '$end_of_table' will be returned.

Use next/2 to find subsequent keys in the table.

```
fixtable(Tab, true|false) -> true | false
```

Types:

- Tab = tid() | atom()

> **Warning:**
> The function is retained for backwards compatibility only. Use safe_fixtable/2 instead.

Fixes a table for safe traversal. The function is primarily used by the Mnesia DBMS to implement functions which allow write operations in a table, although the table is in the process of being copied to disk or to another node. It does not keep track of when and how tables are fixed.

```
foldl(Function, Acc0, Tab) -> Acc1
```

Types:

- Function = fun(A, AccIn) -> AccOut
- Tab = tid() | atom()
- Acc0 = Acc1 = AccIn = AccOut = term()

Acc0 is returned if the table is empty. This function is similar to lists:foldl/3. The order in which the elements of the table are traversed is unspecified, except for tables of type ordered_set, for which they are traversed first to last.

If Function inserts objects into the table, or another process inserts objects into the table, those objects *may* (depending on key ordering) be included in the traversal.

```
foldr(Function, Acc0, Tab) -> Acc1
```

Types:

- Function = fun(A, AccIn) -> AccOut
- Tab = tid() | atom()
- Acc0 = Acc1 = AccIn = AccOut = term()

Acc0 is returned if the table is empty. This function is similar to lists:foldr/3. The order in which the elements of the table are traversed is unspecified, except for tables of type ordered_set, for which they are traversed last to first.

If Function inserts objects into the table, or another process inserts objects into the table, those objects *may* (depending on key ordering) be included in the traversal.

```
from_dets(Tab, DetsTab) -> true
```

Types:

- Tab = tid() | atom()

- DetsTab = atom()

Fills an already created ETS table with the objects in the already opened Dets table named `DetsTab`. The existing objects of the ETS table are kept unless overwritten.

Throws a badarg error if any of the tables does not exist or the dets table is not open.

`fun2ms(LiteralFun) -> MatchSpec`

Types:

- LiteralFun – see below
- MatchSpec = match_spec()

Pseudo function that by means of a `parse_transform` translates `LiteralFun` typed as parameter in the function call to a match_spec [page 140]. With "literal" is meant that the fun needs to textually be written as the parameter of the function, it cannot be held in a variable which in turn is passed to the function).

The parse transform is implemented in the module `ms_transform` and the source *must* include the file `ms_transform.hrl` in `stdlib` for this pseudo function to work. Failing to include the hrl file in the source will result in a runtime error, not a compile time ditto. The include file is easiest included by adding the line `-include_lib("stdlib/include/ms_transform.hrl").` to the source file.

The fun is very restricted, it can take only a single parameter (the object to match): a sole variable or a tuple. It needs to use the `is_XXX` guard tests. Language constructs that have no representation in a match_spec (like `if`, `case`, `receive` etc) are not allowed.

The return value is the resulting match_spec.

Example:

```
1> ets:fun2ms(fun({M,N}) when N > 3 -> M end).
[{{'$1','$2'},[{'>','$2',3}],['$1']}]
```

Variables from the environment can be imported, so that this works:

```
2> X=3.
3
3> ets:fun2ms(fun({M,N}) when N > X -> M end).
[{{'$1','$2'},[{'>','$2',{const,3}}],['$1']}]
```

The imported variables will be replaced by match_spec `const` expressions, which is consistent with the static scoping for Erlang funs. Local or global function calls can not be in the guard or body of the fun however. Calls to builtin match_spec functions of course is allowed:

```
4> ets:fun2ms(fun({M,N}) when N > X, is_atomm(M) -> M end).
Error: fun containing local Erlang function calls
('is_atomm' called in guard) cannot be translated into match_spec
{error,transform_error}
5> ets:fun2ms(fun({M,N}) when N > X, is_atom(M) -> M end).
[{{'$1','$2'},[{'>','$2',{const,3}},{is_atom,'$1'}],['$1']}]
```

As can be seen by the example, the function can be called from the shell too. The fun needs to be literally in the call when used from the shell as well. Other means than the parse_transform are used in the shell case, but more or less the same restrictions apply (the exception being records, as they are not handled by the shell).

> **Warning:**
> If the parse_transform is not applied to a module which calls this pseudo function, the call will fail in runtime (with a `badarg`). The module `ets` actually exports a function with this name, but it should never really be called except for when using the function in the shell. If the `parse_transform` is properly applied by including the `ms_transform.hrl` header file, compiled code will never call the function, but the function call is replaced by a literal match_spec.

For more information, see ms_transform(3) [page 254].

`i() -> void()`

Displays information about all ETS tables on tty.

`i(Tab) -> void()`

Types:

- Tab = tid() | atom()

Browses the table `Tab` on tty.

`info(Tab) -> [{Item, Value}] | undefined`

Types:

- Tab = tid() | atom()
- Item = atom(), see below
- Value = term(), see below

Returns information about the table `Tab` as a list of `{Item, Value}` tuples. If `Tab` has the correct type for a table identifier, but does not refer to an existing ETS table, `undefined` is returned. If `Tab` is not of the correct type, this function fails with reason `badarg`.

- `Item=memory, Value=int()`
  The number of words allocated to the table.
- `Item=owner, Value=pid()`
  The pid of the owner of the table.
- `Item=name, Value=atom()`
  The name of the table.
- `Item=size, Value=int()`
  The number of objects inserted in the table.
- `Item=node, Value=atom()`
  The node where the table is stored. This field is no longer meaningful as tables cannot be accessed from other nodes.

- `Item=named_table, Value=true|false`
  Indicates if the table is named or not.
- `Item=type, Value=set|ordered_set|bag|duplicate_bag`
  The table type.
- `Item=keypos, Value=int()`
  The key position.
- `Item=protection, Value=public|protected|private`
  The table access rights.

`info(Tab, Item) -> Value | undefined`

Types:

- Tab = tid() | atom()
- Item, Value - see below

Returns the information associated with `Item` for the table `Tab`, or returns `undefined` if `Tab` does not refer an existing ETS table. If `Tab` is not of the correct type, or if `Item` is not one of the allowed values, this function fails with reason `badarg`.

> **Warning:**
> In R11B and earlier, this function would not fail but return `undefined` for invalid values for `Item`.

In addition to the {`Item`,`Value`} pairs defined for `info/1`, the following items are allowed:

- `Item=fixed, Value=true|false`
  Indicates if the table is fixed by any process or not.
- `Item=safe_fixed, Value={FirstFixed,Info}|false`
  If the table has been fixed using `safe_fixtable/2`, the call returns a tuple where `FirstFixed` is the time when the table was first fixed by a process, which may or may not be one of the processes it is fixed by right now.
  `Info` is a possibly empty lists of tuples {`Pid`,`RefCount`}, one tuple for every process the table is fixed by right now. `RefCount` is the value of the reference counter, keeping track of how many times the table has been fixed by the process.
  If the table never has been fixed, the call returns `false`.

`init_table(Name, InitFun) -> true`

Types:

- Name = atom()
- InitFun = fun(Arg) -> Res
- Arg = read | close
- Res = end_of_input | {[object()], InitFun} | term()

Replaces the existing objects of the table Tab with objects created by calling the input function InitFun, see below. This function is provided for compatibility with the dets module, it is not more efficient than filling a table by using ets:insert/2.

When called with the argument read the function InitFun is assumed to return end_of_input when there is no more input, or {Objects, Fun}, where Objects is a list of objects and Fun is a new input function. Any other value Value is returned as an error {error, {init_fun, Value}}. Each input function will be called exactly once, and should an error occur, the last function is called with the argument close, the reply of which is ignored.

If the type of the table is set and there is more than one object with a given key, one of the objects is chosen. This is not necessarily the last object with the given key in the sequence of objects returned by the input functions. This holds also for duplicated objects stored in tables of type duplicate_bag.

`insert(Tab, ObjectOrObjects) -> true`

Types:

- Tab = tid() | atom()
- ObjectOrObjects = tuple() | [tuple()]

Inserts the object or all of the objects in the list ObjectOrObjects into the table Tab. If the table is a set and the key of the inserted objects *matches* the key of any object in the table, the old object will be replaced. If the table is an ordered_set and the key of the inserted object *compares equal* to the key of any object in the table, the old object is also replaced. If the list contains more than one object with *matching* keys and the table is a set, one will be inserted, which one is not defined. The same thing holds for ordered_set, but will also happen if the keys *compare equal*.

`insert_new(Tab, ObjectOrObjects) -> bool()`

Types:

- Tab = tid() | atom()
- ObjectOrObjects = tuple() | [tuple()]

This function works exactly like insert/2, with the exception that instead of overwriting objects with the same key (in the case of set or ordered_set) or adding more objects with keys already existing in the table (in the case of bag and duplicate_bag), it simply returns false. If ObjectOrObjects is a list, the function checks *every* key prior to inserting anything. Nothing will be inserted if not *all* keys present in the list are absent from the table.

`is_compiled_ms(Term) -> bool()`

Types:

- Term = term()

This function is used to check if a term is a valid compiled match_spec [page 140]. The compiled match_spec is an opaque datatype which can *not* be sent between Erlang nodes nor be stored on disk. Any attempt to create an external representation of a compiled match_spec will result in an empty binary (<<>>). As an example, the following expression:

```
ets:is_compiled_ms(ets:match_spec_compile([{'_',[],[true]}])).
```

will yield true, while the following expressions:

```
MS = ets:match_spec_compile([{'_',[],[true]}]),
Broken = binary_to_term(term_to_binary(MS)),
ets:is_compiled_ms(Broken).
```

will yield false, as the variable `Broken` will contain a compiled match_spec that has passed through external representation.

> **Note:**
> The fact that compiled match_specs has no external representation is for performance reasons. It may be subject to change in future releases, while this interface will still remain for backward compatibility reasons.

`last(Tab) -> Key | '$end_of_table'`

Types:

- Tab = tid() | atom()
- Key = term()

Returns the last key `Key` according to Erlang term order in the table `Tab` of the `ordered_set` type. If the table is of any other type, the function is synonymous to `first/2`. If the table is empty, `'$end_of_table'` is returned.

Use `prev/2` to find preceding keys in the table.

`lookup(Tab, Key) -> [Object]`

Types:

- Tab = tid() | atom()
- Key = term()
- Object = tuple()

Returns a list of all objects with the key `Key` in the table `Tab`.

In the case of `set, bag and duplicate_bag`, an object is returned only if the given key *matches* the key of the object in the table. If the table is an `ordered_set` however, an object is returned if the key given *compares equal* to the key of an object in the table. The difference being the same as between `=:=` and `==`. As an example, one might insert an object with the `integer()1` as a key in an `ordered_set` and get the object returned as a result of doing a `lookup/2` with the `float()1.0` as the key to search for.

If the table is of type `set` or `ordered_set`, the function returns either the empty list or a list with one element, as there cannot be more than one object with the same key. If the table is of type `bag` or `duplicate_bag`, the function returns a list of arbitrary length.

Note that the time order of object insertions is preserved; The first object inserted with the given key will be first in the resulting list, and so on.

Insert and look-up times in tables of type `set`, `bag` and `duplicate_bag` are constant, regardless of the size of the table. For the `ordered_set` data-type, time is proportional to the (binary) logarithm of the number of objects.

`lookup_element(Tab, Key, Pos) -> Elem`

Types:

- Tab = tid() | atom()
- Key = term()
- Pos = int()
- Elem = term() | [term()]

If the table Tab is of type set or ordered_set, the function returns the Pos:th element of the object with the key Key.

If the table is of type bag or duplicate_bag, the functions returns a list with the Pos:th element of every object with the key Key.

If no object with the key Key exists, the function will exit with reason badarg.

The difference between set, bag and duplicate_bag on one hand, and ordered_set on the other, regarding the fact that ordered_set's view keys as equal when they *compare equal* whereas the other table types only regard them equal when they *match*, naturally holds for lookup_element as well as for lookup.

`match(Tab, Pattern) -> [Match]`

Types:

- Tab = tid() | atom()
- Pattern = tuple()
- Match = [term()]

Matches the objects in the table Tab against the pattern Pattern.

A pattern is a term that may contain:

- bound parts (Erlang terms),
- '_' which matches any Erlang term, and
- pattern variables: '$N' where N=0,1,...

The function returns a list with one element for each matching object, where each element is an ordered list of pattern variable bindings. An example:

```
6> ets:match(T, '$1'). % Matches every object in the table
[[{rufsen,dog,7}],[{brunte,horse,5}],[{ludde,dog,5}]]
7> ets:match(T, {'_',dog,'$1'}).
[[7],[5]]
8> ets:match(T, {'_',cow,'$1'}).
[]
```

If the key is specified in the pattern, the match is very efficient. If the key is not specified, i.e. if it is a variable or an underscore, the entire table must be searched. The search time can be substantial if the table is very large.

On tables of the ordered_set type, the result is in the same order as in a first/next traversal.

`match(Tab, Pattern, Limit) -> {[Match],Continuation} | '$end_of_table'`

Types:

- Tab = tid() | atom()
- Pattern = tuple()
- Match = [term()]
- Continuation = term()

Works like ets:match/2 but only returns a limited (Limit) number of matching objects. The Continuation term can then be used in subsequent calls to ets:match/1 to get the next chunk of matching objects. This is a space efficient way to work on objects in a table which is still faster than traversing the table object by object using ets:first/1 and ets:next/1.

'$end_of_table' is returned if the table is empty.

match(Continuation) -> {[Match],Continuation} | '$end_of_table'

Types:

- Match = [term()]
- Continuation = term()

Continues a match started with ets:match/3. The next chunk of the size given in the initial ets:match/3 call is returned together with a new Continuation that can be used in subsequent calls to this function.

'$end_of_table' is returned when there are no more objects in the table.

match_delete(Tab, Pattern) -> true

Types:

- Tab = tid() | atom()
- Pattern = tuple()

Deletes all objects which match the pattern Pattern from the table Tab. See match/2 for a description of patterns.

match_object(Tab, Pattern) -> [Object]

Types:

- Tab = tid() | atom()
- Pattern = Object = tuple()

Matches the objects in the table Tab against the pattern Pattern. See match/2 for a description of patterns. The function returns a list of all objects which match the pattern.

If the key is specified in the pattern, the match is very efficient. If the key is not specified, i.e. if it is a variable or an underscore, the entire table must be searched. The search time can be substantial if the table is very large.

On tables of the ordered_set type, the result is in the same order as in a first/next traversal.

match_object(Tab, Pattern, Limit) -> {[Match],Continuation} | '$end_of_table'

Types:

- Tab = tid() | atom()
- Pattern = tuple()
- Match = [term()]
- Continuation = term()

Works like ets:match_object/2 but only returns a limited (Limit) number of matching objects. The Continuation term can then be used in subsequent calls to ets:match_object/1 to get the next chunk of matching objects. This is a space efficient way to work on objects in a table which is still faster than traversing the table object by object using ets:first/1 and ets:next/1.

'$end_of_table' is returned if the table is empty.

match_object(Continuation) -> {[Match],Continuation} | '$end_of_table'

Types:

- Match = [term()]
- Continuation = term()

Continues a match started with ets:match_object/3. The next chunk of the size given in the initial ets:match_object/3 call is returned together with a new Continuation that can be used in subsequent calls to this function.

'$end_of_table' is returned when there are no more objects in the table.

match_spec_compile(MatchSpec) -> CompiledMatchSpec

Types:

- MatchSpec = match_spec()
- CompiledMatchSpec = comp_match_spec()

This function transforms a match_spec [page 140] into an internal representation that can be used in subsequent calls to ets:match_spec_run/2. The internal representation is opaque and can not be converted to external term format and then back again without losing its properties (meaning it can not be sent to a process on another node and still remain a valid compiled match_spec, nor can it be stored on disk). The validity of a compiled match_spec can be checked using ets:is_compiled_ms/1.

If the term MatchSpec can not be compiled (does not represent a valid match_spec), a badarg fault is thrown.

> **Note:**
> This function has limited use in normal code, it is used by Dets to perform the dets:select operations.

match_spec_run(List,CompiledMatchSpec) -> list()

Types:

- List = [tuple()]
- CompiledMatchSpec = comp_match_spec()

This function executes the matching specified in a compiled match_spec [page 140] on a list of tuples. The CompiledMatchSpec term should be the result of a call to ets:match_spec_compile/1 and is hence the internal representation of the match_spec one wants to use.

The matching will be executed on each element in List and the function returns a list containing all results. If an element in List does not match, nothing is returned for that element. The length of the result list is therefore equal or less than the the length of the parameter List. The two calls in the following example will give the same result (but certainly not the same execution time...):

```
Table = ets:new...
MatchSpec = ....
% The following call...
ets:match_spec_run(ets:tab2list(Table),
ets:match_spec_compile(MatchSpec)),
% ...will give the same result as the more common (and more efficient)
ets:select(Table,MatchSpec),
```

> **Note:**
> This function has limited use in normal code, it is used by Dets to perform the dets:select operations and by Mnesia during transactions.

`member(Tab, Key) -> true | false`

Types:

- Tab = tid() | atom()
- Key = term()

Works like lookup/2, but does not return the objects. The function returns true if one or more elements in the table has the key Key, false otherwise.

`new(Name, Options) -> tid()`

Types:

- Name = atom()
- Options = [Option]
- Option = Type | Access | named_table | {keypos,Pos}
- Type = set | ordered_set | bag | duplicate_bag
- Access = public | protected | private
- Pos = int()

Creates a new table and returns a table identifier which can be used in subsequent operations. The table identifier can be sent to other processes so that a table can be shared between different processes within a node.

The parameter Options is a list of atoms which specifies table type, access rights, key position and if the table is named or not. If one or more options are left out, the default values are used. This means that not specifying any options ([]) is the same as specifying [set,protected,{keypos,1}].

- `set` The table is a `set` table - one key, one object, no order among objects. This is the default table type.
- `ordered_set` The table is a `ordered_set` table - one key, one object, ordered in Erlang term order, which is the order implied by the < and > operators. Tables of this type have a somewhat different behavior in some situations than tables of the other types. Most notably the `ordered_set` tables regard keys as equal when they *compare equal*, not only when they match. This means that to an `ordered_set`, the `integer()1` and the `float()1.0` are regarded as equal. This also means that the key used to lookup an element not necessarily *matches* the key in the elements returned, if `float()`'s and `integer()`'s are mixed in keys of a table.
- `bag` The table is a `bag` table which can have many objects, but only one instance of each object, per key.
- `duplicate_bag` The table is a `duplicate_bag` table which can have many objects, including multiple copies of the same object, per key.
- `public` Any process may read or write to the table.
- `protected` The owner process can read and write to the table. Other processes can only read the table. This is the default setting for the access rights.
- `private` Only the owner process can read or write to the table.
- `named_table` If this option is present, the name `Name` is associated with the table identifier. The name can then be used instead of the table identifier in subsequent operations.
- {`keypos`,`Pos`} Specfies which element in the stored tuples should be used as key. By default, it is the first element, i.e. `Pos=1`. However, this is not always appropriate. In particular, we do not want the first element to be the key if we want to store Erlang records in a table.

  Note that any tuple stored in the table must have at least `Pos` number of elements.

`next(Tab, Key1) -> Key2 | '$end_of_table'`

Types:

- Tab = tid() | atom()
- Key1 = Key2 = term()

Returns the next key `Key2`, following the key `Key1` in the table `Tab`. If the table is of the `ordered_set` type, the next key in Erlang term order is returned. If the table is of any other type, the next key according to the table's internal order is returned. If there is no next key, `'$end_of_table'` is returned.

Use `first/1` to find the first key in the table.

Unless a table of type `set`, `bag` or `duplicate_bag` is protected using `safe_fixtable/2`, see below, a traversal may fail if concurrent updates are made to the table. If the table is of type `ordered_set`, the function returns the next key in order, even if the object does no longer exist.

`prev(Tab, Key1) -> Key2 | '$end_of_table'`

Types:

- Tab = tid() | atom()
- Key1 = Key2 = term()

Returns the previous key Key2, preceding the key Key1 according the Erlang term order in the table Tab of the ordered_set type. If the table is of any other type, the function is synonymous to next/2. If there is no previous key, '$end_of_table' is returned.

Use last/1 to find the last key in the table.

rename(Tab, Name) -> Name

Types:

- Tab = Name = atom()

Renames the named table Tab to the new name Name. Afterwards, the old name can not be used to access the table. Renaming an unnamed table has no effect.

repair_continuation(Continuation, MatchSpec) -> Continuation

Types:

- Continuation = term()
- MatchSpec = match_spec()

This function can be used to restore an opaque continuation returned by ets:select/3 or ets:select/1 if the continuation has passed through external term format (been sent between nodes or stored on disk).

The reason for this function is that continuation terms contain compiled match_specs and therefore will be invalidated if converted to external term format. Given that the original match_spec is kept intact, the continuation can be restored, meaning it can once again be used in subsequent ets:select/1 calls even though it has been stored on disk or on another node.

As an example, the following seqence of calls will fail:

```
T=ets:new(x,[]),
...
{_,C} = ets:select(T,ets:fun2ms(fun({N,_}=A)
when (N rem 10) =:= 0 ->
A
end),10),
Broken = binary_to_term(term_to_binary(C)),
ets:select(Broken).
```

...while the following sequence will work:

```
T=ets:new(x,[]),
...
MS = ets:fun2ms(fun({N,_}=A)
when (N rem 10) =:= 0 ->
A
end),
{_,C} = ets:select(T,MS,10),
Broken = binary_to_term(term_to_binary(C)),
ets:select(ets:repair_continuation(Broken,MS)).
```

...as the call to `ets:repair_continuation/2` will reestablish the (deliberately) invalidated continuation `Broken`.

> **Note:**
> This function is very rarely needed in application code. It is used by Mnesia to implement distributed `select/3` and `select/1` sequences. A normal application would either use Mnesia or keep the continuation from being converted to external format.
>
> The reason for not having an external representation of a compiled match_spec is performance. It may be subject to change in future releases, while this interface will remain for backward compatibility.

`safe_fixtable(Tab, true|false) -> true`

Types:

- Tab = tid() | atom()

Fixes a table of the `set`, `bag` or `duplicate_bag` table type for safe traversal.

A process fixes a table by calling `safe_fixtable(Tab,true)`. The table remains fixed until the process releases it by calling `safe_fixtable(Tab,false)`, or until the process terminates.

If several processes fix a table, the table will remain fixed until all processes have released it (or terminated). A reference counter is kept on a per process basis, and N consecutive fixes requires N releases to actually release the table.

When a table is fixed, a sequence of `first/1` and `next/2` calls are guaranteed to succeed and each object in the table will only be returned once, even if objects are removed or inserted during the traversal. The keys for new objects inserted during the traversal *may* be returned by next/2 [page 152] (it depends on the internal ordering of the keys). An example:

```
clean_all_with_value(Tab,X) ->
    safe_fixtable(Tab,true),
    clean_all_with_value(Tab,X,ets:first(Tab)),
    safe_fixtable(Tab,false).

clean_all_with_value(Tab,X,'$end_of_table') ->
    true;
clean_all_with_value(Tab,X,Key) ->
    case ets:lookup(Tab,Key) of
        [{Key,X}] ->
            ets:delete(Tab,Key);
        _ ->
            true
    end,
    clean_all_with_value(Tab,X,ets:next(Tab,Key)).
```

Note that no deleted objects are actually removed from a fixed table until it has been released. If a process fixes a table but never releases it, the memory used by the deleted objects will never be freed. The performance of operations on the table will also degrade significantly.

Use `info/2` to retrieve information about which processes have fixed which tables. A system with a lot of processes fixing tables may need a monitor which sends alarms when tables have been fixed for too long.

Note that for tables of the `ordered_set` type, `safe_fixtable/2` is not necessary as calls to `first/1` and `next/2` will always succeed.

`select(Tab, MatchSpec) -> [Match]`

Types:

- Tab = tid() | atom()
- Match = term()
- MatchSpec = match_spec()

Matches the objects in the table `Tab` using a match_spec [page 140]. This is a more general call than the `ets:match/2` and `ets:match_object/2` calls. In its simplest forms the match_specs look like this:

- MatchSpec = [MatchFunction]
- MatchFunction = {MatchHead, [Guard], [Result]}
- MatchHead = “Pattern as in ets:match”
- Guard = {“Guardtest name”, ...}
- Result = “Term construct”

This means that the match_spec is always a list of one or more tuples (of arity 3). The tuples first element should be a pattern as described in the documentation of `ets:match/2`. The second element of the tuple should be a list of 0 or more guard tests (described below). The third element of the tuple should be a list containing a description of the value to actually return. In almost all normal cases the list contains exactly one term which fully describes the value to return for each object.

The return value is constructed using the “match variables” bound in the MatchHead or using the special match variables `'$_'` (the whole matching object) and `'$$'` (all match variables in a list), so that the following `ets:match/2` expression:

```
ets:match(Tab,{'$1','$2','$3'})
```

is exactly equivalent to:

```
ets:select(Tab,[{{'$1','$2','$3'},[],['$$']}])
```

- and the following `ets:match_object/2` call:

```
ets:match_object(Tab,{'$1','$2','$1'})
```

is exactly equivalent to

```
ets:select(Tab,[{{'$1','$2','$1'},[],['$_']}])
```

Composite terms can be constructed in the `Result` part either by simply writing a list, so that this code:

```
ets:select(Tab,[{{'$1','$2','$3'},[],['$$']}])
```

gives the same output as:

```
ets:select(Tab,[{{'$1','$2','$3'},[],[['$1','$2','$3']]}])
```

i.e. all the bound variables in the match head as a list. If tuples are to be constructed, one has to write a tuple of arity 1 with the single element in the tuple being the tuple one wants to construct (as an ordinary tuple could be mistaken for a `Guard`). Therefore the following call:

```
ets:select(Tab,[{{'$1','$2','$1'},[],['$_']}])
```

gives the same output as:

```
ets:select(Tab,[{{'$1','$2','$1'},[],[{{'$1','$2','$3'}}]}])
```

- this syntax is equivalent to the syntax used in the trace patterns (see [dbg(3)]).

The `Guards` are constructed as tuples where the first element is the name of the test and the rest of the elements are the parameters of the test. To check for a specific type (say a list) of the element bound to the match variable `'$1'`, one would write the test as `{is_list, '$1'}`. If the test fails, the object in the table will not match and the next `MatchFunction` (if any) will be tried. Most guard tests present in Erlang can be used, but only the new versions prefixed `is_` are allowed (like `is_float`, `is_atom` etc).

The `Guard` section can also contain logic and arithmetic operations, which are written with the same syntax as the guard tests (prefix notation), so that a guard test written in Erlang looking like this:

```
is_integer(X), is_integer(Y), X + Y < 4711
```

is expressed like this (X replaced with '$1' and Y with '$2'):

```
[{is_integer, '$1'}, {is_integer, '$2'}, {'<', {'+', '$1', '$2'}, 4711}]
```

`select(Tab, MatchSpec, Limit) -> {[Match],Continuation} | '$end_of_table'`

Types:

- Tab = tid() | atom()
- Match = term()
- MatchSpec = match_spec()
- Continuation = term()

Works like `ets:select/2` but only returns a limited (`Limit`) number of matching objects. The `Continuation` term can then be used in subsequent calls to `ets:select/1` to get the next chunk of matching objects. This is a space efficient way to work on objects in a table which is still faster than traversing the table object by object using `ets:first/1` and `ets:next/1`.

`'$end_of_table'` is returned if the table is empty.

`select(Continuation) -> {[Match],Continuation} | '$end_of_table'`

Types:

- Match = term()
- Continuation = term()

Continues a match started with `ets:select/3`. The next chunk of the size given in the initial `ets:select/3` call is returned together with a new `Continuation` that can be used in subsequent calls to this function.

`'$end_of_table'` is returned when there are no more objects in the table.

`select_delete(Tab, MatchSpec) -> NumDeleted`

Types:

- Tab = tid() | atom()
- Object = tuple()
- MatchSpec = match_spec()
- NumDeleted = integer()

Matches the objects in the table `Tab` using a match_spec [page 140]. If the match_spec returns `true` for an object, that object is removed from the table. For any other result from the match_spec the object is retained. This is a more general call than the `ets:match_delete/2` call.

The function returns the number of objects actually deleted from the table.

> **Note:**
> The `match_spec` has to return the atom `true` if the object is to be deleted. No other return value will get the object deleted, why one can not use the same match specification for looking up elements as for deleting them.

`select_count(Tab, MatchSpec) -> NumMatched`

Types:

- Tab = tid() | atom()
- Object = tuple()
- MatchSpec = match_spec()
- NumMatched = integer()

Matches the objects in the table `Tab` using a match_spec [page 140]. If the match_spec returns `true` for an object, that object considered a match and is counted. For any other result from the match_spec the object is not considered a match and is therefore not counted.

The function could be described as a `match_delete/2` that does not actually delete any elements, but only counts them.

The function returns the number of objects matched.

`slot(Tab, I) -> [Object] | '$end_of_table'`

Types:

- Tab = tid() | atom()
- I = int()
- Object = tuple()

This function is mostly for debugging purposes, Normally one should use `first/next` or `last/prev` instead.

Returns all objects in the `I`:th slot of the table `Tab`. A table can be traversed by repeatedly calling the function, starting with the first slot `I=0` and ending when `'$end_of_table'` is returned. The function will fail with reason `badarg` if the `I` argument is out of range.

Unless a table of type `set`, `bag` or `duplicate_bag` is protected using `safe_fixtable/2`, see above, a traversal may fail if concurrent updates are made to the table. If the table is of type `ordered_set`, the function returns a list containing the `I`:th object in Erlang term order.

`tab2file(Tab, Filename) -> ok | {error,Reason}`

Types:

- Tab = tid() | atom()
- Filename = string() | atom()
- Reason = term()

Dumps the table `Tab` to the file `Filename`.

Equivalent to `tab2file(Tab, Filename,[])`

`tab2file(Tab, Filename, Options) -> ok | {error,Reason}`

Types:

- Tab = tid() | atom()
- Filename = string() | atom()
- Options = [Option]
- Option = {extended_info, [ExtInfo]}
- ExtInfo = object_count | md5sum
- Reason = term()

Dumps the table `Tab` to the file `Filename`.

When dumping the table, certain information about the table is dumped to a header at the beginning of the dump. This information contains data about the table type, name, protection, size, version and if it's a named table. It also contains notes about what extended information is added to the file, which can be a count of the objects in the file or a MD5 sum of the header and records in the file.

The size field in the header might not correspond to the actual number of records in the file if the table is public and records are added or removed from the table during dumping. Public tables updated during dump, and that one wants to verify when reading, needs at least one field of extended information for the read verification process to be reliable later.

The `extended_info` option specifies what extra information is written to the table dump:

`object_count` The number of objects actually written to the file is noted in the file footer, why verification of file truncation is possible even if the file was updated during dump.

`md5sum` The header and objects in the file are checksummed using the built in MD5 functions. The MD5 sum of all objects is written in the file footer, so that verification while reading will detect the slightest bitflip in the file data. Using this costs a fair amount of CPU time.

Whenever the `extended_info` option is used, it results in a file not readable by versions of ets prior to that in stdlib-1.15.1

`tab2list(Tab) -> [Object]`

Types:

- Tab = tid() | atom()
- Object = tuple()

Returns a list of all objects in the table `Tab`.

`tabfile_info(Filename) -> {ok, TableInfo} | {error, Reason}`

Types:

- Filename = string() | atom()
- TableInfo = [InfoItem]
- InfoItem = {InfoTag, term()}
- InfoTag = name | type | protection | named_table | keypos | size | extended_info | version
- Reason = term()

Returns information about the table dumped to file by tab2file/2 [page 158] or tab2file/3 [page 158]

The following items are returned:

name The name of the dumped table. If the table was a named table, a table with the same name cannot exist when the table is loaded from file with file2tab/2 [page 141]. If the table is not saved as a named table, this field has no significance at all when loading the table from file.

type The ets type of the dumped table (i.e. `set`, `bag`, `duplicate_bag` or `ordered_set`). This type will be used when loading the table again.

protection The protection of the dumped table (i.e. `private`, `protected` or `public`). A table loaded from the file will get the same protection.

named_table `true` if the table was a named table when dumped to file, otherwise `false`. Note that when a named table is loaded from a file, there cannot exist a table in the system with the same name.

keypos The `keypos` of the table dumped to file, which will be used when loading the table again.

size The number of objects in the table when the table dump to file started, which in case of a `public` table need not correspond to the number of objects actually saved to the file, as objects might have been added or deleted by another process during table dump.

extended_info The extended information written in the file footer to allow stronger verification during table loading from file, as specified to tab2file/3 [page 158]. Note that this function only tells *which* information is present, not the values in the file footer. The value is a list containing one or more of the atoms `object_count` and `md5sum`.

version A tuple {Major,Minor} containing the major and minor version of the file format for ets table dumps. This version field was added beginning with stdlib-1.5.1, files dumped with older versions will return {0,0} in this field.

An error is returned if the file is inaccessible, badly damaged or not an file produced with tab2file/2 [page 158] or tab2file/3 [page 158].

`table(Tab [, Options]) -> QueryHandle`

Types:

- Tab = tid() | atom()
- QueryHandle = -a query handle, see qlc(3)-
- Options = [Option] | Option
- Option = {n_objects, NObjects} | {traverse, TraverseMethod}
- NObjects = default | integer() > 0
- TraverseMethod = first_next | last_prev | select | {select, MatchSpec}
- MatchSpec = match_spec()

Returns a QLC (Query List Comprehension) query handle. The module `qlc` implements a query language aimed mainly at Mnesia but ETS tables, Dets tables, and lists are also recognized by QLC as sources of data. Calling `ets:table/1,2` is the means to make the ETS table `Tab` usable to QLC.

When there are only simple restrictions on the key position QLC uses `ets:lookup/2` to look up the keys, but when that is not possible the whole table is traversed. The option `traverse` determines how this is done:

- `first_next`. The table is traversed one key at a time by calling `ets:first/1` and `ets:next/2`.
- `last_prev`. The table is traversed one key at a time by calling `ets:last/1` and `ets:prev/2`.
- `select`. The table is traversed by calling `ets:select/3` and `ets:select/1`. The option `n_objects` determines the number of objects returned (the third argument of `select/3`); the default is to return 100 objects at a time. The match_spec [page 140] (the second argument of `select/3`) is assembled by QLC: simple filters are translated into equivalent match_specs while more complicated filters have to be applied to all objects returned by `select/3` given a match_spec that matches all objects.
- {`select, MatchSpec`}. As for `select` the table is traversed by calling `ets:select/3` and `ets:select/1`. The difference is that the match_spec is explicitly given. This is how to state match_specs that cannot easily be expressed within the syntax provided by QLC.

The following example uses an explicit match_spec to traverse the table:

```
9> ets:insert(Tab = ets:new(t, []), [{1,a},{2,b},{3,c},{4,d}]),
MS = ets:fun2ms(fun({X,Y}) when (X > 1) or (X < 5) -> {Y} end),
QH1 = ets:table(Tab, [{traverse, {select, MS}}]).
```

An example with implicit match_spec:

```
10> QH2 = qlc:q([{Y} || {X,Y} <- ets:table(Tab), (X > 1) or (X < 5)]).
```

The latter example is in fact equivalent to the former which can be verified using the function `qlc:info/1`:

```
11> qlc:info(QH1) =:= qlc:info(QH2).
true
```

`qlc:info/1` returns information about a query handle, and in this case identical information is returned for the two query handles.

`test_ms(Tuple, MatchSpec) -> {ok, Result} | {error, Errors}`

Types:

- Tuple = tuple()
- MatchSpec = match_spec()
- Result = term()
- Errors = [{warning|error, string()}]

This function is a utility to test a match_spec [page 140] used in calls to `ets:select/2`. The function both tests `MatchSpec` for "syntactic" correctness and runs the match_spec against the object `Tuple`. If the match_spec contains errors, the tuple `{error, Errors}` is returned where `Errors` is a list of natural language descriptions of what was wrong with the match_spec. If the match_spec is syntactically OK, the function returns `{ok,Term}` where `Term` is what would have been the result in a real `ets:select/2` call or `false` if the match_spec does not match the object `Tuple`.

This is a useful debugging and test tool, especially when writing complicated `ets:select/2` calls.

`to_dets(Tab, DetsTab) -> Tab`

Types:

- Tab = tid() | atom()
- DetsTab = atom()

Fills an already created/opened Dets table with the objects in the already opened ETS table named `Tab`. The Dets table is emptied before the objects are inserted.

```
update_counter(Tab, Key, UpdateOp) -> Result
update_counter(Tab, Key, [UpdateOp]) -> [Result]
update_counter(Tab, Key, Incr) -> Result
```

Types:

- Tab = tid() | atom()
- Key = term()
- UpdateOp = {Pos,Incr} | {Pos,Incr,Threshold,SetValue}
- Pos = Incr = Threshold = SetValue = Result = int()

This function provides an efficient way to update one or more counters, without the hassle of having to look up an object, update the object by incrementing an element and insert the resulting object into the table again. (The update is done atomically; i.e. no process can access the ets table in the middle of the operation.)

It will destructively update the object with key `Key` in the table `Tab` by adding `Incr` to the element at the `Pos`:th position. The new counter value is returned. If no position is specified, the element directly following the key (`<keypos>+1`) is updated.

If a `Threshold` is specified, the counter will be reset to the value `SetValue` if the following conditions occur:

- The `Incr` is not negative (`>= 0`) and the result would be greater than (`>`) `Threshold`
- The `Incr` is negative (`< 0`) and the result would be less than (`<`) `Threshold`

A list of `UpdateOp` can be supplied to do several update operations within the object. The operations are carried out in the order specified in the list. If the same counter position occurs more than one time in the list, the corresponding counter will thus be updated several times, each time based on the previous result. The return value is a list of the new counter values from each update operation in the same order as in the operation list. If an empty list is specified, nothing is updated and an empty list is returned. If the function should fail, no updates will be done at all.

The given Key is used to identify the object by either *matching* the key of an object in a `set` table, or *compare equal* to the key of an object in an `ordered_set` table (see lookup/2 [page 147] and new/2 [page 151] for details on the difference).

The function will fail with reason `badarg` if:

- the table is not of type `set` or `ordered_set`,
- no object with the right key exists,
- the object has the wrong arity,
- the element to update is not an integer,
- the element to update is also the key, or,
- any of `Pos`, `Incr`, `Threshold` or `SetValue` is not an integer

```
update_element(Tab, Key, {Pos,Value}) -> true | false
update_element(Tab, Key, [{Pos,Value}]) -> true | false
```

Types:

- Tab = tid() | atom()
- Key = Value = term()
- Pos = int()

This function provides an efficient way to update one or more elements within an object, without the hassle of having to look up, update and write back the entire object.

It will destructively update the object with key `Key` in the table `Tab`. The element at the `Pos`:th position will be given the value `Value`.

A list of `{Pos,Value}` can be supplied to update several elements within the same object. If the same position occurs more than one in the list, the last value in the list will be written. If the list is empty or the function fails, no updates will be done at all. The function is also atomic in the sense that other processes can never see any intermediate results.

The function returns `true` if an object with the key `Key` was found, `false` otherwise.

The given Key is used to identify the object by either *matching* the key of an object in a `set` table, or *compare equal* to the key of an object in an `ordered_set` table (see lookup/2 [page 147] and new/2 [page 151] for details on the difference).

The function will fail with reason `badarg` if:

- the table is not of type `set` or `ordered_set`,
- `Pos` is less than 1 or greater than the object arity, or,
- the element to update is also the key

file_sorter

Erlang Module

The functions of this module sort terms on files, merge already sorted files, and check files for sortedness. Chunks containing binary terms are read from a sequence of files, sorted internally in memory and written on temporary files, which are merged producing one sorted file as output. Merging is provided as an optimization; it is faster when the files are already sorted, but it always works to sort instead of merge.

On a file, a term is represented by a header and a binary. Two options define the format of terms on files:

- {`header, HeaderLength`}. HeaderLength determines the number of bytes preceding each binary and containing the length of the binary in bytes. Default is 4. The order of the header bytes is defined as follows: if `B` is a binary containing a header only, the size `Size` of the binary is calculated as `<<Size:HeaderLength/unit:8>> = B`.
- {`format, Format`}. The format determines the function that is applied to binaries in order to create the terms that will be sorted. The default value is `binary_term`, which is equivalent to `funbinary_to_term/1`. The value `binary` is equivalent to `fun(X) -> X end`, which means that the binaries will be sorted as they are. This is the fastest format. If `Format` is `term`, `io:read/2` is called to read terms. In that case only the default value of the `header` option is allowed. The `format` option also determines what is written to the sorted output file: if `Format` is `term` then `io:format/3` is called to write each term, otherwise the binary prefixed by a header is written. Note that the binary written is the same binary that was read; the results of applying the `Format` function are thrown away as soon as the terms have been sorted. Reading and writing terms using the `io` module is very much slower than reading and writing binaries.

Other options are:

- {`order, Order`}. The default is to sort terms in ascending order, but that can be changed by the value `descending` or by giving an ordering function `Fun`. `Fun(A,B)` should return `true` if `A` comes before `B` in the ordering, `false` otherwise. Using an ordering function will slow down the sort considerably. The `keysort`, `keymerge` and `keycheck` functions do not accept ordering functions.
- {`unique, bool()`}. When sorting or merging files, only the first of a sequence of terms that compare equal is output if this option is set to `true`. The default value is `false` which implies that all terms that compare equal are output. When checking files for sortedness, a check that no pair of consecutive terms compares equal is done if this option is set to `true`.

- {tmpdir, TempDirectory}. The directory where temporary files are put can be chosen explicitly. The default, implied by the value "", is to put temporary files on the same directory as the sorted output file. If output is a function (see below), the directory returned by file:get_cwd() is used instead. The names of temporary files are derived from the Erlang nodename (node()), the process identifier of the current Erlang emulator (os:getpid()), and a timestamp (erlang:now()); a typical name would be fs_mynode@myhost_1763_1043_337000_266005.17, where 17 is a sequence number. Existing files will be overwritten. Temporary files are deleted unless some uncaught EXIT signal occurs.
- {compressed, bool()}. Temporary files and the output file may be compressed. The default value false implies that written files are not compressed. Regardless of the value of the compressed option, compressed files can always be read. Note that reading and writing compressed files is significantly slower than reading and writing uncompressed files.
- {size, Size}. By default approximately 512*1024 bytes read from files are sorted internally. This option should rarely be needed.
- {no_files, NoFiles}. By default 16 files are merged at a time. This option should rarely be needed.

To summarize, here is the syntax of the options:

- Options = [Option] | Option
- Option = {header, HeaderLength} | {format, Format} | {order, Order} | {unique, bool()} | {tmpdir, TempDirectory} | {compressed, bool()} | {size, Size} | {no_files, NoFiles}
- HeaderLength = int() > 0
- Format = binary_term | term | binary | FormatFun
- FormatFun = fun(Binary) -> Term
- Order = ascending | descending | OrderFun
- OrderFun = fun(Term, Term) -> bool()
- TempDirectory = "" | file_name()
- Size = int() >= 0
- NoFiles = int() > 1

As an alternative to sorting files, a function of one argument can be given as input. When called with the argument read the function is assumed to return end_of_input or {end_of_input, Value}} when there is no more input (Value is explained below), or {Objects, Fun}, where Objects is a list of binaries or terms depending on the format and Fun is a new input function. Any other value is immediately returned as value of the current call to sort or keysort. Each input function will be called exactly once, and should an error occur, the last function is called with the argument close, the reply of which is ignored.

A function of one argument can be given as output. The results of sorting or merging the input is collected in a non-empty sequence of variable length lists of binaries or terms depending on the format. The output function is called with one list at a time, and is assumed to return a new output function. Any other return value is immediately returned as value of the current call to the sort or merge function. Each output function is called exactly once. When some output function has been applied to all of the results or an error occurs, the last function is called with the argument close, and the reply is

returned as value of the current call to the sort or merge function. If a function is given as input and the last input function returns {end_of_input, Value}, the function given as output will be called with the argument {value, Value}. This makes it easy to initiate the sequence of output functions with a value calculated by the input functions.

As an example, consider sorting the terms on a disk log file. A function that reads chunks from the disk log and returns a list of binaries is used as input. The results are collected in a list of terms.

```
sort(Log) ->
    {ok, _} = disk_log:open([{name,Log}, {mode,read_only}]),
    Input = input(Log, start),
    Output = output([]),
    Reply = file_sorter:sort(Input, Output, {format,term}),
    ok = disk_log:close(Log),
    Reply.

input(Log, Cont) ->
    fun(close) ->
            ok;
       (read) ->
            case disk_log:chunk(Log, Cont) of
                {error, Reason} ->
                    {error, Reason};
                {Cont2, Terms} ->
                    {Terms, input(Log, Cont2)};
                {Cont2, Terms, _Badbytes} ->
                    {Terms, input(Log, Cont2)};
                eof ->
                    end_of_input
            end
    end.

output(L) ->
    fun(close) ->
            lists:append(lists:reverse(L));
       (Terms) ->
            output([Terms | L])
    end.
```

Further examples of functions as input and output can be found at the end of the file_sorter module; the term format is implemented with functions.

The possible values of Reason returned when an error occurs are:

- bad_object, {bad_object, FileName}. Applying the format function failed for some binary, or the key(s) could not be extracted from some term.
- {bad_term, FileName}. io:read/2 failed to read some term.
- {file_error, FileName, Reason2}. See file(3) for an explanation of Reason2.
- {premature_eof, FileName}. End-of-file was encountered inside some binary term.

Types

```
Binary = binary()
FileName = file_name()
FileNames = [FileName]
ICommand = read | close
IReply = end_of_input | {end_of_input, Value} | {[Object], Infun} | InputReply
Infun = fun(ICommand) -> IReply
Input = FileNames | Infun
InputReply = Term
KeyPos = int() > 0 | [int() > 0]
OCommand = {value, Value} | [Object] | close
OReply = Outfun | OutputReply
Object = Term | Binary
Outfun = fun(OCommand) -> OReply
Output = FileName | Outfun
OutputReply = Term
Term = term()
Value = Term
```

Exports

`sort(FileName) -> Reply`
`sort(Input, Output) -> Reply`
`sort(Input, Output, Options) -> Reply`

Types:

- Reply = ok | {error, Reason} | InputReply | OutputReply

Sorts terms on files.

`sort(FileName)` is equivalent to `sort([FileName], FileName)`.

`sort(Input, Output)` is equivalent to `sort(Input, Output, [])`.

`keysort(KeyPos, FileName) -> Reply`
`keysort(KeyPos, Input, Output) -> Reply`
`keysort(KeyPos, Input, Output, Options) -> Reply`

Types:

- Reply = ok | {error, Reason} | InputReply | OutputReply

Sorts tuples on files. The sort is performed on the element(s) mentioned in `KeyPos`. If two tuples compare equal on one element, next element according to `KeyPos` is compared. The sort is stable.

`keysort(N, FileName)` is equivalent to `keysort(N, [FileName], FileName)`.

`keysort(N, Input, Output)` is equivalent to `keysort(N, Input, Output, [])`.

`merge(FileNames, Output) -> Reply`
`merge(FileNames, Output, Options) -> Reply`

Types:

- Reply = ok | {error, Reason} | OutputReply

Merges terms on files. Each input file is assumed to be sorted.

`merge(FileNames, Output)` is equivalent to `merge(FileNames, Output, [])`.

```
keymerge(KeyPos, FileNames, Output) -> Reply
keymerge(KeyPos, FileNames, Output, Options) -> Reply
```

Types:

- Reply = ok | {error, Reason} | OutputReply

Merges tuples on files. Each input file is assumed to be sorted on key(s).

`keymerge(KeyPos, FileNames, Output)` is equivalent to `keymerge(KeyPos, FileNames, Output, [])`.

```
check(FileName) -> Reply
check(FileNames, Options) -> Reply
```

Types:

- Reply = {ok, [Result]} | {error, Reason}
- Result = {FileName, TermPosition, Term}
- TermPosition = int() > 1

Checks files for sortedness. If a file is not sorted, the first out-of-order element is returned. The first term on a file has position 1.

`check(FileName)` is equivalent to `check([FileName], [])`.

```
keycheck(KeyPos, FileName) -> CheckReply
keycheck(KeyPos, FileNames, Options) -> Reply
```

Types:

- Reply = {ok, [Result]} | {error, Reason}
- Result = {FileName, TermPosition, Term}
- TermPosition = int() > 1

Checks files for sortedness. If a file is not sorted, the first out-of-order element is returned. The first term on a file has position 1.

`keycheck(KeyPos, FileName)` is equivalent to `keycheck(KeyPos, [FileName], [])`.

filelib

Erlang Module

This module contains utilities on a higher level than the `file` module.

DATA TYPES

```
filename() = string() | atom() | DeepList
dirname() = filename()
DeepList = [char() | atom() | DeepList]
```

Exports

`ensure_dir(Name) -> ok | {error, Reason}`

Types:

- Name = filename() | dirname()
- Reason = posix() – see file(3)

The `ensure_dir/1` function ensures that all parent directories for the given file or directory name `Name` exist, trying to create them if necessary.

Returns `ok` if all parent directories already exist or could be created, or `{error, Reason}` if some parent directory does not exist and could not be created for some reason.

`file_size(Filename) -> integer()`

The `file_size` function returns the size of the given file.

`fold_files(Dir, RegExp, Recursive, Fun, AccIn) -> AccOut`

Types:

- Dir = dirname()
- RegExp = regexp()
- Recursive = true|false
- Fun = fun(F, AccIn) -> AccOut
- AccIn = AccOut = term()

The `fold_files/5` function folds the function `Fun` over all (regular) files `F` in the directory `Dir` that match the regular expression `RegExp`. If `Recursive` is true all sub-directories to `Dir` are processed. The match is tried on just the filename without the directory part.

`is_dir(Name) -> true | false`

Types:

- Name = filename() | dirname()

The `is_dir/1` function returns `true` if `Name` refers to a directory, and `false` otherwise.

`is_file(Name) -> true | false`

Types:

- Name = filename() | dirname()

The `is_file/1` function returns `true` if `Name` refers to a file or a directory, and `false` otherwise.

`is_regular(Name) -> true | false`

Types:

- Name = filename()

The `is_regular/1` function returns `true` if `Name` refers to a file (regular file), and `false` otherwise.

`last_modified(Name) -> {{Year,Month,Day},{Hour,Min,Sec}}`

Types:

- Name = filename() | dirname()

The `last_modified/1` function returns the date and time the given file or directory was last modified.

`wildcard(Wildcard) -> list()`

Types:

- Wildcard = filename() | dirname()

The `wildcard/1` function returns a list of all files that match Unix-style wildcard-string `Wildcard`.

The wildcard string looks like an ordinary filename, except that certain "wildcard characters" are interpreted in a special way. The following characters are special:

? Matches one character.

* Matches any number of characters up to the end of the filename, the next dot, or the next slash.

{Item,...} Alternation. Matches one of the alternatives.

Other characters represent themselves. Only filenames that have exactly the same character in the same position will match. (Matching is case-sensitive; i.e. "a" will not match "A").

Note that multiple "*" characters are allowed (as in Unix wildcards, but opposed to Windows/DOS wildcards).

Examples:

The following examples assume that the current directory is the top of an Erlang/OTP installation.

To find all `.beam` files in all applications, the following line can be used:

```
filelib:wildcard("lib/*/ebin/*.beam").
```

To find either `.erl` or `.hrl` in all applications `src` directories, the following

```
filelib:wildcard("lib/*/src/*.?rl")
```

or the following line

```
filelib:wildcard("lib/*/src/*.{erl,hrl}")
```

can be used.

To find all `.hrl` files in either `src` or `include` directories, use:

```
filelib:wildcard("lib/*/{src,include}/*.hrl").
```

To find all `.erl` or `.hrl` files in either `src` or `include` directories, use:

```
filelib:wildcard("lib/*/{src,include}/*.{erl,hrl}")
```

`wildcard(Wildcard, Cwd) -> list()`

Types:

- Wildcard = filename() | dirname()
- Cwd = dirname()

The `wildcard/2` function works like `wildcard/1`, except that instead of the actual working directory, `Cwd` will be used.

filename

Erlang Module

The module `filename` provides a number of useful functions for analyzing and manipulating file names. These functions are designed so that the Erlang code can work on many different platforms with different formats for file names. With file name is meant all strings that can be used to denote a file. They can be short relative names like `foo.erl`, very long absolute name which include a drive designator and directory names like `D:\usr/local\bin\erl/lib\tools\foo.erl`, or any variations in between.

In Windows, all functions return file names with forward slashes only, even if the arguments contain back slashes. Use `join/1` to normalize a file name by removing redundant directory separators.

DATA TYPES

```
name() = string() | atom() | DeepList
  DeepList = [char() | atom() | DeepList]
```

Exports

`absname(Filename) -> string()`

Types:

- Filename = name()

Converts a relative `Filename` and returns an absolute name. No attempt is made to create the shortest absolute name, because this can give incorrect results on file systems which allow links.

Unix examples:

```
1> pwd().
"/usr/local"
2> filename:absname("foo").
"/usr/local/foo"
3> filename:absname("../x").
"/usr/local/../x"
4> filename:absname("/").
"/"
```

Windows examples:

```
1> pwd().
"D:/usr/local"
2> filename:absname("foo").
"D:/usr/local/foo"
3> filename:absname("../x").
"D:/usr/local/../x"
4> filename:absname("/").
"D:/"
```

`absname(Filename, Dir) -> string()`

Types:

- Filename = name()
- Dir = string()

This function works like `absname/1`, except that the directory to which the file name should be made relative is given explicitly in the `Dir` argument.

`absname_join(Dir, Filename) -> string()`

Types:

- Dir = string()
- Filename = name()

Joins an absolute directory with a relative filename. Similar to `join/2`, but on platforms with tight restrictions on raw filename length and no support for symbolic links (read: VxWorks), leading parent directory components in `Filename` are matched against trailing directory components in `Dir` so they can be removed from the result - minimizing its length.

`basename(Filename) -> string()`

Types:

- Filename = name()

Returns the last component of `Filename`, or `Filename` itself if it does not contain any directory separators.

```
5> filename:basename("foo").
"foo"
6> filename:basename("/usr/foo").
"foo"
7> filename:basename("/").
[]
```

`basename(Filename, Ext) -> string()`

Types:

- Filename = Ext = name()

Returns the last component of `Filename` with the extension `Ext` stripped. This function should be used to remove a specific extension which might, or might not, be there. Use `rootname(basename(Filename))` to remove an extension that exists, but you are not sure which one it is.

```
8> filename:basename("~/src/kalle.erl", ".erl").
"kalle"
9> filename:basename("~/src/kalle.beam", ".erl").
"kalle.beam"
10> filename:basename("~/src/kalle.old.erl", ".erl").
"kalle.old"
11> filename:rootname(filename:basename("~/src/kalle.erl")).
"kalle"
12> filename:rootname(filename:basename("~/src/kalle.beam")).
"kalle"
```

`dirname(Filename) -> string()`

Types:

- Filename = name()

Returns the directory part of `Filename`.

```
13> filename:dirname("/usr/src/kalle.erl").
"/usr/src"
14> filename:dirname("kalle.erl").
"."

5> filename:dirname("\\usr\\src/kalle.erl"). % Windows
"/usr/src"
```

`extension(Filename) -> string()`

Types:

- Filename = name()

Returns the file extension of `Filename`, including the period. Returns an empty string if there is no extension.

```
15> filename:extension("foo.erl").
".erl"
16> filename:extension("beam.src/kalle").
[]
```

`flatten(Filename) -> string()`

Types:

- Filename = name()

Converts a possibly deep list filename consisting of characters and atoms into the corresponding flat string filename.

`join(Components) -> string()`

Types:

- Components = [string()]

Joins a list of file name `Components` with directory separators. If one of the elements of `Components` includes an absolute path, for example `"/xxx"`, the preceding elements, if any, are removed from the result.

The result is “normalized”:

- Redundant directory separators are removed.
- In Windows, all directory separators are forward slashes and the drive letter is in lower case.

```
17> filename:join(["/usr", "local", "bin"]).
"/usr/local/bin"
18> filename:join(["a/b///c/"]).
"a/b/c"

6> filename:join(["B:a\\b///c/"]). % Windows
"b:a/b/c"
```

`join(Name1, Name2) -> string()`

Types:

- Name1 = Name2 = string()

Joins two file name components with directory separators. Equivalent to `join([Name1, Name2])`.

`nativename(Path) -> string()`

Types:

- Path = string()

Converts `Path` to a form accepted by the command shell and native applications on the current platform. On Windows, forward slashes is converted to backward slashes. On all platforms, the name is normalized as done by `join/1`.

```
19> filename:nativename("/usr/local/bin/"). % Unix
"/usr/local/bin"

7> filename:nativename("/usr/local/bin/"). % Windows
"\\usr\\local\\bin"
```

`pathtype(Path) -> absolute | relative | volumerelative`

Returns the type of path, one of `absolute`, `relative`, or `volumerelative`.

`absolute` The path name refers to a specific file on a specific volume.

Unix example: `/usr/local/bin`

Windows example: `D:/usr/local/bin`

`relative` The path name is relative to the current working directory on the current volume.

Example: `foo/bar, ../src`

`volumerelative` The path name is relative to the current working directory on a specified volume, or it is a specific file on the current working volume.

Windows example: `D:bar.erl, /bar/foo.erl`

`rootname(Filename) -> string()`
`rootname(Filename, Ext) -> string()`

Types:

- Filename = Ext = name()

Remove a filename extension. `rootname/2` works as `rootname/1`, except that the extension is removed only if it is `Ext`.

```
20> filename:rootname("/beam.src/kalle").
/beam.src/kalle"
21> filename:rootname("/beam.src/foo.erl").
"/beam.src/foo"
22> filename:rootname("/beam.src/foo.erl", ".erl").
"/beam.src/foo"
23> filename:rootname("/beam.src/foo.beam", ".erl").
"/beam.src/foo.beam"
```

`split(Filename) -> Components`

Types:

- Filename = name()
- Components = [string()]

Returns a list whose elements are the path components of `Filename`.

```
24> filename:split("/usr/local/bin").
["/","usr","local","bin"]
25> filename:split("foo/bar").
["foo","bar"]
26> filename:split("a:\\msdev\\include").
["a:/","msdev","include"]
```

`find_src(Beam) -> {SourceFile, Options} | {error,{ErrorReason,Module}}`
`find_src(Beam, Rules) -> {SourceFile, Options} | {error,{ErrorReason,Module}}`

Types:

- Beam = Module | Filename
- Module = atom()
- Filename = string() | atom()
- SourceFile = string()
- Options = [Opt]
- Opt = {i, string()} | {outdir, string()} | {d, atom()}
- ErrorReason = non_existing | preloaded | interpreted

Finds the source filename and compiler options for a module. The result can be fed to `compile:file/2` in order to compile the file again.

The `Beam` argument, which can be a string or an atom, specifies either the module name or the path to the source code, with or without the `".erl"` extension. In either case, the module must be known by the code server, i.e. `code:which(Module)` must succeed.

`Rules` describes how the source directory can be found, when the object code directory is known. It is a list of tuples {`BinSuffix`, `SourceSuffix`} and is interpreted as follows: If the end of the directory name where the object is located matches `BinSuffix`, then the source code directory has the same name, but with `BinSuffix` replaced by `SourceSuffix`. `Rules` defaults to:

```
[{"", ""}, {"ebin", "src"}, {"ebin", "esrc"}]
```

If the source file is found in the resulting directory, then the function returns that location together with `Options`. Otherwise, the next rule is tried, and so on.

The function returns {`SourceFile, Options`} if it succeeds. `SourceFile` is the absolute path to the source file without the `".erl"` extension. `Options` include the options which are necessary to recompile the file with `compile:file/2`, but excludes options such as `report` or `verbose` which do not change the way code is generated. The paths in the {`outdir, Path`} and {`i, Path`} options are guaranteed to be absolute.

gb_sets

Erlang Module

An implementation of ordered sets using Prof. Arne Andersson's General Balanced Trees. This can be much more efficient than using ordered lists, for larger sets, but depends on the application.

Complexity note

The complexity on set operations is bounded by either O(|S|) or O(|T| * log(|S|)), where S is the largest given set, depending on which is fastest for any particular function call. For operating on sets of almost equal size, this implementation is about 3 times slower than using ordered-list sets directly. For sets of very different sizes, however, this solution can be arbitrarily much faster; in practical cases, often between 10 and 100 times. This implementation is particularly suited for accumulating elements a few at a time, building up a large set (more than 100-200 elements), and repeatedly testing for membership in the current set.

As with normal tree structures, lookup (membership testing), insertion and deletion have logarithmic complexity.

Compatibility

All of the following functions in this module also exist and do the same thing in the `sets` and `ordsets` modules. That is, by only changing the module name for each call, you can try out different set representations.

- `add_element/2`
- `del_element/2`
- `filter/2`
- `fold/3`
- `from_list/1`
- `intersection/1`
- `intersection/2`
- `is_element/2`
- `is_set/1`
- `is_subset/2`
- `new/0`
- `size/1`
- `subtract/2`

- to_list/1
- union/1
- union/2

DATA TYPES

gb_set() = a GB set

Exports

add(Element, Set1) -> Set2
add_element(Element, Set1) -> Set2

Types:

- Element = term()
- Set1 = Set2 = gb_set()

Returns a new gb_set formed from Set1 with Element inserted. If Element is already an element in Set1, nothing is changed.

balance(Set1) -> Set2

Types:

- Set1 = Set2 = gb_set()

Rebalances the tree representation of Set1. Note that this is rarely necessary, but may be motivated when a large number of elements have been deleted from the tree without further insertions. Rebalancing could then be forced in order to minimise lookup times, since deletion only does not rebalance the tree.

delete(Element, Set1) -> Set2

Types:

- Element = term()
- Set1 = Set2 = gb_set()

Returns a new gb_set formed from Set1 with Element removed. Assumes that Element is present in Set1.

delete_any(Element, Set1) -> Set2
del_element(Element, Set1) -> Set2

Types:

- Element = term()
- Set1 = Set2 = gb_set()

Returns a new gb_set formed from Set1 with Element removed. If Element is not an element in Set1, nothing is changed.

difference(Set1, Set2) -> Set3

`subtract(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = gb_set()

Returns only the elements of `Set1` which are not also elements of `Set2`.

`empty() -> Set`
`new() -> Set`

Types:

- Set = gb_set()

Returns a new empty gb_set.

`filter(Pred, Set1) -> Set2`

Types:

- Pred = fun (E) -> bool()
- E = term()
- Set1 = Set2 = gb_set()

Filters elements in `Set1` using predicate function `Pred`.

`fold(Function, Acc0, Set) -> Acc1`

Types:

- Function = fun (E, AccIn) -> AccOut
- Acc0 = Acc1 = AccIn = AccOut = term()
- E = term()
- Set = gb_set()

Folds `Function` over every element in `Set` returning the final value of the accumulator.

`from_list(List) -> Set`

Types:

- List = [term()]
- Set = gb_set()

Returns a gb_set of the elements in `List`, where `List` may be unordered and contain duplicates.

`from_ordset(List) -> Set`

Types:

- List = [term()]
- Set = gb_set()

Turns an ordered-set list `List` into a gb_set. The list must not contain duplicates.

`insert(Element, Set1) -> Set2`

Types:

- Element = term()

- Set1 = Set2 = gb_set()

Returns a new gb_set formed from `Set1` with `Element` inserted. Assumes that `Element` is not present in `Set1`.

`intersection(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = gb_set()

Returns the intersection of `Set1` and `Set2`.

`intersection(SetList) -> Set`

Types:

- SetList = [gb_set()]
- Set = gb_set()

Returns the intersection of the non-empty list of gb_sets.

`is_empty(Set) -> bool()`

Types:

- Set = gb_set()

Returns `true` if `Set` is an empty set, and `false` otherwise.

`is_member(Element, Set) -> bool()`
`is_element(Element, Set) -> bool()`

Types:

- Element = term()
- Set = gb_set()

Returns `true` if `Element` is an element of `Set`, otherwise `false`.

`is_set(Set) -> bool()`

Types:

- Set = gb_set()

Returns `true` if `Set` appears to be a gb_set, otherwise `false`.

`is_subset(Set1, Set2) -> bool()`

Types:

- Set1 = Set2 = gb_set()

Returns `true` when every element of `Set1` is also a member of `Set2`, otherwise `false`.

`iterator(Set) -> Iter`

Types:

- Set = gb_set()
- Iter = term()

Returns an iterator that can be used for traversing the entries of Set; see next/1. The implementation of this is very efficient; traversing the whole set using next/1 is only slightly slower than getting the list of all elements using to_list/1 and traversing that. The main advantage of the iterator approach is that it does not require the complete list of all elements to be built in memory at one time.

```
largest(Set) -> term()
```

Types:

- Set = gb_set()

Returns the largest element in Set. Assumes that Set is nonempty.

```
next(Iter1) -> {Element, Iter2 | none}
```

Types:

- Iter1 = Iter2 = Element = term()

Returns {Element, Iter2} where Element is the smallest element referred to by the iterator Iter1, and Iter2 is the new iterator to be used for traversing the remaining elements, or the atom none if no elements remain.

```
singleton(Element) -> gb_set()
```

Types:

- Element = term()

Returns a gb_set containing only the element Element.

```
size(Set) -> int()
```

Types:

- Set = gb_set()

Returns the number of elements in Set.

```
smallest(Set) -> term()
```

Types:

- Set = gb_set()

Returns the smallest element in Set. Assumes that Set is nonempty.

```
take_largest(Set1) -> {Element, Set2}
```

Types:

- Set1 = Set2 = gb_set()
- Element = term()

Returns {Element, Set2}, where Element is the largest element in Set1, and Set2 is this set with Element deleted. Assumes that Set1 is nonempty.

```
take_smallest(Set1) -> {Element, Set2}
```

Types:

- Set1 = Set2 = gb_set()

- Element = term()

Returns {Element, Set2}, where Element is the smallest element in Set1, and Set2 is this set with Element deleted. Assumes that Set1 is nonempty.

`to_list(Set) -> List`

Types:

- Set = gb_set()
- List = [term()]

Returns the elements of Set as a list.

`union(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = gb_set()

Returns the merged (union) gb_set of Set1 and Set2.

`union(SetList) -> Set`

Types:

- SetList = [gb_set()]
- Set = gb_set()

Returns the merged (union) gb_set of the list of gb_sets.

SEE ALSO

gb_trees(3) [page 183], ordsets(3) [page 270], sets(3) [page 354]

gb_trees

Erlang Module

An efficient implementation of Prof. Arne Andersson's General Balanced Trees. These have no storage overhead compared to unbalaced binary trees, and their performance is in general better than AVL trees.

Data structure

Data structure:

```
- {Size, Tree}, where 'Tree' is composed of nodes of the form:
  - {Key, Value, Smaller, Bigger}, and the "empty tree" node:
  - nil.
```

There is no attempt to balance trees after deletions. Since deletions do not increase the height of a tree, this should be OK.

Original balance condition *h(T) <= ceil(c * log(|T|))* has been changed to the similar (but not quite equivalent) condition *2 ^ h(T) <= |T| ^ c*. This should also be OK.

Performance is comparable to the AVL trees in the Erlang book (and faster in general due to less overhead); the difference is that deletion works for these trees, but not for the book's trees. Behaviour is logaritmic (as it should be).

DATA TYPES

```
gb_tree() = a GB tree
```

Exports

`balance(Tree1) -> Tree2`

Types:

- Tree1 = Tree2 = gb_tree()

Rebalances `Tree1`. Note that this is rarely necessary, but may be motivated when a large number of nodes have been deleted from the tree without further insertions. Rebalancing could then be forced in order to minimise lookup times, since deletion only does not rebalance the tree.

`delete(Key, Tree1) -> Tree2`

Types:

- Key = term()
- Tree1 = Tree2 = gb_tree()

Removes the node with key `Key` from `Tree1`; returns new tree. Assumes that the key is present in the tree, crashes otherwise.

```
delete_any(Key, Tree1) -> Tree2
```

Types:

- Key = term()
- Tree1 = Tree2 = gb_tree()

Removes the node with key `Key` from `Tree1` if the key is present in the tree, otherwise does nothing; returns new tree.

```
empty() -> Tree
```

Types:

- Tree = gb_tree()

Returns a new empty tree

```
enter(Key, Val, Tree1) -> Tree2
```

Types:

- Key = Val = term()
- Tree1 = Tree2 = gb_tree()

Inserts `Key` with value `Val` into `Tree1` if the key is not present in the tree, otherwise updates `Key` to value `Val` in `Tree1`. Returns the new tree.

```
from_orddict(List) -> Tree
```

Types:

- List = [{Key, Val}]
- Key = Val = term()
- Tree = gb_tree()

Turns an ordered list `List` of key-value tuples into a tree. The list must not contain duplicate keys.

```
get(Key, Tree) -> Val
```

Types:

- Key = Val = term()
- Tree = gb_tree()

Retrieves the value stored with `Key` in `Tree`. Assumes that the key is present in the tree, crashes otherwise.

```
lookup(Key, Tree) -> {value, Val} | none
```

Types:

- Key = Val = term()
- Tree = gb_tree()

Looks up Key in Tree; returns {value, Val}, or none if Key is not present.

```
insert(Key, Val, Tree1) -> Tree2
```

Types:

- Key = Val = term()
- Tree1 = Tree2 = gb_tree()

Inserts Key with value Val into Tree1; returns the new tree. Assumes that the key is not present in the tree, crashes otherwise.

```
is_defined(Key, Tree) -> bool()
```

Types:

- Tree = gb_tree()

Returns true if Key is present in Tree, otherwise false.

```
is_empty(Tree) -> bool()
```

Types:

- Tree = gb_tree()

Returns true if Tree is an empty tree, and false otherwise.

```
iterator(Tree) -> Iter
```

Types:

- Tree = gb_tree()
- Iter = term()

Returns an iterator that can be used for traversing the entries of Tree; see next/1. The implementation of this is very efficient; traversing the whole tree using next/1 is only slightly slower than getting the list of all elements using to_list/1 and traversing that. The main advantage of the iterator approach is that it does not require the complete list of all elements to be built in memory at one time.

```
keys(Tree) -> [Key]
```

Types:

- Tree = gb_tree()
- Key = term()

Returns the keys in Tree as an ordered list.

```
largest(Tree) -> {Key, Val}
```

Types:

- Tree = gb_tree()
- Key = Val = term()

Returns {Key, Val}, where Key is the largest key in Tree, and Val is the value associated with this key. Assumes that the tree is nonempty.

```
next(Iter1) -> {Key, Val, Iter2} | none
```

Types:

- Iter1 = Iter2 = Key = Val = term()

Returns {Key, Val, Iter2} where Key is the smallest key referred to by the iterator Iter1, and Iter2 is the new iterator to be used for traversing the remaining nodes, or the atom none if no nodes remain.

`size(Tree) -> int()`

Types:

- Tree = gb_tree()

Returns the number of nodes in Tree.

`smallest(Tree) -> {Key, Val}`

Types:

- Tree = gb_tree()
- Key = Val = term()

Returns {Key, Val}, where Key is the smallest key in Tree, and Val is the value associated with this key. Assumes that the tree is nonempty.

`take_largest(Tree1) -> {Key, Val, Tree2}`

Types:

- Tree1 = Tree2 = gb_tree()
- Key = Val = term()

Returns {Key, Val, Tree2}, where Key is the largest key in Tree1, Val is the value associated with this key, and Tree2 is this tree with the corresponding node deleted. Assumes that the tree is nonempty.

`take_smallest(Tree1) -> {Key, Val, Tree2}`

Types:

- Tree1 = Tree2 = gb_tree()
- Key = Val = term()

Returns {Key, Val, Tree2}, where Key is the smallest key in Tree1, Val is the value associated with this key, and Tree2 is this tree with the corresponding node deleted. Assumes that the tree is nonempty.

`to_list(Tree) -> [{Key, Val}]`

Types:

- Tree = gb_tree()
- Key = Val = term()

Converts a tree into an ordered list of key-value tuples.

`update(Key, Val, Tree1) -> Tree2`

Types:

- Key = Val = term()

- Tree1 = Tree2 = gb_tree()

Updates `Key` to value `Val` in `Tree1`; returns the new tree. Assumes that the key is present in the tree.

`values(Tree) -> [Val]`

Types:

- Tree = gb_tree()
- Val = term()

Returns the values in `Tree` as an ordered list, sorted by their corresponding keys. Duplicates are not removed.

SEE ALSO

gb_sets(3) [page 177], dict(3) [page 97]

gen_event

Erlang Module

A behaviour module for implementing event handling functionality. The OTP event handling model consists of a generic event manager process with an arbitrary number of event handlers which are added and deleted dynamically.

An event manager implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into an OTP supervision tree. Refer to *OTP Design Principles* for more information.

Each event handler is implemented as a callback module exporting a pre-defined set of functions. The relationship between the behaviour functions and the callback functions can be illustrated as follows:

```
gen_event module                    Callback module
---------------                     -------------
gen_event:start_link       ----->   -

gen_event:add_handler
gen_event:add_suphandler   ----->   Module:init/1

gen_event:notify
gen_event:sync_notify      ----->   Module:handle_event/2

gen_event:call             ----->   Module:handle_call/2

-                          ----->   Module:handle_info/2

gen_event:delete_handler   ----->   Module:terminate/2

gen_event:swap_handler
gen_event:swap_sup_handler ----->   Module1:terminate/2
                                    Module2:init/1

gen_event:which_handlers   ----->   -

gen_event:stop             ----->   Module:terminate/2

-                          ----->   Module:code_change/3
```

Since each event handler is one callback module, an event manager will have several callback modules which are added and deleted dynamically. Therefore `gen_event` is more tolerant of callback module errors than the other behaviours. If a callback function for an installed event handler fails with `Reason`, or returns a bad value `Term`, the event manager will not fail. It will delete the event handler by calling the callback function `Module:terminate/2` (see below), giving as argument `{error,{'EXIT',Reason}}` or `{error,Term}`, respectively. No other event handler will be affected.

The sys module can be used for debugging an event manager.

Note that an event manager *does* trap exit signals automatically.

The gen_event process can go into hibernation (see [erlang(3)]) if a callback function in a handler module specifies `'hibernate'` in its return value. This might be useful if the server is expected to be idle for a long time. However this feature should be used with care as hibernation implies at least two garbage collections (when hibernating and shortly after waking up) and is not something you'd want to do between each event handled by a busy event manager.

It's also worth noting that when multiple event handlers are invoked, it's sufficient that one single event handler returns a `'hibernate'` request for the whole event manager to go into hibernation.

Unless otherwise stated, all functions in this module fail if the specified event manager does not exist or if bad arguments are given.

Exports

```
start_link() -> Result
start_link(EventMgrName) -> Result
```

Types:

- EventMgrName = {local,Name} | {global,Name}
- Name = atom()
- Result = {ok,Pid} | {error,{already_started,Pid}}
- Pid = pid()

Creates an event manager process as part of a supervision tree. The function should be called, directly or indirectly, by the supervisor. It will, among other things, ensure that the event manager is linked to the supervisor.

If `EventMgrName={local,Name}`, the event manager is registered locally as `Name` using `register/2`. If `EventMgrName={global,Name}`, the event manager is registered globally as `Name` using `global:register_name/2`. If no name is provided, the event manager is not registered.

If the event manager is successfully created the function returns `{ok,Pid}`, where `Pid` is the pid of the event manager. If there already exists a process with the specified `EventMgrName` the function returns `{error,{already_started,Pid}}`, where `Pid` is the pid of that process.

```
start() -> Result
start(EventMgrName) -> Result
```

Types:

- EventMgrName = {local,Name} | {global,Name}
- Name = atom()
- Result = {ok,Pid} | {error,{already_started,Pid}}
- Pid = pid()

Creates a stand-alone event manager process, i.e. an event manager which is not part of a supervision tree and thus has no supervisor.

See `start_link/0,1` for a description of arguments and return values.

`add_handler(EventMgrRef, Handler, Args) -> Result`

Types:

- EventMgr = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler = Module | {Module,Id}
- Module = atom()
- Id = term()
- Args = term()
- Result = ok | {'EXIT',Reason} | term()
- Reason = term()

Adds a new event handler to the event manager `EventMgrRef`. The event manager will call `Module:init/1` to initiate the event handler and its internal state.

`EventMgrRef` can be:

- the pid,
- `Name`, if the event manager is locally registered,
- `{Name,Node}`, if the event manager is locally registered at another node, or
- `{global,Name}`, if the event manager is globally registered.

`Handler` is the name of the callback module `Module` or a tuple `{Module,Id}`, where `Id` is any term. The `{Module,Id}` representation makes it possible to identify a specific event handler when there are several event handlers using the same callback module.

`Args` is an arbitrary term which is passed as the argument to `Module:init/1`.

If `Module:init/1` returns a correct value, the event manager adds the event handler and this function returns `ok`. If `Module:init/1` fails with `Reason` or returns an unexpected value `Term`, the event handler is ignored and this function returns `{'EXIT',Reason}` or `Term`, respectively.

`add_sup_handler(EventMgrRef, Handler, Args) -> Result`

Types:

- EventMgr = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler = Module | {Module,Id}
- Module = atom()
- Id = term()
- Args = term()
- Result = ok | {'EXIT',Reason} | term()
- Reason = term()

Adds a new event handler in the same way as `add_handler/3` but will also supervise the connection between the event handler and the calling process.

- If the calling process later terminates with Reason, the event manager will delete the event handler by calling Module:terminate/2 with {stop,Reason} as argument.
- If the event handler later is deleted, the event manager sends a message{gen_event_EXIT,Handler,Reason} to the calling process. Reason is one of the following:
  - normal, if the event handler has been removed due to a call to delete_handler/3, or remove_handler has been returned by a callback function (see below).
  - shutdown, if the event handler has been removed because the event manager is terminating.
  - {swapped,NewHandler,Pid}, if the process Pid has replaced the event handler with another event handler NewHandler using a call to swap_handler/3 or swap_sup_handler/3.
  - a term, if the event handler is removed due to an error. Which term depends on the error.

See add_handler/3 for a description of the arguments and return values.

```
notify(EventMgrRef, Event) -> ok
sync_notify(EventMgrRef, Event) -> ok
```

Types:

- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Event = term()

Sends an event notification to the event manager EventMgrRef. The event manager will call Module:handle_event/2 for each installed event handler to handle the event.

notify is asynchronous and will return immediately after the event notification has been sent. sync_notify is synchronous in the sense that it will return ok after the event has been handled by all event handlers.

See add_handler/3 for a description of EventMgrRef.

Event is an arbitrary term which is passed as one of the arguments to Module:handle_event/2.

notify will not fail even if the specified event manager does not exist, unless it is specified as Name.

```
call(EventMgrRef, Handler, Request) -> Result
call(EventMgrRef, Handler, Request, Timeout) -> Result
```

Types:

- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler = Module | {Module,Id}
- Module = atom()
- Id = term()
- Request = term()
- Timeout = int()>0 | infinity

- Result = Reply | {error,Error}
- Reply = term()
- Error = bad_module | {'EXIT',Reason} | term()
- Reason = term()

Makes a synchronous call to the event handler `Handler` installed in the event manager `EventMgrRef` by sending a request and waiting until a reply arrives or a timeout occurs. The event manager will call `Module:handle_call/2` to handle the request.

See `add_handler/3` for a description of `EventMgrRef` and `Handler`.

`Request` is an arbitrary term which is passed as one of the arguments to `Module:handle_call/2`.

`Timeout` is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom `infinity` to wait indefinitely. Default value is 5000. If no reply is received within the specified time, the function call fails.

The return value `Reply` is defined in the return value of `Module:handle_call/2`. If the specified event handler is not installed, the function returns `{error,bad_module}`. If the callback function fails with `Reason` or returns an unexpected value `Term`, this function returns `{error,{'EXIT',Reason}}` or `{error,Term}`, respectively.

`delete_handler(EventMgrRef, Handler, Args) -> Result`

Types:

- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler = Module | {Module,Id}
- Module = atom()
- Id = term()
- Args = term()
- Result = term() | {error,module_not_found} | {'EXIT',Reason}
- Reason = term()

Deletes an event handler from the event manager `EventMgrRef`. The event manager will call `Module:terminate/2` to terminate the event handler.

See `add_handler/3` for a description of `EventMgrRef` and `Handler`.

`Args` is an arbitrary term which is passed as one of the arguments to `Module:terminate/2`.

The return value is the return value of `Module:terminate/2`. If the specified event handler is not installed, the function returns `{error,module_not_found}`. If the callback function fails with `Reason`, the function returns `{'EXIT',Reason}`.

`swap_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result`

Types:

- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler1 = Handler2 = Module | {Module,Id}
- Module = atom()
- Id = term()
- Args1 = Args2 = term()

- Result = ok | {error,Error}
- Error = {'EXIT',Reason} | term()
- Reason = term()

Replaces an old event handler with a new event handler in the event manager `EventMgrRef`.

See `add_handler/3` for a description of the arguments.

First the old event handler `Handler1` is deleted. The event manager calls `Module1:terminate(Args1, ...)`, where `Module1` is the callback module of `Handler1`, and collects the return value.

Then the new event handler `Handler2` is added and initiated by calling `Module2:init({Args2,Term})`, where `Module2` is the callback module of `Handler2` and `Term` the return value of `Module1:terminate/2`. This makes it possible to transfer information from `Handler1` to `Handler2`.

The new handler will be added even if the the specified old event handler is not installed in which case `Term=error`, or if `Module1:terminate/2` fails with `Reason` in which case `Term={'EXIT',Reason}`. The old handler will be deleted even if `Module2:init/1` fails.

If there was a supervised connection between `Handler1` and a process `Pid`, there will be a supervised connection between `Handler2` and `Pid` instead.

If `Module2:init/1` returns a correct value, this function returns `ok`. If `Module2:init/1` fails with `Reason` or returns an unexpected value `Term`, this this function returns `{error,{'EXIT',Reason}}` or `{error,Term}`, respectively.

`swap_sup_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result`

Types:

- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler1 = Handler 2 = Module | {Module,Id}
- Module = atom()
- Id = term()
- Args1 = Args2 = term()
- Result = ok | {error,Error}
- Error = {'EXIT',Reason} | term()
- Reason = term()

Replaces an event handler in the event manager `EventMgrRef` in the same way as `swap_handler/3` but will also supervise the connection between `Handler2` and the calling process.

See `swap_handler/3` for a description of the arguments and return values.

`which_handlers(EventMgrRef) -> [Handler]`

Types:

- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler = Module | {Module,Id}
- Module = atom()
- Id = term()

Returns a list of all event handlers installed in the event manager `EventMgrRef`.

See `add_handler/3` for a description of `EventMgrRef` and `Handler`.

`stop(EventMgrRef) -> ok`

Types:

- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()

Terminates the event manager `EventMgrRef`. Before terminating, the event manager will call `Module:terminate(stop,...)` for each installed event handler.

See `add_handler/3` for a description of the argument.

CALLBACK FUNCTIONS

The following functions should be exported from a `gen_event` callback module.

Exports

`Module:init(InitArgs) -> {ok,State} | {ok,State,hibernate}`

Types:

- InitArgs = Args | {Args,Term}
- Args = Term = term()
- State = term()

Whenever a new event handler is added to an event manager, this function is called to initialize the event handler.

If the event handler is added due to a call to `gen_event:add_handler/3` or `gen_event:add_sup_handler/3`, `InitArgs` is the `Args` argument of these functions.

If the event handler is replacing another event handler due to a call to `gen_event:swap_handler/3` or `gen_event:swap_sup_handler/3`, or due to a `swap` return tuple from one of the other callback functions, `InitArgs` is a tuple `{Args,Term}` where `Args` is the argument provided in the function call/return tuple and `Term` is the result of terminating the old event handler, see `gen_event:swap_handler/3`.

The function should return `{ok,State}` or `{ok,State, hibernate}` where `State` is the initial internal state of the event handler.

If `{ok,State,hibernate}` is returned, the event manager will go into hibernation (by calling proc_lib:hibernate/3 [page 281]), waiting for the next event to occur.

`Module:handle_event(Event, State) -> Result`

Types:

- Event = term()
- State = term()
- Result = {ok,NewState} | {ok,NewState,hibernate}
- | {swap_handler,Args1,NewState,Handler2,Args2} | remove_handler
- NewState = term()

- Args1 = Args2 = term()
- Handler2 = Module2 | {Module2,Id}
- Module2 = atom()
- Id = term()

Whenever an event manager receives an event sent using `gen_event:notify/2` or `gen_event:sync_notify/2`, this function is called for each installed event handler to handle the event.

`Event` is the `Event` argument of `notify`/`sync_notify`.

`State` is the internal state of the event handler.

If the function returns {`ok`,`NewState`} or {`ok`,`NewState`,`hibernate`} the event handler will remain in the event manager with the possible updated internal state `NewState`.

If {`ok`,`NewState`,`hibernate`} is returned, the event manager will also go into hibernation (by calling proc_lib:hibernate/3 [page 281]), waiting for the next event to occur. It is sufficient that one of the event handlers return {`ok`,`NewState`,`hibernate`} for the whole event manager process to hibernate.

If the function returns {`swap_handler`,`Args1`,`NewState`,`Handler2`,`Args2`} the event handler will be replaced by `Handler2` by first calling `Module:terminate(Args1,NewState)` and then `Module2:init({Args2,Term})` where `Term` is the return value of `Module:terminate/2`. See `gen_event:swap_handler/3` for more information.

If the function returns `remove_handler` the event handler will be deleted by calling `Module:terminate(remove_handler,State)`.

`Module:handle_call(Request, State) -> Result`

Types:

- Request = term()
- State = term()
- Result = {ok,Reply,NewState} | {ok,Reply,NewState,hibernate}
- | {swap_handler,Reply,Args1,NewState,Handler2,Args2}
- | {remove_handler, Reply}
- Reply = term()
- NewState = term()
- Args1 = Args2 = term()
- Handler2 = Module2 | {Module2,Id}
- Module2 = atom()
- Id = term()

Whenever an event manager receives a request sent using `gen_event:call/3,4`, this function is called for the specified event handler to handle the request.

`Request` is the `Request` argument of `call`.

`State` is the internal state of the event handler.

The return values are the same as for `handle_event/2` except they also contain a term `Reply` which is the reply given back to the client as the return value of `call`.

`Module:handle_info(Info, State) -> Result`

Types:

- Info = term()
- State = term()
- Result = {ok,NewState} | {ok,NewState,hibernate}
- | {swap_handler,Args1,NewState,Handler2,Args2} | remove_handler
- NewState = term()
- Args1 = Args2 = term()
- Handler2 = Module2 | {Module2,Id}
- Module2 = atom()
- Id = term()

This function is called for each installed event handler when an event manager receives any other message than an event or a synchronous request (or a system message).

`Info` is the received message.

See `Module:handle_event/2` for a description of State and possible return values.

`Module:terminate(Arg, State) -> term()`

Types:

- Arg = Args | {stop,Reason} | stop | remove_handler
- | {error,{'EXIT',Reason}} | {error,Term}
- Args = Reason = Term = term()

Whenever an event handler is deleted from an event manager, this function is called. It should be the opposite of `Module:init/1` and do any necessary cleaning up.

If the event handler is deleted due to a call to `gen_event:delete_handler`, `gen_event:swap_handler/3` or `gen_event:swap_sup_handler/3`, `Arg` is the `Args` argument of this function call.

`Arg={stop,Reason}` if the event handler has a supervised connection to a process which has terminated with reason `Reason`.

`Arg=stop` if the event handler is deleted because the event manager is terminating.

`Arg=remove_handler` if the event handler is deleted because another callback function has returned `remove_handler` or `{remove_handler,Reply}`.

`Arg={error,Term}` if the event handler is deleted because a callback function returned an unexpected value `Term`, or `Arg={error,{'EXIT',Reason}}` if a callback function failed.

`State` is the internal state of the event handler.

The function may return any term. If the event handler is deleted due to a call to `gen_event:delete_handler`, the return value of that function will be the return value of this function. If the event handler is to be replaced with another event handler due to a swap, the return value will be passed to the `init` function of the new event handler. Otherwise the return value is ignored.

`Module:code_change(OldVsn, State, Extra) -> {ok, NewState}`

Types:

- OldVsn = Vsn | {down, Vsn}
- Vsn = term()
- State = NewState = term()
- Extra = term()

This function is called for an installed event handler which should update its internal state during a release upgrade/downgrade, i.e. when the instruction {update,Module,Change,...} where Change={advanced,Extra} is given in the .appup file. See *OTP Design Principles* for more information.

In the case of an upgrade, OldVsn is Vsn, and in the case of a downgrade, OldVsn is {down,Vsn}. Vsn is defined by the vsn attribute(s) of the old version of the callback module Module. If no such attribute is defined, the version is the checksum of the BEAM file.

State is the internal state of the event handler.

Extra is passed as-is from the {advanced,Extra} part of the update instruction.

The function should return the updated internal state.

SEE ALSO

supervisor(3) [page 401], sys(3) [page 411]

gen_fsm

Erlang Module

A behaviour module for implementing a finite state machine. A generic finite state machine process (gen_fsm) implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into an OTP supervision tree. Refer to [OTP Design Principles] for more information.

A gen_fsm assumes all specific parts to be located in a callback module exporting a pre-defined set of functions. The relationship between the behaviour functions and the callback functions can be illustrated as follows:

```
gen_fsm module                    Callback module
--------------                    ---------------
gen_fsm:start_link                -----> Module:init/1

gen_fsm:send_event                -----> Module:StateName/2

gen_fsm:send_all_state_event      -----> Module:handle_event/3

gen_fsm:sync_send_event           -----> Module:StateName/3

gen_fsm:sync_send_all_state_event -----> Module:handle_sync_event/4

-                                 -----> Module:handle_info/3

-                                 -----> Module:terminate/3

-                                 -----> Module:code_change/4
```

If a callback function fails or returns a bad value, the gen_fsm will terminate.

The sys module can be used for debugging a gen_fsm.

Note that a gen_fsm does not trap exit signals automatically, this must be explicitly initiated in the callback module.

Unless otherwise stated, all functions in this module fail if the specified gen_fsm does not exist or if bad arguments are given.

The gen_fsm process can go into hibernation (see [erlang(3)]) if a callback function specifies `'hibernate'` instead of a timeout value. This might be useful if the server is expected to be idle for a long time. However this feature should be used with care as hibernation implies at least two garbage collections (when hibernating and shortly after waking up) and is not something you'd want to do between each call to a busy state machine.

Exports

```
start_link(Module, Args, Options) -> Result
start_link(FsmName, Module, Args, Options) -> Result
```

Types:

- FsmName = {local,Name} | {global,GlobalName}
- Name = atom()
- GlobalName = term()
- Module = atom()
- Args = term()
- Options = [Option]
- Option = {debug,Dbgs} | {timeout,Time} | {spawn_opt,SOpts}
- Dbgs = [Dbg]
- Dbg = trace | log | statistics
- | {log_to_file,FileName} | {install,{Func,FuncState}}
- SOpts = [SOpt]
- SOpt - see erlang:spawn_opt/2,3,4,5
- Result = {ok,Pid} | ignore | {error,Error}
- Pid = pid()
- Error = {already_started,Pid} | term()

Creates a gen_fsm process as part of a supervision tree. The function should be called, directly or indirectly, by the supervisor. It will, among other things, ensure that the gen_fsm is linked to the supervisor.

The gen_fsm process calls `Module:init/1` to initialize. To ensure a synchronized start-up procedure, `start_link/3,4` does not return until `Module:init/1` has returned.

If `FsmName={local,Name}`, the gen_fsm is registered locally as `Name` using `register/2`. If `FsmName={global,GlobalName}`, the gen_fsm is registered globally as `GlobalName` using `global:register_name/2`. If no name is provided, the gen_fsm is not registered.

`Module` is the name of the callback module.

`Args` is an arbitrary term which is passed as the argument to `Module:init/1`.

If the option `{timeout,Time}` is present, the gen_fsm is allowed to spend `Time` milliseconds initializing or it will be terminated and the start function will return `{error,timeout}`.

If the option `{debug,Dbgs}` is present, the corresponding `sys` function will be called for each item in `Dbgs`. See sys(3) [page 411].

If the option `{spawn_opt,SOpts}` is present, `SOpts` will be passed as option list to the `spawn_opt` BIF which is used to spawn the gen_fsm process. See [erlang(3)].

> **Note:**
> Using the spawn option `monitor` is currently not allowed, but will cause the function to fail with reason `badarg`.

If the gen_fsm is successfully created and initialized the function returns `{ok,Pid}`, where `Pid` is the pid of the gen_fsm. If there already exists a process with the specified

FsmName, the function returns {error,{already_started,Pid}} where Pid is the pid of that process.

If Module:init/1 fails with Reason, the function returns {error,Reason}. If Module:init/1 returns {stop,Reason} or ignore, the process is terminated and the function returns {error,Reason} or ignore, respectively.

```
start(Module, Args, Options) -> Result
start(FsmName, Module, Args, Options) -> Result
```

Types:

- FsmName = {local,Name} | {global,GlobalName}
- Name = atom()
- GlobalName = term()
- Module = atom()
- Args = term()
- Options = [Option]
- Option = {debug,Dbgs} | {timeout,Time} | {spawn_opt,SOpts}
- Dbgs = [Dbg]
- Dbg = trace | log | statistics
- | {log_to_file,FileName} | {install,{Func,FuncState}}
- SOpts = [term()]
- Result = {ok,Pid} | ignore | {error,Error}
- Pid = pid()
- Error = {already_started,Pid} | term()

Creates a stand-alone gen_fsm process, i.e. a gen_fsm which is not part of a supervision tree and thus has no supervisor.

See start_link/3,4 [page 199] for a description of arguments and return values.

```
send_event(FsmRef, Event) -> ok
```

Types:

- FsmRef = Name | {Name,Node} | {global,GlobalName} | pid()
- Name = Node = atom()
- GlobalName = term()
- Event = term()

Sends an event asynchronously to the gen_fsm FsmRef and returns ok immediately. The gen_fsm will call Module:StateName/2 to handle the event, where StateName is the name of the current state of the gen_fsm.

FsmRef can be:

- the pid,
- Name, if the gen_fsm is locally registered,
- {Name,Node}, if the gen_fsm is locally registered at another node, or
- {global,GlobalName}, if the gen_fsm is globally registered.

Event is an arbitrary term which is passed as one of the arguments to Module:StateName/2.

```
send_all_state_event(FsmRef, Event) -> ok
```

Types:

- FsmRef = Name | {Name,Node} | {global,GlobalName} | pid()
- Name = Node = atom()
- GlobalName = term()
- Event = term()

Sends an event asynchronously to the gen_fsm `FsmRef` and returns `ok` immediately. The gen_fsm will call `Module:handle_event/3` to handle the event.

See send_event/2 [page 200] for a description of the arguments.

The difference between `send_event` and `send_all_state_event` is which callback function is used to handle the event. This function is useful when sending events that are handled the same way in every state, as only one `handle_event` clause is needed to handle the event instead of one clause in each state name function.

```
sync_send_event(FsmRef, Event) -> Reply
sync_send_event(FsmRef, Event, Timeout) -> Reply
```

Types:

- FsmRef = Name | {Name,Node} | {global,GlobalName} | pid()
- Name = Node = atom()
- GlobalName = term()
- Event = term()
- Timeout = int()>0 | infinity
- Reply = term()

Sends an event to the gen_fsm `FsmRef` and waits until a reply arrives or a timeout occurs. The gen_fsm will call `Module:StateName/3` to handle the event, where `StateName` is the name of the current state of the gen_fsm.

See send_event/2 [page 200] for a description of `FsmRef` and `Event`.

`Timeout` is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom `infinity` to wait indefinitely. Default value is 5000. If no reply is received within the specified time, the function call fails.

The return value `Reply` is defined in the return value of `Module:StateName/3`.

The ancient behaviour of sometimes consuming the server exit message if the server died during the call while linked to the client has been removed in OTP R12B/Erlang 5.6.

```
sync_send_all_state_event(FsmRef, Event) -> Reply
sync_send_all_state_event(FsmRef, Event, Timeout) -> Reply
```

Types:

- FsmRef = Name | {Name,Node} | {global,GlobalName} | pid()
- Name = Node = atom()
- GlobalName = term()
- Event = term()
- Timeout = int()>0 | infinity
- Reply = term()

Sends an event to the gen_fsm `FsmRef` and waits until a reply arrives or a timeout occurs. The gen_fsm will call `Module:handle_sync_event/4` to handle the event.

See send_event/2 [page 200] for a description of `FsmRef` and `Event`. See sync_send_event/3 [page 201] for a description of `Timeout` and `Reply`.

See send_all_state_event/2 [page 201] for a discussion about the difference between `sync_send_event` and `sync_send_all_state_event`.

`reply(Caller, Reply) -> true`

Types:

- Caller - see below
- Reply = term()

This function can be used by a gen_fsm to explicitly send a reply to a client process that called sync_send_event/2,3 [page 201] or sync_send_all_state_event/2,3 [page 201], when the reply cannot be defined in the return value of `Module:State/3` or `Module:handle_sync_event/4`.

`Caller` must be the `From` argument provided to the callback function. `Reply` is an arbitrary term, which will be given back to the client as the return value of `sync_send_event/2,3` or `sync_send_all_state_event/2,3`.

`send_event_after(Time, Event) -> Ref`

Types:

- Time = integer()
- Event = term()
- Ref = reference()

Sends a delayed event internally in the gen_fsm that calls this function after `Time` ms. Returns immediately a reference that can be used to cancel the delayed send using cancel_timer/1 [page 203].

The gen_fsm will call `Module:StateName/2` to handle the event, where `StateName` is the name of the current state of the gen_fsm at the time the delayed event is delivered.

`Event` is an arbitrary term which is passed as one of the arguments to `Module:StateName/2`.

`start_timer(Time, Msg) -> Ref`

Types:

- Time = integer()
- Msg = term()
- Ref = reference()

Sends a timeout event internally in the gen_fsm that calls this function after `Time` ms. Returns immediately a reference that can be used to cancel the timer using cancel_timer/1 [page 203].

The gen_fsm will call `Module:StateName/2` to handle the event, where `StateName` is the name of the current state of the gen_fsm at the time the timeout message is delivered.

`Msg` is an arbitrary term which is passed in the timeout message, `{timeout, Ref, Msg}`, as one of the arguments to `Module:StateName/2`.

```
cancel_timer(Ref) -> RemainingTime | false
```

Types:

- Ref = reference()
- RemainingTime = integer()

Cancels an internal timer referred by `Ref` in the gen_fsm that calls this function.

`Ref` is a reference returned from send_event_after/2 [page 202] or start_timer/2 [page 202].

If the timer has already timed out, but the event not yet been delivered, it is cancelled as if it had *not* timed out, so there will be no false timer event after returning from this function.

Returns the remaining time in ms until the timer would have expired if `Ref` referred to an active timer, `false` otherwise.

```
enter_loop(Module, Options, StateName, StateData)
enter_loop(Module, Options, StateName, StateData, FsmName)
enter_loop(Module, Options, StateName, StateData, Timeout)
enter_loop(Module, Options, StateName, StateData, FsmName, Timeout)
```

Types:

- Module = atom()
- Options = [Option]
- Option = {debug,Dbgs}
- Dbgs = [Dbg]
- Dbg = trace | log | statistics
- | {log_to_file,FileName} | {install,{Func,FuncState}}
- StateName = atom()
- StateData = term()
- FsmName = {local,Name} | {global,GlobalName}
- Name = atom()
- GlobalName = term()
- Timeout = int() | infinity

Makes an existing process into a gen_fsm. Does not return, instead the calling process will enter the gen_fsm receive loop and become a gen_fsm process. The process *must* have been started using one of the start functions in `proc_lib`, see proc_lib(3) [page 277]. The user is responsible for any initialization of the process, including registering a name for it.

This function is useful when a more complex initialization procedure is needed than the gen_fsm behaviour provides.

`Module`, `Options` and `FsmName` have the same meanings as when calling start[_link]/3,4 [page 199]. However, if `FsmName` is specified, the process must have been registered accordingly *before* this function is called.

`StateName`, `StateData` and `Timeout` have the same meanings as in the return value of Module:init/1 [page 204]. Also, the callback module `Module` does not need to export an `init/1` function.

Failure: If the calling process was not started by a `proc_lib` start function, or if it is not registered according to `FsmName`.

CALLBACK FUNCTIONS

The following functions should be exported from a `gen_fsm` callback module.

In the description, the expression *state name* is used to denote a state of the state machine. *state data* is used to denote the internal state of the Erlang process which implements the state machine.

Exports

`Module:init(Args) -> Result`

Types:

- Args = term()
- Return = {ok,StateName,StateData} | {ok,StateName,StateData,Timeout}
- | {ok,StateName,StateData,hibernate}
- | {stop,Reason} | ignore
- StateName = atom()
- StateData = term()
- Timeout = int()>0 | infinity
- Reason = term()

Whenever a gen_fsm is started using gen_fsm:start/3,4 [page 200] or gen_fsm:start_link/3,4 [page 199], this function is called by the new process to initialize.

`Args` is the `Args` argument provided to the start function.

If initialization is successful, the function should return {`ok,StateName,StateData`}, {`ok,StateName,StateData,Timeout`} or {`ok,StateName,StateData,hibernate`}, where `StateName` is the initial state name and `StateData` the initial state data of the gen_fsm.

If an integer timeout value is provided, a timeout will occur unless an event or a message is received within `Timeout` milliseconds. A timeout is represented by the atom `timeout` and should be handled by the `Module:StateName/2` callback functions. The atom `infinity` can be used to wait indefinitely, this is the default value.

If `hibernate` is specified instead of a timeout value, the process will go into hibernation when waiting for the next message to arrive (by calling proc_lib:hibernate/3 [page 281]).

If something goes wrong during the initialization the function should return {`stop,Reason`}, where `Reason` is any term, or `ignore`.

`Module:StateName(Event, StateData) -> Result`

Types:

- Event = timeout | term()
- StateData = term()
- Result = {next_state,NextStateName,NewStateData}
- | {next_state,NextStateName,NewStateData,Timeout}
- | {next_state,NextStateName,NewStateData,hibernate}
- | {stop,Reason,NewStateData}

- NextStateName = atom()
- NewStateData = term()
- Timeout = int()>0 | infinity
- Reason = term()

There should be one instance of this function for each possible state name. Whenever a gen_fsm receives an event sent using gen_fsm:send_event/2 [page 200], the instance of this function with the same name as the current state name `StateName` is called to handle the event. It is also called if a timeout occurs.

`Event` is either the atom `timeout`, if a timeout has occurred, or the `Event` argument provided to `send_event/2`.

`StateData` is the state data of the gen_fsm.

If the function returns {`next_state`,`NextStateName`,`NewStateData`}, {`next_state`,`NextStateName`,`NewStateData`,`Timeout`} or {`next_state`,`NextStateName`,`NewStateData`,`hibernate`}, the gen_fsm will continue executing with the current state name set to `NextStateName` and with the possibly updated state data `NewStateData`. See `Module:init/1` for a description of `Timeout` and `hibernate`.

If the function returns {`stop`,`Reason`,`NewStateData`}, the gen_fsm will call `Module:terminate(Reason,NewStateData)` and terminate.

`Module:handle_event(Event, StateName, StateData) -> Result`

Types:

- Event = term()
- StateName = atom()
- StateData = term()
- Result = {next_state,NextStateName,NewStateData}
- | {next_state,NextStateName,NewStateData,Timeout}
- | {next_state,NextStateName,NewStateData,hibernate}
- | {stop,Reason,NewStateData}
- NextStateName = atom()
- NewStateData = term()
- Timeout = int()>0 | infinity
- Reason = term()

Whenever a gen_fsm receives an event sent using gen_fsm:send_all_state_event/2 [page 201], this function is called to handle the event.

`StateName` is the current state name of the gen_fsm.

See `Module:StateName/2` for a description of the other arguments and possible return values.

`Module:StateName(Event, From, StateData) -> Result`

Types:

- Event = term()
- From = {pid(),Tag}
- StateData = term()
- Result = {reply,Reply,NextStateName,NewStateData}

- | {reply,Reply,NextStateName,NewStateData,Timeout}
- | {reply,Reply,NextStateName,NewStateData,hibernate}
- | {next_state,NextStateName,NewStateData}
- | {next_state,NextStateName,NewStateData,Timeout}
- | {next_state,NextStateName,NewStateData,hibernate}
- | {stop,Reason,Reply,NewStateData} | {stop,Reason,NewStateData}
- Reply = term()
- NextStateName = atom()
- NewStateData = term()
- Timeout = int()>0 | infinity
- Reason = normal | term()

There should be one instance of this function for each possible state name. Whenever a gen_fsm receives an event sent using gen_fsm:sync_send_event/2,3 [page 201], the instance of this function with the same name as the current state name `StateName` is called to handle the event.

`Event` is the `Event` argument provided to `sync_send_event`.

`From` is a tuple {`Pid`,`Tag`} where `Pid` is the pid of the process which called `sync_send_event/2,3` and `Tag` is a unique tag.

`StateData` is the state data of the gen_fsm.

If the function returns {`reply`,`Reply`,`NextStateName`,`NewStateData`}, {`reply`,`Reply`,`NextStateName`,`NewStateData`,`Timeout`} or {`reply`,`Reply`,`NextStateName`,`NewStateData`,`hibernate`}, `Reply` will be given back to `From` as the return value of `sync_send_event/2,3`. The gen_fsm then continues executing with the current state name set to `NextStateName` and with the possibly updated state data `NewStateData`. See `Module:init/1` for a description of `Timeout` and `hibernate`.

If the function returns {`next_state`,`NextStateName`,`NewStateData`}, {`next_state`,`NextStateName`,`NewStateData`,`Timeout`} or {`next_state`,`NextStateName`,`NewStateData`,`hibernate`}, the gen_fsm will continue executing in `NextStateName` with `NewStateData`. Any reply to `From` must be given explicitly using gen_fsm:reply/2 [page 202].

If the function returns {`stop`,`Reason`,`Reply`,`NewStateData`}, `Reply` will be given back to `From`. If the function returns {`stop`,`Reason`,`NewStateData`}, any reply to `From` must be given explicitly using `gen_fsm:reply/2`. The gen_fsm will then call `Module:terminate(Reason,NewStateData)` and terminate.

`Module:handle_sync_event(Event, From, StateName, StateData) -> Result`

Types:

- Event = term()
- From = {pid(),Tag}
- StateName = atom()
- StateData = term()
- Result = {reply,Reply,NextStateName,NewStateData}
- | {reply,Reply,NextStateName,NewStateData,Timeout}
- | {reply,Reply,NextStateName,NewStateData,hibernate}
- | {next_state,NextStateName,NewStateData}
- | {next_state,NextStateName,NewStateData,Timeout}

- | {next_state,NextStateName,NewStateData,hibernate}
- | {stop,Reason,Reply,NewStateData} | {stop,Reason,NewStateData}
- Reply = term()
- NextStateName = atom()
- NewStateData = term()
- Timeout = int()>0 | infinity
- Reason = term()

Whenever a gen_fsm receives an event sent using gen_fsm:sync_send_all_state_event/2,3 [page 201], this function is called to handle the event.

`StateName` is the current state name of the gen_fsm.

See `Module:StateName/3` for a description of the other arguments and possible return values.

`Module:handle_info(Info, StateName, StateData) -> Result`

Types:

- Info = term()
- StateName = atom()
- StateData = term()
- Result = {next_state,NextStateName,NewStateData}
- >| {next_state,NextStateName,NewStateData,Timeout}
- >| {next_state,NextStateName,NewStateData,hibernate}
- >| {stop,Reason,NewStateData}
- NextStateName = atom()
- NewStateData = term()
- Timeout = int()>0 | infinity
- Reason = normal | term()

This function is called by a gen_fsm when it receives any other message than a synchronous or asynchronous event (or a system message).

`Info` is the received message.

See `Module:StateName/2` for a description of the other arguments and possible return values.

`Module:terminate(Reason, StateName, StateData)`

Types:

- Reason = normal | shutdown | term()
- StateName = atom()
- StateData = term()

This function is called by a gen_fsm when it is about to terminate. It should be the opposite of `Module:init/1` and do any necessary cleaning up. When it returns, the gen_fsm terminates with `Reason`. The return value is ignored.

`Reason` is a term denoting the stop reason, `StateName` is the current state name, and `StateData` is the state data of the gen_fsm.

`Reason` depends on why the gen_fsm is terminating. If it is because another callback function has returned a stop tuple `{stop,..}`, `Reason` will have the value specified in that tuple. If it is due to a failure, `Reason` is the error reason.

If the gen_fsm is part of a supervision tree and is ordered by its supervisor to terminate, this function will be called with `Reason=shutdown` if the following conditions apply:

- the gen_fsm has been set to trap exit signals, and
- the shutdown strategy as defined in the supervisor's child specification is an integer timeout value, not `brutal_kill`.

Otherwise, the gen_fsm will be immediately terminated.

Note that for any other reason than `normal` or `shutdown`, the gen_fsm is assumed to terminate due to an error and an error report is issued using [error_logger:format/2].

`Module:code_change(OldVsn, StateName, StateData, Extra) -> {ok, NextStateName, NewStateData}`

Types:

- OldVsn = Vsn | {down, Vsn}
- Vsn = term()
- StateName = NextStateName = atom()
- StateData = NewStateData = term()
- Extra = term()

This function is called by a gen_fsm when it should update its internal state data during a release upgrade/downgrade, i.e. when the instruction `{update,Module,Change,...}` where `Change={advanced,Extra}` is given in the appup file. See [OTP Design Principles].

In the case of an upgrade, `OldVsn` is `Vsn`, and in the case of a downgrade, `OldVsn` is `{down,Vsn}`. `Vsn` is defined by the `vsn` attribute(s) of the old version of the callback module `Module`. If no such attribute is defined, the version is the checksum of the BEAM file.

`StateName` is the current state name and `StateData` the internal state data of the gen_fsm.

`Extra` is passed as-is from the `{advanced,Extra}` part of the update instruction.

The function should return the new current state name and updated internal data.

SEE ALSO

gen_event(3) [page 188], gen_server(3) [page 209], supervisor(3) [page 401], proc_lib(3) [page 277], sys(3) [page 411]

gen_server

Erlang Module

A behaviour module for implementing the server of a client-server relation. A generic server process (gen_server) implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into an OTP supervision tree. Refer to [OTP Design Principles] for more information.

A gen_server assumes all specific parts to be located in a callback module exporting a pre-defined set of functions. The relationship between the behaviour functions and the callback functions can be illustrated as follows:

```
gen_server module            Callback module
-----------------            ---------------
gen_server:start_link -----> Module:init/1

gen_server:call
gen_server:multi_call -----> Module:handle_call/3

gen_server:cast
gen_server:abcast     -----> Module:handle_cast/2

-                     -----> Module:handle_info/2

-                     -----> Module:terminate/2

-                     -----> Module:code_change/3
```

If a callback function fails or returns a bad value, the gen_server will terminate.

The `sys` module can be used for debugging a gen_server.

Note that a gen_server does not trap exit signals automatically, this must be explicitly initiated in the callback module.

Unless otherwise stated, all functions in this module fail if the specified gen_server does not exist or if bad arguments are given.

The gen_server process can go into hibernation (see [erlang(3)]) if a callback function specifies `'hibernate'` instead of a timeout value. This might be useful if the server is expected to be idle for a long time. However this feature should be used with care as hibernation implies at least two garbage collections (when hibernating and shortly after waking up) and is not something you'd want to do between each call to a busy server.

Exports

```
start_link(Module, Args, Options) -> Result
start_link(ServerName, Module, Args, Options) -> Result
```

Types:

- ServerName = {local,Name} | {global,GlobalName}
- Name = atom()
- GlobalName = term()
- Module = atom()
- Args = term()
- Options = [Option]
- Option = {debug,Dbgs} | {timeout,Time} | {spawn_opt,SOpts}
- Dbgs = [Dbg]
- Dbg = trace | log | statistics | {log_to_file,FileName} | {install,{Func,FuncState}}
- SOpts = [term()]
- Result = {ok,Pid} | ignore | {error,Error}
- Pid = pid()
- Error = {already_started,Pid} | term()

Creates a gen_server process as part of a supervision tree. The function should be called, directly or indirectly, by the supervisor. It will, among other things, ensure that the gen_server is linked to the supervisor.

The gen_server process calls `Module:init/1` to initialize. To ensure a synchronized start-up procedure, `start_link/3,4` does not return until `Module:init/1` has returned.

If `ServerName={local,Name}` the gen_server is registered locally as `Name` using `register/2`. If `ServerName={global,GlobalName}` the gen_server is registered globally as `GlobalName` using `global:register_name/2`. If no name is provided, the gen_server is not registered.

`Module` is the name of the callback module.

`Args` is an arbitrary term which is passed as the argument to `Module:init/1`.

If the option `{timeout,Time}` is present, the gen_server is allowed to spend `Time` milliseconds initializing or it will be terminated and the start function will return `{error,timeout}`.

If the option `{debug,Dbgs}` is present, the corresponding `sys` function will be called for each item in `Dbgs`. See sys(3) [page 411].

If the option `{spawn_opt,SOpts}` is present, `SOpts` will be passed as option list to the `spawn_opt` BIF which is used to spawn the gen_server. See [erlang(3)].

> **Note:**
> Using the spawn option `monitor` is currently not allowed, but will cause the function to fail with reason `badarg`.

If the gen_server is successfully created and initialized the function returns `{ok,Pid}`, where `Pid` is the pid of the gen_server. If there already exists a process with the

specified ServerName the function returns {error,{already_started,Pid}}, where Pid is the pid of that process.

If Module:init/1 fails with Reason, the function returns {error,Reason}. If Module:init/1 returns {stop,Reason} or ignore, the process is terminated and the function returns {error,Reason} or ignore, respectively.

```
start(Module, Args, Options) -> Result
start(ServerName, Module, Args, Options) -> Result
```

Types:

- ServerName = {local,Name} | {global,GlobalName}
- Name = atom()
- GlobalName = term()
- Module = atom()
- Args = term()
- Options = [Option]
- Option = {debug,Dbgs} | {timeout,Time} | {spawn_opt,SOpts}
- Dbgs = [Dbg]
- Dbg = trace | log | statistics | {log_to_file,FileName} | {install,{Func,FuncState}}
- SOpts = [term()]
- Result = {ok,Pid} | ignore | {error,Error}
- Pid = pid()
- Error = {already_started,Pid} | term()

Creates a stand-alone gen_server process, i.e. a gen_server which is not part of a supervision tree and thus has no supervisor.

See start_link/3,4 [page 210] for a description of arguments and return values.

```
call(ServerRef, Request) -> Reply
call(ServerRef, Request, Timeout) -> Reply
```

Types:

- ServerRef = Name | {Name,Node} | {global,GlobalName} | pid()
- Node = atom()
- GlobalName = term()
- Request = term()
- Timeout = int()>0 | infinity
- Reply = term()

Makes a synchronous call to the gen_server ServerRef by sending a request and waiting until a reply arrives or a timeout occurs. The gen_server will call Module:handle_call/3 to handle the request.

ServerRef can be:

- the pid,
- Name, if the gen_server is locally registered,
- {Name,Node}, if the gen_server is locally registered at another node, or
- {global,GlobalName}, if the gen_server is globally registered.

Request is an arbitrary term which is passed as one of the arguments to Module:handle_call/3.

Timeout is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom infinity to wait indefinitely. Default value is 5000. If no reply is received within the specified time, the function call fails.

The return value Reply is defined in the return value of Module:handle_call/3.

The call may fail for several reasons, including timeout and the called gen_server dying before or during the call.

The ancient behaviour of sometimes consuming the server exit message if the server died during the call while linked to the client has been removed in OTP R12B/Erlang 5.6.

```
multi_call(Name, Request) -> Result
multi_call(Nodes, Name, Request) -> Result
multi_call(Nodes, Name, Request, Timeout) -> Result
```

Types:

- Nodes = [Node]
- Node = atom()
- Name = atom()
- Request = term()
- Timeout = int()>=0 | infinity
- Result = {Replies,BadNodes}
- Replies = [{Node,Reply}]
- Reply = term()
- BadNodes = [Node]

Makes a synchronous call to all gen_servers locally registered as Name at the specified nodes by first sending a request to every node and then waiting for the replies. The gen_servers will call Module:handle_call/3 to handle the request.

The function returns a tuple {Replies,BadNodes} where Replies is a list of {Node,Reply} and BadNodes is a list of node that either did not exist, or where the gen_server Name did not exist or did not reply.

Nodes is a list of node names to which the request should be sent. Default value is the list of all known nodes [node()|nodes()].

Name is the locally registered name of each gen_server.

Request is an arbitrary term which is passed as one of the arguments to Module:handle_call/3.

Timeout is an integer greater than zero which specifies how many milliseconds to wait for each reply, or the atom infinity to wait indefinitely. Default value is infinity. If no reply is received from a node within the specified time, the node is added to BadNodes.

When a reply Reply is received from the gen_server at a node Node, {Node,Reply} is added to Replies. Reply is defined in the return value of Module:handle_call/3.

> **Warning:**
> If one of the nodes is not capable of process monitors, for example C or Java nodes, and the gen_server is not started when the requests are sent, but starts within 2 seconds, this function waits the whole `Timeout`, which may be infinity.
>
> This problem does not exist if all nodes are Erlang nodes.

To avoid that late answers (after the timeout) pollutes the caller's message queue, a middleman process is used to do the actual calls. Late answers will then be discarded when they arrive to a terminated process.

```
cast(ServerRef, Request) -> ok
```

Types:

- ServerRef = Name | {Name,Node} | {global,GlobalName} | pid()
- Node = atom()
- GlobalName = term()
- Request = term()

Sends an asynchronous request to the gen_server `ServerRef` and returns `ok` immediately, ignoring if the destination node or gen_server does not exist. The gen_server will call `Module:handle_cast/2` to handle the request.

See call/2,3 [page 211] for a description of `ServerRef`.

`Request` is an arbitrary term which is passed as one of the arguments to `Module:handle_cast/2`.

```
abcast(Name, Request) -> abcast
abcast(Nodes, Name, Request) -> abcast
```

Types:

- Nodes = [Node]
- Node = atom()
- Name = atom()
- Request = term()

Sends an asynchronous request to the gen_servers locally registered as `Name` at the specified nodes. The function returns immediately and ignores nodes that do not exist, or where the gen_server `Name` does not exist. The gen_servers will call `Module:handle_cast/2` to handle the request.

See multi_call/2,3,4 [page 212] for a description of the arguments.

```
reply(Client, Reply) -> Result
```

Types:

- Client - see below
- Reply = term()
- Result = term()

This function can be used by a gen_server to explicitly send a reply to a client that called `call/2,3` or `multi_call/2,3,4`, when the reply cannot be defined in the return value of `Module:handle_call/3`.

`Client` must be the `From` argument provided to the callback function. `Reply` is an arbitrary term, which will be given back to the client as the return value of `call/2,3` or `multi_call/2,3,4`.

The return value `Result` is not further defined, and should always be ignored.

```
enter_loop(Module, Options, State)
enter_loop(Module, Options, State, ServerName)
enter_loop(Module, Options, State, Timeout)
enter_loop(Module, Options, State, ServerName, Timeout)
```

Types:

- Module = atom()
- Options = [Option]
- Option = {debug,Dbgs}
- Dbgs = [Dbg]
- Dbg = trace | log | statistics
- | {log_to_file,FileName} | {install,{Func,FuncState}}
- State = term()
- ServerName = {local,Name} | {global,GlobalName}
- Name = atom()
- GlobalName = term()
- Timeout = int() | infinity

Makes an existing process into a gen_server. Does not return, instead the calling process will enter the gen_server receive loop and become a gen_server process. The process *must* have been started using one of the start functions in `proc_lib`, see proc_lib(3) [page 277]. The user is responsible for any initialization of the process, including registering a name for it.

This function is useful when a more complex initialization procedure is needed than the gen_server behaviour provides.

`Module`, `Options` and `ServerName` have the same meanings as when calling gen_server:start[_link]/3,4 [page 210]. However, if `ServerName` is specified, the process must have been registered accordingly *before* this function is called.

`State` and `Timeout` have the same meanings as in the return value of Module:init/1 [page 215]. Also, the callback module `Module` does not need to export an `init/1` function.

Failure: If the calling process was not started by a `proc_lib` start function, or if it is not registered according to `ServerName`.

CALLBACK FUNCTIONS

The following functions should be exported from a `gen_server` callback module.

Exports

`Module:init(Args) -> Result`

Types:

- Args = term()
- Result = {ok,State} | {ok,State,Timeout} | {ok,State,hibernate}
- | {stop,Reason} | ignore
- State = term()
- Timeout = int()>=0 | infinity
- Reason = term()

Whenever a gen_server is started using gen_server:start/3,4 [page 211] or gen_server:start_link/3,4 [page 210], this function is called by the new process to initialize.

`Args` is the `Args` argument provided to the start function.

If the initialization is successful, the function should return {`ok`,`State`}, {`ok`,`State`,`Timeout`} or {`ok`,`State`,`hibernate`}, where `State` is the internal state of the gen_server.

If an integer timeout value is provided, a timeout will occur unless a request or a message is received within `Timeout` milliseconds. A timeout is represented by the atom `timeout` which should be handled by the `handle_info/2` callback function. The atom `infinity` can be used to wait indefinitely, this is the default value.

If `hibernate` is specified instead of a timeout value, the process will go into hibernation when waiting for the next message to arrive (by calling proc_lib:hibernate/3 [page 281]).

If something goes wrong during the initialization the function should return {`stop`,`Reason`} where `Reason` is any term, or `ignore`.

`Module:handle_call(Request, From, State) -> Result`

Types:

- Request = term()
- From = {pid(),Tag}
- State = term()
- Result = {reply,Reply,NewState} | {reply,Reply,NewState,Timeout}
- | {reply,Reply,NewState,hibernate}
- | {noreply,NewState} | {noreply,NewState,Timeout}
- | {noreply,NewState,hibernate}
- | {stop,Reason,Reply,NewState} | {stop,Reason,NewState}
- Reply = term()
- NewState = term()
- Timeout = int()>=0 | infinity
- Reason = term()

Whenever a gen_server receives a request sent using gen_server:call/2,3 [page 211] or gen_server:multi_call/2,3,4 [page 212], this function is called to handle the request.

`Request` is the `Request` argument provided to `call` or `multi_call`.

`From` is a tuple `{Pid,Tag}` where `Pid` is the pid of the client and `Tag` is a unique tag.

`State` is the internal state of the gen_server.

If the function returns `{reply,Reply,NewState}`, `{reply,Reply,NewState,Timeout}` or `{reply,Reply,NewState,hibernate}`, `Reply` will be given back to `From` as the return value of `call/2,3` or included in the return value of `multi_call/2,3,4`. The gen_server then continues executing with the possibly updated internal state `NewState`. See `Module:init/1` for a description of `Timeout` and `hibernate`.

If the functions returns `{noreply,NewState}`, `{noreply,NewState,Timeout}` or `{noreply,NewState,hibernate}`, the gen_server will continue executing with `NewState`. Any reply to `From` must be given explicitly using gen_server:reply/2 [page 213].

If the function returns `{stop,Reason,Reply,NewState}`, `Reply` will be given back to `From`. If the function returns `{stop,Reason,NewState}`, any reply to `From` must be given explicitly using `gen_server:reply/2`. The gen_server will then call `Module:terminate(Reason,NewState)` and terminate.

`Module:handle_cast(Request, State) -> Result`

Types:

- Request = term()
- State = term()
- Result = {noreply,NewState} | {noreply,NewState,Timeout}
- | {noreply,NewState,hibernate}
- | {stop,Reason,NewState}
- NewState = term()
- Timeout = int()>=0 | infinity
- Reason = term()

Whenever a gen_server receives a request sent using gen_server:cast/2 [page 213] or gen_server:abcast/2,3 [page 213], this function is called to handle the request.

See `Module:handle_call/3` for a description of the arguments and possible return values.

`Module:handle_info(Info, State) -> Result`

Types:

- Info = timeout | term()
- State = term()
- Result = {noreply,NewState} | {noreply,NewState,Timeout}
- | {noreply,NewState,hibernate}
- | {stop,Reason,NewState}
- NewState = term()
- Timeout = int()>=0 | infinity
- Reason = normal | term()

This function is called by a gen_server when a timeout occurs or when it receives any other message than a synchronous or asynchronous request (or a system message).

`Info` is either the atom `timeout`, if a timeout has occurred, or the received message.

See `Module:handle_call/3` for a description of the other arguments and possible return values.

`Module:terminate(Reason, State)`

Types:

- Reason = normal | shutdown | term()
- State = term()

This function is called by a gen_server when it is about to terminate. It should be the opposite of `Module:init/1` and do any necessary cleaning up. When it returns, the gen_server terminates with `Reason`. The return value is ignored.

`Reason` is a term denoting the stop reason and `State` is the internal state of the gen_server.

`Reason` depends on why the gen_server is terminating. If it is because another callback function has returned a stop tuple `{stop,..}`, `Reason` will have the value specified in that tuple. If it is due to a failure, `Reason` is the error reason.

If the gen_server is part of a supervision tree and is ordered by its supervisor to terminate, this function will be called with `Reason=shutdown` if the following conditions apply:

- the gen_server has been set to trap exit signals, and
- the shutdown strategy as defined in the supervisor's child specification is an integer timeout value, not `brutal_kill`.

Otherwise, the gen_server will be immediately terminated.

Note that for any other reason than `normal` or `shutdown`, the gen_server is assumed to terminate due to an error and an error report is issued using [error_logger:format/2].

`Module:code_change(OldVsn, State, Extra) -> {ok, NewState}`

Types:

- OldVsn = Vsn | {down, Vsn}
- Vsn = term()
- State = NewState = term()
- Extra = term()

This function is called by a gen_server when it should update its internal state during a release upgrade/downgrade, i.e. when the instruction `{update,Module,Change,...}` where `Change={advanced,Extra}` is given in the appup file. See [OTP Design Principles] for more information.

In the case of an upgrade, `OldVsn` is `Vsn`, and in the case of a downgrade, `OldVsn` is `{down,Vsn}`. `Vsn` is defined by the `vsn` attribute(s) of the old version of the callback module `Module`. If no such attribute is defined, the version is the checksum of the BEAM file.

`State` is the internal state of the gen_server.

`Extra` is passed as-is from the `{advanced,Extra}` part of the update instruction.

The function should return the updated internal state.

SEE ALSO

gen_event(3) [page 188], gen_fsm(3) [page 198], supervisor(3) [page 401], proc_lib(3) [page 277], sys(3) [page 411]

io

Erlang Module

This module provides an interface to standard Erlang IO servers. The output functions all return `ok` if they are successful, or exit if they are not.

In the following description, all functions have an optional parameter `IoDevice`. If included, it must be the pid of a process which handles the IO protocols. Normally, it is the `IoDevice` returned by [file:open/2].

For a description of the IO protocols refer to Armstrong, Virding and Williams, 'Concurrent Programming in Erlang', Chapter 13, unfortunately now very outdated, but the general principles still apply.

DATA TYPES

```
io_device()
  as returned by file:open/2, a process handling IO protocols
```

Exports

`columns([IoDevice]) -> {ok,int()} | {error, enotsup}`

Types:

- IoDevice = io_device()

Retrieves the number of columns of the `IoDevice` (i.e. the width of a terminal). The function only succeeds for terminal devices, for all other devices the function returns `{error, enotsup}`

`put_chars([IoDevice,] IoData) -> ok`

Types:

- IoDevice = io_device()
- IoData = iodata() – see erlang(3)

Writes the characters of `IoData` to the standard output (`IoDevice`).

`nl([IoDevice]) -> ok`

Types:

- IoDevice = io_device()

Writes new line to the standard output (`IoDevice`).

`get_chars([IoDevice,] Prompt, Count) -> string() | eof`

Types:

- IoDevice = io_device()
- Prompt = atom() | string()
- Count = int()

Reads `Count` characters from standard input (`IoDevice`), prompting it with `Prompt`. It returns:

`String` The input characters.

`eof` End of file was encountered.

`get_line([IoDevice,] Prompt) -> string() | eof`

Types:

- IoDevice = io_device()
- Prompt = atom() | string()

Reads a line from the standard input (`IoDevice`), prompting it with `Prompt`. It returns:

`String` The characters in the line terminated by a LF (or end of file).

`eof` End of file was encountered.

`setopts([IoDevice,] Opts) -> ok | {error, Reason}`

Types:

- IoDevice = io_device()
- Opts = [Opt]
- Opt = binary | list
- Reason = term()

Set options for standard input/output (`IoDevice`). Possible options are:

`binary` Makes `get_chars/2,3` and `get_line/1,2` return binaries instead of lists of chars.

`list` Makes `get_chars/2,3` and `get_line/1,2` return lists of chars, which is the default.

`expand_fun` Provide a function for tab-completion (expansion) like the erlang shell. This function is called when the user presses the Tab key. The expansion is active when calling line-reading functions such as `get_line/1,2`.

The function is called with the current line, upto the cursor, as a reversed string. It should return a three-tuple: `{yes|no, string(), [string(), ...]}`. The first element gives a beep if `no`, otherwise the expansion is silent, the second is a string that will be entered at the cursor position, and the third is a list of possible expansions. If this list is non-empty, the list will be printed and the current input line will be written once again.

Trivial example (beep on anything except empty line, which is expanded to “quit”):

```
fun("") -> {yes, "quit", []};
   (_) -> {no, "", ["quit"]} end
```

> **Note:**
> The `binary` option does not work against IO servers on remote nodes running an older version of Erlang/OTP than R9C.

`write([IoDevice,] Term) -> ok`

Types:

- IoDevice = io_device()
- Term = term()

Writes the term `Term` to the standard output (`IoDevice`).

`read([IoDevice,] Prompt) -> Result`

Types:

- IoDevice = io_device()
- Prompt = atom() | string()
- Result = {ok, Term} | eof | {error, ErrorInfo}
- Term = term()
- ErrorInfo – see section Error Information below

Reads a term `Term` from the standard input (`IoDevice`), prompting it with `Prompt`. It returns:

`{ok, Term}` The parsing was successful.

`eof` End of file was encountered.

`{error, ErrorInfo}` The parsing failed.

`read(IoDevice, Prompt, StartLine) -> Result`

Types:

- IoDevice = io_device()
- Prompt = atom() | string()
- StartLine = int()
- Result = {ok, Term, EndLine} | {eof, EndLine} | {error, ErrorInfo, EndLine}
- Term = term()
- EndLine = int()
- ErrorInfo – see section Error Information below

Reads a term `Term` from `IoDevice`, prompting it with `Prompt`. Reading starts at line number `StartLine`. It returns:

`{ok, Term, EndLine}` The parsing was successful.

`{eof, EndLine}` End of file was encountered.

`{error, ErrorInfo, EndLine}` The parsing failed.

`fwrite(Format) ->`
`fwrite([IoDevice,] Format, Data) -> ok`

```
format(Format) ->
format([IoDevice,] Format, Data) -> ok
```

Types:

- IoDevice = io_device()
- Format = atom() | string() | binary()
- Data = [term()]

Writes the items in `Data` (`[]`) on the standard output (`IoDevice`) in accordance with `Format`. `Format` contains plain characters which are copied to the output device, and control sequences for formatting, see below. If `Format` is an atom or a binary, it is first converted to a list with the aid of `atom_to_list/1` or `binary_to_list/1`.

```
1> io:fwrite("Hello world!~n", []).
Hello world!
ok
```

The general format of a control sequence is `~F.P.PadC`. The character `C` determines the type of control sequence to be used, `F` and `P` are optional numeric arguments. If `F`, `P`, or `Pad` is `*`, the next argument in `Data` is used as the numeric value of `F` or `P`.

`F` is the `field width` of the printed argument. A negative value means that the argument will be left justified within the field, otherwise it will be right justified. If no field width is specified, the required print width will be used. If the field width specified is too small, then the whole field will be filled with `*` characters.

`P` is the `precision` of the printed argument. A default value is used if no precision is specified. The interpretation of precision depends on the control sequences. Unless otherwise specified, the argument `within` is used to determine print width.

`Pad` is the padding character. This is the character used to pad the printed representation of the argument so that it conforms to the specified field width and precision. Only one padding character can be specified and, whenever applicable, it is used for both the field width and precision. The default padding character is `' '` (space).

The following control sequences are available:

`~` The character `~` is written.

`c` The argument is a number that will be interpreted as an ASCII code. The precision is the number of times the character is printed and it defaults to the field width, which in turn defaults to 1. The following example illustrates:

```
2> io:fwrite("|~10.5c|~-10.5c|~5c|~n", [$a, $b, $c]).
|     aaaaa|bbbbb     |ccccc|
ok
```

`f` The argument is a float which is written as `[-]ddd.ddd`, where the precision is the number of digits after the decimal point. The default precision is 6 and it cannot be less than 1.

`e` The argument is a float which is written as `[-]d.ddde+-ddd`, where the precision is the number of digits written. The default precision is 6 and it cannot be less than 2.

`g` The argument is a float which is written as `f`, if it is >= 0.1 and < 10000.0. Otherwise, it is written in the `e` format. The precision is the number of significant digits. It defaults to 6 and should not be less than 2. If the absolute value of the float does not allow it to be written in the `f` format with the desired number of significant digits, it is also written in the `e` format.

`s` Prints the argument with the `string` syntax. The argument is an [I/O list], a binary, or an atom. The characters are printed without quotes. In this format, the printed argument is truncated to the given precision and field width.

This format can be used for printing any object and truncating the output so it fits a specified field:

```
3> io:fwrite("|~10w|~n", [{hey, hey, hey}]).
|**********|
ok
4> io:fwrite("|~10s|~n", [io_lib:write({hey, hey, hey})]).
|{hey,hey,h|
ok
```

`w` Writes data with the standard syntax. This is used to output Erlang terms. Atoms are printed within quotes if they contain embedded non-printable characters, and floats are printed accurately as the shortest, correctly rounded string.

`p` Writes the data with standard syntax in the same way as `~w`, but breaks terms whose printed representation is longer than one line into many lines and indents each line sensibly. It also tries to detect lists of printable characters and to output these as strings. For example:

```
5> T = [{attributes,[[{id,age,1.50000},{mode,explicit},
{typename,"INTEGER"}], [{id,cho},{mode,explicit},{typename,'Cho'}]]},
{typename,'Person'},{tag,{'PRIVATE',3}},{mode,implicit}].
...
6> io:fwrite("~w~n", [T]).
[{attributes,[[{id,age,1.5},{mode,explicit},{typename,
[73,78,84,69,71,69,82]}],[{id,cho},{mode,explicit},{typena
me,'Cho'}]]},{typename,'Person'},{tag,{'PRIVATE',3}},{mode
,implicit}]
ok
7> io:fwrite("~62p~n", [T]).
[{attributes,[[{id,age,1.5},
               {mode,explicit},
               {typename,"INTEGER"}],
              [{id,cho},{mode,explicit},{typename,'Cho'}]]},
 {typename,'Person'},
 {tag,{'PRIVATE',3}},
 {mode,implicit}]
ok
```

The field width specifies the maximum line length. It defaults to 80. The precision specifies the initial indentation of the term. It defaults to the number of characters printed on this line in the same call to `io:fwrite` or `io:format`. For example, using `T` above:

```
8> io:fwrite("Here T = ~62p~n", [T]).
Here T = [{attributes,[[{id,age,1.5},
                        {mode,explicit},
                        {typename,"INTEGER"}],
                       [{id,cho},
```

```
                              {mode,explicit},
                              {typename,'Cho'}]]},
                    {typename,'Person'},
                    {tag,{'PRIVATE',3}},
                    {mode,implicit}]
ok
```

`W` Writes data in the same way as `~w`, but takes an extra argument which is the maximum depth to which terms are printed. Anything below this depth is replaced with `...`. For example, using `T` above:

```
9> io:fwrite("~W~n", [T,9]).
[{attributes,[[{id,age,1.5},{mode,explicit},{typename,...}],
[{id,cho},{mode,...},{...}]]},{typename,'Person'},
{tag,{'PRIVATE',3}},{mode,implicit}]
ok
```

If the maximum depth has been reached, then it is impossible to read in the resultant output. Also, the `,...` form in a tuple denotes that there are more elements in the tuple but these are below the print depth.

`P` Writes data in the same way as `~p`, but takes an extra argument which is the maximum depth to which terms are printed. Anything below this depth is replaced with `...`. For example:

```
10> io:fwrite("~62P~n", [T,9]).
[{attributes,[[{id,age,1.5},{mode,explicit},{typename,...}],
              [{id,cho},{mode,...},{...}]]},
 {typename,'Person'},
 {tag,{'PRIVATE',3}},
 {mode,implicit}]
ok
```

`B` Writes an integer in base 2..36, the default base is 10. A leading dash is printed for negative integers.

The precision field selects base. For example:

```
11> io:fwrite("~.16B~n", [31]).
1F
ok
12> io:fwrite("~.2B~n", [-19]).
-10011
ok
13> io:fwrite("~.36B~n", [5*36+35]).
5Z
ok
```

`X` Like `B`, but takes an extra argument that is a prefix to insert before the number, but after the leading dash, if any.

The prefix can be a possibly deep list of characters or an atom.

```
14> io:fwrite("~X~n", [31,"10#"]).
10#31
ok
15> io:fwrite("~.16X~n", [-31,"0x"]).
-0x1F
ok
```

`#` Like `B`, but prints the number with an Erlang style '#'-separated base prefix.

```
16> io:fwrite("~.10#~n", [31]).
10#31
ok
17> io:fwrite("~.16#~n", [-31]).
-16#1F
ok
```

`b` Like `B`, but prints lowercase letters.

`x` Like `X`, but prints lowercase letters.

`+` Like `#`, but prints lowercase letters.

`n` Writes a new line.

`i` Ignores the next term.

Returns:

`ok` The formatting succeeded.

If an error occurs, there is no output. For example:

```
18> io:fwrite("~s ~w ~i ~w ~c ~n",['abc def', 'abc def', {foo, 1},{foo, 1}, 65]).
abc def 'abc def'  {foo,1} A
ok
19> io:fwrite("~s", [65]).
** exception exit: {badarg,[{io,format,[<0.22.0>,"~s","A"]},
                            {erl_eval,do_apply,5},
                            {shell,exprs,6},
                            {shell,eval_exprs,6},
                            {shell,eval_loop,3}]}
     in function  io:o_request/2
```

In this example, an attempt was made to output the single character '65' with the aid of the string formatting directive "~s".

`fread([IoDevice,] Prompt, Format) -> Result`

Types:

- IoDevice = io_device()
- Prompt = atom() | string()
- Format = string()
- Result = {ok, Terms} | eof | {error, What}
- Terms = [term()]
- What = term()

Reads characters from the standard input (`IoDevice`), prompting it with `Prompt`. Interprets the characters in accordance with `Format`. `Format` contains control sequences which directs the interpretation of the input.

`Format` may contain:

- White space characters (SPACE, TAB and NEWLINE) which cause input to be read to the next non-white space character.
- Ordinary characters which must match the next input character.
- Control sequences, which have the general format `~*FC`. The character `*` is an optional return suppression character. It provides a method to specify a field which is to be omitted. `F` is the `field width` of the input field and `C` determines the type of control sequence.

  Unless otherwise specified, leading white-space is ignored for all control sequences. An input field cannot be more than one line wide. The following control sequences are available:

  `~` A single `~` is expected in the input.

  `d` A decimal integer is expected.

  `u` An unsigned integer in base 2..36 is expected. The field width parameter is used to specify base. Leading white-space characters are not skipped.

  `-` An optional sign character is expected. A sign character '-' gives the return value `-1`. Sign character '+' or none gives `1`. The field width parameter is ignored. Leading white-space characters are not skipped.

  `#` An integer in base 2..36 with Erlang-style base prefix (for example `"16#ffff"`) is expected.

  `f` A floating point number is expected. It must follow the Erlang floating point number syntax.

  `s` A string of non-white-space characters is read. If a field width has been specified, this number of characters are read and all trailing white-space characters are stripped. An Erlang string (list of characters) is returned.

  `a` Similar to `s`, but the resulting string is converted into an atom.

  `c` The number of characters equal to the field width are read (default is 1) and returned as an Erlang string. However, leading and trailing white-space characters are not omitted as they are with `s`. All characters are returned.

  `l` Returns the number of characters which have been scanned up to that point, including white-space characters.

  It returns:

  `{ok, Terms}` The read was successful and `Terms` is the list of successfully matched and read items.

  `eof` End of file was encountered.

  `{error, What}` The read operation failed and the parameter `What` gives a hint about the error.

Examples:

```
20> io:fread('enter>', "~f~f~f").
enter>1.9 35.5e3 15.0
{ok,[1.9,3.55e4,15.0]}
21> io:fread('enter>', "~10f~d").
enter>      5.67899
{ok,[5.678,99]}
```

```
22> io:fread('enter>', ":~10s:~10c:").
enter>:   alan   :   joe    :
{ok, ["alan", "   joe    "]}
```

```
rows([IoDevice]) -> {ok,int()} | {error, enotsup}
```

Types:

- IoDevice = io_device()

Retrieves the number of rows of the `IoDevice` (i.e. the height of a terminal). The function only succeeds for terminal devices, for all other devices the function returns `{error, enotsup}`

```
scan_erl_exprs(Prompt) ->
scan_erl_exprs([IoDevice,] Prompt, StartLine) -> Result
```

Types:

- IoDevice = io_device()
- Prompt = atom() | string()
- StartLine = int()
- Result = {ok, Tokens, EndLine} | {eof, EndLine} | {error, ErrorInfo, EndLine}
- Tokens – see erl_scan(3)
- EndLine = int()
- ErrorInfo – see section Error Information below

Reads data from the standard input (`IoDevice`), prompting it with `Prompt`. Reading starts at line number `StartLine` (1). The data is tokenized as if it were a sequence of Erlang expressions until a final `'.'` is reached. This token is also returned. It returns:

`{ok, Tokens, EndLine}` The tokenization succeeded.

`{eof, EndLine}` End of file was encountered.

`{error, ErrorInfo, EndLine}` An error occurred.

Example:

```
23> io:scan_erl_exprs('enter>').
enter>abc(), "hey".
{ok,[{atom,1,abc},{'(',1},{')',1},{',',1},{string,1,"hey"},{dot,1}],2}
24> io:scan_erl_exprs('enter>').
enter>1.0er.
{error,{1,erl_scan,{illegal,float}},2}
```

```
scan_erl_form(Prompt) ->
scan_erl_form([IoDevice,] Prompt, StartLine) -> Result
```

Types:

- IoDevice = io_device()
- Prompt = atom() | string()
- StartLine = int()
- Result = {ok, Tokens, EndLine} | {eof, EndLine} | {error, ErrorInfo, EndLine}
- Tokens – see erl_scan(3)
- EndLine = int()

- ErrorInfo – see section Error Information below

Reads data from the standard input (IoDevice), prompting it with Prompt. Starts reading at line number StartLine (1). The data is tokenized as if it were an Erlang form - one of the valid Erlang expressions in an Erlang source file - until a final '.' is reached. This last token is also returned. The return values are the same as for scan_erl_exprs/1,2,3 above.

```
parse_erl_exprs(Prompt) ->
parse_erl_exprs([IoDevice,] Prompt, StartLine) -> Result
```

Types:

- IoDevice = io_device()
- Prompt = atom() | string()
- StartLine = int()
- Result = {ok, Expr_list, EndLine} | {eof, EndLine} | {error, ErrorInfo, EndLine}
- Expr_list – see erl_parse(3)
- EndLine = int()
- ErrorInfo – see section Error Information below

Reads data from the standard input (IoDevice), prompting it with Prompt. Starts reading at line number StartLine (1). The data is tokenized and parsed as if it were a sequence of Erlang expressions until a final '.' is reached. It returns:

{ok, Expr_list, EndLine} The parsing was successful.

{eof, EndLine} End of file was encountered.

{error, ErrorInfo, EndLine} An error occurred.

Example:

```
25> io:parse_erl_exprs('enter>').
enter>abc(), "hey".
{ok, [{call,1,{atom,1,abc},[]},{string,1,"hey"}],2}
26> io:parse_erl_exprs ('enter>').
enter>abc("hey".
{error,{1,erl_parse,["syntax error before: ",["'.'"]]},2}
```

```
parse_erl_form(Prompt) ->
parse_erl_form([IoDevice,] Prompt, StartLine) -> Result
```

Types:

- IoDevice = io_device()
- Prompt = atom() | string()
- StartLine = int()
- Result = {ok, AbsForm, EndLine} | {eof, EndLine} | {error, ErrorInfo, EndLine}
- AbsForm – see erl_parse(3)
- EndLine = int()
- ErrorInfo – see section Error Information below

Reads data from the standard input (IoDevice), prompting it with Prompt. Starts reading at line number StartLine (1). The data is tokenized and parsed as if it were an Erlang form - one of the valid Erlang expressions in an Erlang source file - until a final '.' is reached. It returns:

{ok, AbsForm, EndLine} The parsing was successful.

{eof, EndLine} End of file was encountered.

{error, ErrorInfo, EndLine} An error occurred.

Standard Input/Output

All Erlang processes have a default standard IO device. This device is used when no IoDevice argument is specified in the above function calls. However, it is sometimes desirable to use an explicit IoDevice argument which refers to the default IO device. This is the case with functions that can access either a file or the default IO device. The atom standard_io has this special meaning. The following example illustrates this:

```
27> io:read('enter>').
enter>foo.
{ok,foo}
28> io:read(standard_io, 'enter>').
enter>bar.
{ok,bar}
```

There is always a process registered under the name of user. This can be used for sending output to the user.

Error Information

The ErrorInfo mentioned above is the standard ErrorInfo structure which is returned from all IO modules. It has the format:

```
{ErrorLine, Module, ErrorDescriptor}
```

A string which describes the error is obtained with the following call:

```
apply(Module, format_error, ErrorDescriptor)
```

io_lib

Erlang Module

This module contains functions for converting to and from strings (lists of characters). They are used for implementing the functions in the `io` module. There is no guarantee that the character lists returned from some of the functions are flat, they can be deep lists. `lists:flatten/1` can be used for flattening deep lists.

DATA TYPES

```
chars() = [char() | chars()]
```

Exports

`nl() -> chars()`

Returns a character list which represents a new line character.

`write(Term) ->`
`write(Term, Depth) -> chars()`

Types:

- Term = term()
- Depth = int()

Returns a character list which represents `Term`. The `Depth` (-1) argument controls the depth of the structures written. When the specified depth is reached, everything below this level is replaced by "...". For example:

```
1> lists:flatten(io_lib:write({1,[2],[3],[4,5],6,7,8,9})).
"{1,[2],[3],[4,5],6,7,8,9}"
2> lists:flatten(io_lib:write({1,[2],[3],[4,5],6,7,8,9}, 5)).
"{1,[2],[3],[...],...}"
```

`print(Term) ->`
`print(Term, Column, LineLength, Depth) -> chars()`

Types:

- Term = term()
- Column = LineLenght = Depth = int()

Also returns a list of characters which represents Term, but breaks representations which are longer than one line into many lines and indents each line sensibly. It also tries to detect and output lists of printable characters as strings. Column is the starting column (1), LineLength the maximum line length (80), and Depth (-1) the maximum print depth.

```
fwrite(Format, Data) ->
format(Format, Data) -> chars()
```

Types:

- Format = atom() | string() | binary()
- Data = [term()]

Returns a character list which represents Data formatted in accordance with Format. See io:fwrite/1,2,3 [page 221] for a detailed description of the available formatting options. A fault is generated if there is an error in the format string or argument list.

```
fread(Format, String) -> Result
```

Types:

- Format = String = string()
- Result = {ok, InputList, LeftOverChars} | {more, RestFormat, Nchars, InputStack} | {error, What}
- InputList = chars()
- LeftOverChars = string()
- RestFormat = string()
- Nchars = int()
- InputStack = chars()
- What = term()

Tries to read String in accordance with the control sequences in Format. See io:fread/3 [page 225] for a detailed description of the available formatting options. It is assumed that String contains whole lines. It returns:

{ok, InputList, LeftOverChars} The string was read. InputList is the list of successfully matched and read items, and LeftOverChars are the input characters not used.

{more, RestFormat, Nchars, InputStack} The string was read, but more input is needed in order to complete the original format string. RestFormat is the remaining format string, NChars the number of characters scanned, and InputStack is the reversed list of inputs matched up to that point.

{error, What} The read operation failed and the parameter What gives a hint about the error.

Example:

```
3> io_lib:fread("~f~f~f", "15.6 17.3e-6 24.5").
{ok,[15.6,1.73e-5,24.5],[]}
```

```
fread(Continuation, String, Format) -> Return
```

Types:

- Continuation = see below

- String = Format = string()
- Return = {done, Result, LeftOverChars} | {more, Continuation}
- Result = {ok, InputList} | eof | {error, What}
- InputList = chars()
- What = term()()
- LeftOverChars = string()

This is the re-entrant formatted reader. The continuation of the first call to the functions must be `[]`. Refer to Armstrong, Virding, Williams, 'Concurrent Programming in Erlang', Chapter 13 for a complete description of how the re-entrant input scheme works.

The function returns:

`{done, Result, LeftOverChars}` The input is complete. The result is one of the following:

`{ok, InputList}` The string was read. `InputList` is the list of successfully matched and read items, and `LeftOverChars` are the remaining characters.

`eof` End of file has been encountered. `LeftOverChars` are the input characters not used.

`{error, What}` An error occurred and the parameter `What` gives a hint about the error.

`{more, Continuation}` More data is required to build a term. `Continuation` must be passed to `fread/3`, when more data becomes available.

`write_atom(Atom) -> chars()`

Types:

- Atom = atom()

Returns the list of characters needed to print the atom `Atom`.

`write_string(String) -> chars()`

Types:

- String = string()

Returns the list of characters needed to print `String` as a string.

`write_char(Integer) -> chars()`

Types:

- Integer = int()

Returns the list of characters needed to print a character constant.

`indentation(String, StartIndent) -> int()`

Types:

- String = string()
- StartIndent = int()

Returns the indentation if `String` has been printed, starting at `StartIndent`.

`char_list(Term) -> bool()`

Types:

- Term = term()

Returns `true` if `Term` is a flat list of characters, otherwise it returns `false`.

`deep_char_list(Term) -> bool()`

Types:

- Term = term()

Returns `true` if `Term` is a, possibly deep, list of characters, otherwise it returns `false`.

`printable_list(Term) -> bool()`

Types:

- Term = term()

Returns `true` if `Term` is a flat list of printable characters, otherwise it returns `false`.

lib

Erlang Module

> **Warning:**
> This module is retained for compatibility. It may disappear without warning in a future release.

Exports

`flush_receive() -> void()`

Flushes the message buffer of the current process.

`error_message(Format, Args) -> ok`

Types:

- Format = atom() | string() | binary()
- Args = [term()]

Prints error message `Args` in accordance with `Format`. Similar to `io:format/2`, see io(3) [page 221].

`progname() -> atom()`

Returns the name of the script that started the current Erlang session.

`nonl(String1) -> String2`

Types:

- String1 = String2 = string()

Removes the last newline character, if any, in `String1`.

`send(To, Msg)`

Types:

- To = pid() | Name | {Name,Node}
- Name = Node = atom()
- Msg = term()

This function to makes it possible to send a message using the `apply/3` BIF.

`sendw(To, Msg)`

Types:

- To = pid() | Name | {Name,Node}
- Name = Node = atom()
- Msg = term()

As `send/2`, but waits for an answer. It is implemented as follows:

```
sendw(To, Msg) ->
    To ! {self(),Msg},
    receive
        Reply -> Reply
    end.
```

The message returned is not necessarily a reply to the message sent.

lists

Erlang Module

This module contains functions for list processing. The functions are organized in two groups: those in the first group perform a particular operation on one or more lists, whereas those in the second group are higher-order functions, using a fun as argument to perform an operation on one list.

Unless otherwise stated, all functions assume that position numbering starts at 1. That is, the first element of a list is at position 1.

Exports

`all(Pred, List) -> bool()`

Types:

- Pred = fun(Elem) -> bool()
- Elem = term()
- List = [term()]

Returns `true` if `Pred(Elem)` returns `true` for all elements `Elem` in `List`, otherwise `false`.

`any(Pred, List) -> bool()`

Types:

- Pred = fun(Elem) -> bool()
- Elem = term()
- List = [term()]

Returns `true` if `Pred(Elem)` returns `true` for at least one element `Elem` in `List`.

`append(ListOfLists) -> List1`

Types:

- ListOfLists = [List]
- List = List1 = [term()]

Returns a list in which all the sub-lists of `ListOfLists` have been appended. For example:

```
> lists:append([[1, 2, 3], [a, b], [4, 5, 6]]).
[1,2,3,a,b,4,5,6]
```

`append(List1, List2) -> List3`

Types:

- List1 = List2 = List3 = [term()]

Returns a new list `List3` which is made from the elements of `List1` followed by the elements of `List2`. For example:

```
> lists:append("abc", "def").
"abcdef"
```

`lists:append(A, B)` is equivalent to `A ++ B`.

`concat(Things) -> string()`

Types:

- Things = [Thing]
- Thing = atom() | integer() | float() | string()

Concatenates the text representation of the elements of `Things`. The elements of `Things` can be atoms, integers, floats or strings.

```
> lists:concat([doc, '/', file, '.', 3]).
"doc/file.3"
```

`delete(Elem, List1) -> List2`

Types:

- Elem = term()
- List1 = List2 = [term()]

Returns a copy of `List1` where the first element matching `Elem` is deleted, if there is such an element.

`dropwhile(Pred, List1) -> List2`

Types:

- Pred = fun(Elem) -> bool()
- Elem = term()
- List1 = List2 = [term()]

Drops elements `Elem` from `List1` while `Pred(Elem)` returns `true` and returns the remaining list.

`duplicate(N, Elem) -> List`

Types:

- N = int()
- Elem = term()
- List = [term()]

Returns a list which contains N copies of the term `Elem`. For example:

```
> lists:duplicate(5, xx).
[xx,xx,xx,xx,xx]
```

`filter(Pred, List1) -> List2`

Types:

- Pred = fun(Elem) -> bool()
- Elem = term()
- List1 = List2 = [term()]

`List2` is a list of all elements `Elem` in `List1` for which `Pred(Elem)` returns `true`.

`flatlength(DeepList) -> int()`

Types:

- DeepList = [term() | DeepList]

Equivalent to `length(flatten(DeepList))`, but more efficient.

`flatmap(Fun, List1) -> List2`

Types:

- Fun = fun(A) -> [B]
- List1 = [A]
- List2 = [B]
- A = B = term()

Takes a function from `A`s to lists of `B`s, and a list of `A`s (`List1`) and produces a list of `B`s by applying the function to every element in `List1` and appending the resulting lists.

That is, `flatmap` behaves as if it had been defined as follows:

```
flatmap(Fun, List1) ->
    append(map(Fun, List1))
```

Example:

```
> lists:flatmap(fun(X)->[X,X] end, [a,b,c]).
[a,a,b,b,c,c]
```

`flatten(DeepList) -> List`

Types:

- DeepList = [term() | DeepList]
- List = [term()]

Returns a flattened version of `DeepList`.

`flatten(DeepList, Tail) -> List`

Types:

- DeepList = [term() | DeepList]
- Tail = List = [term()]

Returns a flattened version of `DeepList` with the tail `Tail` appended.

`foldl(Fun, Acc0, List) -> Acc1`

Types:

- Fun = fun(Elem, AccIn) -> AccOut
- Elem = term()
- Acc0 = Acc1 = AccIn = AccOut = term()

- List = [term()]

Calls `Fun(Elem, AccIn)` on successive elements `A` of `List`, starting with `AccIn == Acc0`. `Fun/2` must return a new accumulator which is passed to the next call. The function returns the final value of the accumulator. `Acc0` is returned if the list is empty. For example:

```
> lists:foldl(fun(X, Sum) -> X + Sum end, 0, [1,2,3,4,5]).
15
> lists:foldl(fun(X, Prod) -> X * Prod end, 1, [1,2,3,4,5]).
120
```

`foldr(Fun, Acc0, List) -> Acc1`

Types:

- Fun = fun(Elem, AccIn) -> AccOut
- Elem = term()
- Acc0 = Acc1 = AccIn = AccOut = term()
- List = [term()]

Like `foldl/3`, but the list is traversed from right to left. For example:

```
> P = fun(A, AccIn) -> io:format("~p ", [A]), AccIn end.
#Fun<erl_eval.12.2225172>
> lists:foldl(P, void, [1,2,3]).
1 2 3 void
> lists:foldr(P, void, [1,2,3]).
3 2 1 void
```

`foldl/3` is tail recursive and would usually be preferred to `foldr/3`.

`foreach(Fun, List) -> void()`

Types:

- Fun = fun(Elem) -> void()
- Elem = term()
- List = [term()]

Calls `Fun(Elem)` for each element `Elem` in `List`. This function is used for its side effects and the evaluation order is defined to be the same as the order of the elements in the list.

`keydelete(Key, N, TupleList1) -> TupleList2`

Types:

- Key = term()
- N = 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = [Tuple]
- Tuple = tuple()

Returns a copy of `TupleList1` where the first occurrence of a tuple whose `Nth` element compares equal to `Key` is deleted, if there is such a tuple.

`keymap(Fun, N, TupleList1) -> TupleList2`

Types:

- Fun = fun(Term1) -> Term2
- Term1 = Term2 = term()
- N = 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = [tuple()]

Returns a list of tuples where, for each tuple in TupleList1, the Nth element Term1 of the tuple has been replaced with the result of calling Fun(Term1).

Examples:

```
> Fun = fun(Atom) -> atom_to_list(Atom) end.
#Fun<erl_eval.6.10732646>
2> lists:keymap(Fun, 2, [{name,jane,22},{name,lizzie,20},{name,lydia,15}]).
[{name,"jane",22},{name,"lizzie",20},{name,"lydia",15}]
```

`keymember(Key, N, TupleList) -> bool()`

Types:

- Key = term()
- N = 1..tuple_size(Tuple)
- TupleList = [Tuple]
- Tuple = tuple()

Returns true if there is a tuple in TupleList whose Nth element compares equal to Key, otherwise false.

`keymerge(N, TupleList1, TupleList2) -> TupleList3`

Types:

- N = 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = TupleList3 = [Tuple]
- Tuple = tuple()

Returns the sorted list formed by merging TupleList1 and TupleList2. The merge is performed on the Nth element of each tuple. Both TupleList1 and TupleList2 must be key-sorted prior to evaluating this function. When two tuples compare equal, the tuple from TupleList1 is picked before the tuple from TupleList2.

`keyreplace(Key, N, TupleList1, NewTuple) -> TupleList2`

Types:

- Key = term()
- N = 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = [Tuple]
- NewTuple = Tuple = tuple()

Returns a copy of TupleList1 where the first occurrence of a T tuple whose Nth element compares equal to Key is replaced with NewTuple, if there is such a tuple T.

`keysearch(Key, N, TupleList) -> {value, Tuple} | false`

Types:

- Key = term()
- N = 1..tuple_size(Tuple)

- TupleList = [Tuple]
- Tuple = tuple()

Searches the list of tuples `TupleList` for a tuple whose `N`th element compares equal to `Key`. Returns `{value, Tuple}` if such a tuple is found, otherwise `false`.

`keysort(N, TupleList1) -> TupleList2`

Types:

- N = 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = [Tuple]
- Tuple = tuple()

Returns a list containing the sorted elements of the list `TupleList1`. Sorting is performed on the `N`th element of the tuples.

`keystore(Key, N, TupleList1, NewTuple) -> TupleList2`

Types:

- Key = term()
- N = 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = [Tuple]
- NewTuple = Tuple = tuple()

Returns a copy of `TupleList1` where the first occurrence of a tuple `T` whose `N`th element compares equal to `Key` is replaced with `NewTuple`, if there is such a tuple `T`. If there is no such tuple `T` a copy of `TupleList1` where `[NewTuple]` has been appended to the end is returned.

`keytake(Key, N, TupleList1) -> {value, Tuple, TupleList2} | false`

Types:

- Key = term()
- N = 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = [Tuple]
- Tuple = tuple()

Searches the list of tuples `TupleList1` for a tuple whose `N`th element compares equal to `Key`. Returns `{value, Tuple, TupleList2}` if such a tuple is found, otherwise `false`. `TupleList2` is a copy of `TupleList1` where the first occurrence of `Tuple` has been removed.

`last(List) -> Last`

Types:

- List = [term()], length(List) > 0
- Last = term()

Returns the last element in `List`.

`map(Fun, List1) -> List2`

Types:

- Fun = fun(A) -> B

- List1 = [A]
- List2 = [B]
- A = B = term()

Takes a function from As to Bs, and a list of As and produces a list of Bs by applying the function to every element in the list. This function is used to obtain the return values. The evaluation order is implementation dependent.

`mapfoldl(Fun, Acc0, List1) -> {List2, Acc1}`

Types:

- Fun = fun(A, AccIn) -> {B, AccOut}
- Acc0 = Acc1 = AccIn = AccOut = term()
- List1 = [A]
- List2 = [B]
- A = B = term()

`mapfold` combines the operations of `map/2` and `foldl/3` into one pass. An example, summing the elements in a list and double them at the same time:

```
> lists:mapfoldl(fun(X, Sum) -> {2*X, X+Sum} end,
0, [1,2,3,4,5]).
{[2,4,6,8,10],15}
```

`mapfoldr(Fun, Acc0, List1) -> {List2, Acc1}`

Types:

- Fun = fun(A, AccIn) -> {B, AccOut}
- Acc0 = Acc1 = AccIn = AccOut = term()
- List1 = [A]
- List2 = [B]
- A = B = term()

`mapfold` combines the operations of `map/2` and `foldr/3` into one pass.

`max(List) -> Max`

Types:

- List = [term()], length(List) > 0
- Max = term()

Returns the first element of `List` that compares greater than or equal to all other elements of `List`.

`member(Elem, List) -> bool()`

Types:

- Elem = term()
- List = [term()]

Returns `true` if `Elem` matches some element of `List`, otherwise `false`.

`merge(ListOfLists) -> List1`

Types:

- ListOfLists = [List]
- List = List1 = [term()]

Returns the sorted list formed by merging all the sub-lists of `ListOfLists`. All sub-lists must be sorted prior to evaluating this function. When two elements compare equal, the element from the sub-list with the lowest position in `ListOfLists` is picked before the other element.

`merge(List1, List2) -> List3`

Types:

- List1 = List2 = List3 = [term()]

Returns the sorted list formed by merging `List1` and `List2`. Both `List1` and `List2` must be sorted prior to evaluating this function. When two elements compare equal, the element from `List1` is picked before the element from `List2`.

`merge(Fun, List1, List2) -> List3`

Types:

- Fun = fun(A, B) -> bool()
- List1 = [A]
- List2 = [B]
- List3 = [A | B]
- A = B = term()

Returns the sorted list formed by merging `List1` and `List2`. Both `List1` and `List2` must be sorted according to the ordering function `Fun` prior to evaluating this function. `Fun(A, B)` should return `true` if `A` comes before `B` in the ordering, `false` otherwise. When two elements compare equal, the element from `List1` is picked before the element from `List2`.

`merge3(List1, List2, List3) -> List4`

Types:

- List1 = List2 = List3 = List4 = [term()]

Returns the sorted list formed by merging `List1`, `List2` and `List3`. All of `List1`, `List2` and `List3` must be sorted prior to evaluating this function. When two elements compare equal, the element from `List1`, if there is such an element, is picked before the other element, otherwise the element from `List2` is picked before the element from `List3`.

`min(List) -> Min`

Types:

- List = [term()], length(List) > 0
- Min = term()

Returns the first element of `List` that compares less than or equal to all other elements of `List`.

`nth(N, List) -> Elem`

Types:

- N = 1..length(List)
- List = [term()]
- Elem = term()

Returns the Nth element of List. For example:

```
> lists:nth(3, [a, b, c, d, e]).
c
```

`nthtail(N, List1) -> Tail`

Types:

- N = 0..length(List1)
- List1 = Tail = [term()]

Returns the Nth tail of List, that is, the sublist of List starting at N+1 and continuing up to the end of the list. For example:

```
> lists:nthtail(3, [a, b, c, d, e]).
[d,e]
> tl(tl(tl([a, b, c, d, e]))).
[d,e]
> lists:nthtail(0, [a, b, c, d, e]).
[a,b,c,d,e]
> lists:nthtail(5, [a, b, c, d, e]).
[]
```

`partition(Pred, List) -> {Satisfying, NonSatisfying}`

Types:

- Pred = fun(Elem) -> bool()
- Elem = term()
- List = Satisfying = NonSatisfying = [term()]

Partitions List into two lists, where the first list contains all elements for which Pred(Elem) returns true, and the second list contains all elements for which Pred(Elem) returns false.

Examples:

```
> lists:partition(fun(A) -> A rem 2 == 1 end, [1,2,3,4,5,6,7]).
{[1,3,5,7],[2,4,6]}
> lists:partition(fun(A) -> is_atom(A) end, [a,b,1,c,d,2,3,4,e]).
{[a,b,c,d,e],[1,2,3,4]}
```

See also splitwith/2 for a different way to partition a list.

`prefix(List1, List2) -> bool()`

Types:

- List1 = List2 = [term()]

Returns true if List1 is a prefix of List2, otherwise false.

`reverse(List1) -> List2`

Types:

- List1 = List2 = [term()]

Returns a list with the top level elements in `List1` in reverse order.

`reverse(List1, Tail) -> List2`

Types:

- List1 = Tail = List2 = [term()]

Returns a list with the top level elements in `List1` in reverse order, with the tail `Tail` appended. For example:

```
> lists:reverse([1, 2, 3, 4], [a, b, c]).
[4,3,2,1,a,b,c]
```

`seq(From, To) -> Seq`
`seq(From, To, Incr) -> Seq`

Types:

- From = To = Incr = int()
- Seq = [int()]

Returns a sequence of integers which starts with `From` and contains the successive results of adding `Incr` to the previous element, until `To` has been reached or passed (in the latter case, `To` is not an element of the sequence). `Incr` defaults to 1.

Failure: If `To<From` and `Incr` is positive, or if `To>From` and `Incr` is negative, or if `Incr==0` and `From/=To`.

Examples:

```
> lists:seq(1, 10).
[1,2,3,4,5,6,7,8,9,10]
> lists:seq(1, 20, 3).
[1,4,7,10,13,16,19]
> lists:seq(1, 1, 0).
[1]
```

`sort(List1) -> List2`

Types:

- List1 = List2 = [term()]

Returns a list containing the sorted elements of `List1`.

`sort(Fun, List1) -> List2`

Types:

- Fun = fun(Elem1, Elem2) -> bool()
- Elem1 = Elem2 = term()
- List1 = List2 = [term()]

Returns a list containing the sorted elements of `List1`, according to the ordering function `Fun`. `Fun(A, B)` should return `true` if `A` comes before `B` in the ordering, `false` otherwise.

`split(N, List1) -> {List2, List3}`

Types:

- N = 0..length(List1)
- List1 = List2 = List3 = [term()]

Splits `List1` into `List2` and `List3`. `List2` contains the first `N` elements and `List3` the rest of the elements (the `N`th tail).

`splitwith(Pred, List) -> {List1, List2}`

Types:

- Pred = fun(Elem) -> bool()
- Elem = term()
- List = List1 = List2 = [term()]

Partitions `List` into two lists according to `Pred`. `splitwith/2` behaves as if it is defined as follows:

```
splitwith(Pred, List) ->
    {takewhile(Pred, List), dropwhile(Pred, List)}.
```

Examples:

```
> lists:splitwith(fun(A) -> A rem 2 == 1 end, [1,2,3,4,5,6,7]).
{[1],[2,3,4,5,6,7]}
> lists:splitwith(fun(A) -> is_atom(A) end, [a,b,1,c,d,2,3,4,e]).
{[a,b],[1,c,d,2,3,4,e]}
```

See also `partition/2` for a different way to partition a list.

`sublist(List1, Len) -> List2`

Types:

- List1 = List2 = [term()]
- Len = int()

Returns the sub-list of `List1` starting at position 1 and with (max) `Len` elements. It is not an error for `Len` to exceed the length of the list – in that case the whole list is returned.

`sublist(List1, Start, Len) -> List2`

Types:

- List1 = List2 = [term()]
- Start = 1..(length(List1)+1)
- Len = int()

Returns the sub-list of `List1` starting at `Start` and with (max) `Len` elements. It is not an error for `Start+Len` to exceed the length of the list.

```
> lists:sublist([1,2,3,4], 2, 2).
[2,3]
> lists:sublist([1,2,3,4], 2, 5).
[2,3,4]
> lists:sublist([1,2,3,4], 5, 2).
[]
```

`subtract(List1, List2) -> List3`

Types:

- List1 = List2 = List3 = [term()]

Returns a new list `List3` which is a copy of `List1`, subjected to the following procedure: for each element in `List2`, its first occurrence in `List1` is deleted. For example:

```
> lists:subtract("123212", "212").
"312".
```

`lists:subtract(A,B)` is equivalent to `A -- B`.

`suffix(List1, List2) -> bool()`

Returns `true` if `List1` is a suffix of `List2`, otherwise `false`.

`sum(List) -> number()`

Types:

- List = [number()]

Returns the sum of the elements in `List`.

`takewhile(Pred, List1) -> List2`

Types:

- Pred = fun(Elem) -> bool()
- Elem = term()
- List1 = List2 = [term()]

Takes elements `Elem` from `List1` while `Pred(Elem)` returns `true`, that is, the function returns the longest prefix of the list for which all elements satisfy the predicate.

`ukeymerge(N, TupleList1, TupleList2) -> TupleList3`

Types:

- N = 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = TupleList3 = [Tuple]
- Tuple = tuple()

Returns the sorted list formed by merging `TupleList1` and `TupleList2`. The merge is performed on the `N`th element of each tuple. Both `TupleList1` and `TupleList2` must be key-sorted without duplicates prior to evaluating this function. When two tuples compare equal, the tuple from `TupleList1` is picked and the one from `TupleList2` deleted.

`ukeysort(N, TupleList1) -> TupleList2`

Types:

- N = 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = [Tuple]
- Tuple = tuple()

Returns a list containing the sorted elements of the list `TupleList1` where all but the first tuple of the tuples comparing equal have been deleted. Sorting is performed on the `N`th element of the tuples.

`umerge(ListOfLists) -> List1`

Types:

- ListOfLists = [List]
- List = List1 = [term()]

Returns the sorted list formed by merging all the sub-lists of `ListOfLists`. All sub-lists must be sorted and contain no duplicates prior to evaluating this function. When two elements compare equal, the element from the sub-list with the lowest position in `ListOfLists` is picked and the other one deleted.

`umerge(List1, List2) -> List3`

Types:

- List1 = List2 = List3 = [term()]

Returns the sorted list formed by merging `List1` and `List2`. Both `List1` and `List2` must be sorted and contain no duplicates prior to evaluating this function. When two elements compare equal, the element from `List1` is picked and the one from `List2` deleted.

`umerge(Fun, List1, List2) -> List3`

Types:

- Fun = fun(A, B) -> bool()
- List1 = [A]
- List2 = [B]
- List3 = [A | B]
- A = B = term()

Returns the sorted list formed by merging `List1` and `List2`. Both `List1` and `List2` must be sorted according to the ordering function `Fun` and contain no duplicates prior to evaluating this function. `Fun(A, B)` should return `true` if `A` comes before `B` in the ordering, `false` otherwise. When two elements compare equal, the element from `List1` is picked and the one from `List2` deleted.

`umerge3(List1, List2, List3) -> List4`

Types:

- List1 = List2 = List3 = List4 = [term()]

Returns the sorted list formed by merging `List1`, `List2` and `List3`. All of `List1`, `List2` and `List3` must be sorted and contain no duplicates prior to evaluating this function. When two elements compare equal, the element from `List1` is picked if there is such an element, otherwise the element from `List2` is picked, and the other one deleted.

`unzip(List1) -> {List2, List3}`

Types:

- List1 = [{X, Y}]

- List2 = [X]
- List3 = [Y]
- X = Y = term()

"Unzips" a list of two-tuples into two lists, where the first list contains the first element of each tuple, and the second list contains the second element of each tuple.

`unzip3(List1) -> {List2, List3, List4}`

Types:

- List1 = [{X, Y, Z}]
- List2 = [X]
- List3 = [Y]
- List4 = [Z]
- X = Y = Z = term()

"Unzips" a list of three-tuples into three lists, where the first list contains the first element of each tuple, the second list contains the second element of each tuple, and the third list contains the third element of each tuple.

`usort(List1) -> List2`

Types:

- List1 = List2 = [term()]

Returns a list containing the sorted elements of `List1` where all but the first element of the elements comparing equal have been deleted.

`usort(Fun, List1) -> List2`

Types:

- Fun = fun(Elem1, Elem2) -> bool()
- Elem1 = Elem2 = term()
- List1 = List2 = [term()]

Returns a list which contains the sorted elements of `List1` where all but the first element of the elements comparing equal according to the ordering function `Fun` have been deleted. `Fun(A, B)` should return `true` if `A` comes before `B` in the ordering, `false` otherwise.

`zip(List1, List2) -> List3`

Types:

- List1 = [X]
- List2 = [Y]
- List3 = [{X, Y}]
- X = Y = term()

"Zips" two lists of equal length into one list of two-tuples, where the first element of each tuple is taken from the first list and the second element is taken from corresponding element in the second list.

`zip3(List1, List2, List3) -> List4`

Types:

- List1 = [X]
- List2 = [Y]
- List3 = [Z]
- List3 = [{X, Y, Z}]
- X = Y = Z = term()

"Zips" three lists of equal length into one list of three-tuples, where the first element of each tuple is taken from the first list, the second element is taken from corresponding element in the second list, and the third element is taken from the corresponding element in the third list.

`zipwith(Combine, List1, List2) -> List3`

Types:

- Combine = fun(X, Y) -> T
- List1 = [X]
- List2 = [Y]
- List3 = [T]
- X = Y = T = term()

Combine the elements of two lists of equal length into one list. For each pair `X, Y` of list elements from the two lists, the element in the result list will be `Combine(X, Y)`.

`zipwith(fun(X, Y) -> {X,Y} end, List1, List2)` is equivalent to `zip(List1, List2)`.

Examples:

```
> lists:zipwith(fun(X, Y) -> X+Y end, [1,2,3], [4,5,6]).
[5,7,9]
```

`zipwith3(Combine, List1, List2, List3) -> List4`

Types:

- Combine = fun(X, Y, Z) -> T
- List1 = [X]
- List2 = [Y]
- List3 = [Z]
- List4 = [T]
- X = Y = Z = T = term()

Combine the elements of three lists of equal length into one list. For each triple `X, Y, Z` of list elements from the three lists, the element in the result list will be `Combine(X, Y, Z)`.

`zipwith3(fun(X, Y, Z) -> {X,Y,Z} end, List1, List2, List3)` is equivalent to `zip3(List1, List2, List3)`.

Examples:

```
> lists:zipwith3(fun(X, Y, Z) -> X+Y+Z end, [1,2,3], [4,5,6], [7,8,9]).
[12,15,18]
> lists:zipwith3(fun(X, Y, Z) -> [X,Y,Z] end, [a,b,c], [x,y,z], [1,2,3]).
[[a,x,1],[b,y,2],[c,z,3]]
```

log_mf_h

Erlang Module

The log_mf_h is a gen_event handler module which can be installed in any gen_event process. It logs onto disk all events which are sent to an event manager. Each event is written as a binary which makes the logging very fast. However, a tool such as the Report Browser (rb) must be used in order to read the files. The events are written to multiple files. When all files have been used, the first one is re-used and overwritten. The directory location, the number of files, and the size of each file are configurable. The directory will include one file called index, and report files 1, 2, ...

Exports

```
init(Dir, MaxBytes, MaxFiles)
init(Dir, MaxBytes, MaxFiles, Pred) -> Args
```

Types:

- Dir = string()
- MaxBytes = integer()
- MaxFiles = 0 < integer() < 256
- Pred = fun(Event) -> boolean()
- Event = term()
- Args = args()

Initiates the event handler. This function returns Args, which should be used in a call to gen_event:add_handler(EventMgr, log_mf_h, Args).

Dir specifies which directory to use for the log files. MaxBytes specifies the size of each individual file. MaxFiles specifies how many files are used. Pred is a predicate function used to filter the events. If no predicate function is specified, all events are logged.

See Also

gen_event(3) [page 188], rb(3)

math

Erlang Module

This module provides an interface to a number of mathematical functions.

> **Note:**
> Not all functions are implemented on all platforms. In particular, the `erf/1` and `erfc/1` functions are not implemented on Windows.

Exports

`pi() -> float()`

A useful number.

`sin(X)`
`cos(X)`
`tan(X)`
`asin(X)`
`acos(X)`
`atan(X)`
`atan2(Y, X)`
`sinh(X)`
`cosh(X)`
`tanh(X)`
`asinh(X)`
`acosh(X)`
`atanh(X)`
`exp(X)`
`log(X)`
`log10(X)`
`pow(X, Y)`
`sqrt(X)`

Types:

- X = Y = number()

A collection of math functions which return floats. Arguments are numbers.

`erf(X) -> float()`

Types:

- X = number()

Returns the error function of `X`, where

```
erf(X) = 2/sqrt(pi)*integral from 0 to X of exp(-t*t) dt.
```

`erfc(X) -> float()`

Types:

- X = number()

`erfc(X)` returns `1.0 - erf(X)`, computed by methods that avoid cancellation for large `X`.

Bugs

As these are the C library, the bugs are the same.

ms_transform

Erlang Module

This module implements the parse_transform that makes calls to `ets` and `dbg:fun2ms/1` translate into literal match specifications. It also implements the back end for the same functions when called from the Erlang shell.

The translations from fun's to match_specs is accessed through the two "pseudo functions" `ets:fun2ms/1` and `dbg:fun2ms/1`.

Actually this introduction is more or less an introduction to the whole concept of match specifications. Since everyone trying to use `ets:select` or `dbg` seems to end up reading this page, it seems in good place to explain a little more than just what this module does.

There are some caveats one should be aware of, please read through the whole manual page if it's the first time you're using the transformations.

Match specifications are used more or less as filters. They resemble usual Erlang matching in a list comprehension or in a `fun` used in conjunction with `lists:foldl` etc. The syntax of pure match specifications is somewhat awkward though, as they are made up purely by Erlang terms and there is no syntax in the language to make the match specifications more readable.

As the match specifications execution and structure is quite like that of a fun, it would for most programmers be more straight forward to simply write it using the familiar fun syntax and having that translated into a match specification automatically. Of course a real fun is more powerful than the match specifications allow, but bearing the match specifications in mind, and what they can do, it's still more convenient to write it all as a fun. This module contains the code that simply translates the fun syntax into match_spec terms.

Let's start with an ets example. Using `ets:select` and a match specification, one can filter out rows of a table and construct a list of tuples containing relevant parts of the data in these rows. Of course one could use `ets:foldl` instead, but the select call is far more efficient. Without the translation, one has to struggle with writing match specifications terms to accommodate this, or one has to resort to the less powerful `ets:match(_object)` calls, or simply give up and use the more inefficient method of `ets:foldl`. Using the `ets:fun2ms` transformation, a `ets:select` call is at least as easy to write as any of the alternatives.

As an example, consider a simple table of employees:

```
-record(emp, {empno,     %Employee number as a string, the key
              surname,   %Surname of the employee
              givenname, %Given name of employee
              dept,      %Department one of {dev,sales,prod,adm}
              empyear}). %Year the employee was employed
```

We create the table using:

```
ets:new(emp_tab,[{keypos,#emp.empno},named_table,ordered_set]).
```

Let's also fill it with some randomly chosen data for the examples:

```
[{emp,"011103","Black","Alfred",sales,2000},
 {emp,"041231","Doe","John",prod,2001},
 {emp,"052341","Smith","John",dev,1997},
 {emp,"076324","Smith","Ella",sales,1995},
 {emp,"122334","Weston","Anna",prod,2002},
 {emp,"535216","Chalker","Samuel",adm,1998},
 {emp,"789789","Harrysson","Joe",adm,1996},
 {emp,"963721","Scott","Juliana",dev,2003},
 {emp,"989891","Brown","Gabriel",prod,1999}]
```

Now, the amount of data in the table is of course to small to justify complicated ets searches, but on real tables, using `select` to get exactly the data you want will increase efficiency remarkably.

Lets say for example that we'd want the employee numbers of everyone in the sales department. One might use `ets:match` in such a situation:

```
1> ets:match(emp_tab, {'_', '$1', '_', '_', sales, '_'}).
[["011103"],["076324"]]
```

Even though `ets:match` does not require a full match specification, but a simpler type, it's still somewhat unreadable, and one has little control over the returned result, it's always a list of lists. OK, one might use `ets:foldl` or `ets:foldr` instead:

```
ets:foldr(fun(#emp{empno = E, dept = sales},Acc) -> [E | Acc];
              (_,Acc) -> Acc
          end,
          [],
          emp_tab).
```

Running that would result in `["011103","076324"]` , which at least gets rid of the extra lists. The fun is also quite straightforward, so the only problem is that all the data from the table has to be transferred from the table to the calling process for filtering. That's inefficient compared to the `ets:match` call where the filtering can be done "inside" the emulator and only the result is transferred to the process. Remember that ets tables are all about efficiency, if it wasn't for efficiency all of ets could be implemented in Erlang, as a process receiving requests and sending answers back. One uses ets because one wants performance, and therefore one wouldn't want all of the table transferred to the process for filtering. OK, let's look at a pure `ets:select` call that does what the `ets:foldr` does:

```
ets:select(emp_tab,[{#emp{empno = '$1', dept = sales, _='_'},[],['$1']}]).
```

Even though the record syntax is used, it's still somewhat hard to read and even harder to write. The first element of the tuple, `#emp{empno = '$1', dept = sales, _='_'}` tells what to match, elements not matching this will not be returned at all, as in the `ets:match` example. The second element, the empty list is a list of guard expressions, which we need none, and the third element is the list of expressions constructing the return value (in ets this almost always is a list containing one single term). In our case `'$1'` is bound to the employee number in the head (first element of tuple), and hence it is the employee number that is returned. The result is `["011103","076324"]`, just as in the `ets:foldr` example, but the result is retrieved much more efficiently in terms of execution speed and memory consumption.

We have one efficient but hardly readable way of doing it and one inefficient but fairly readable (at least to the skilled Erlang programmer) way of doing it. With the use of

ets:fun2ms, one could have something that is as efficient as possible but still is written as a filter using the fun syntax:

```
-include_lib("stdlib/include/ms_transform.hrl").

% ...

ets:select(emp_tab, ets:fun2ms(
                      fun(#emp{empno = E, dept = sales}) ->
                              E
                      end)).
```

This may not be the shortest of the expressions, but it requires no special knowledge of match specifications to read. The fun's head should simply match what you want to filter out and the body returns what you want returned. As long as the fun can be kept within the limits of the match specifications, there is no need to transfer all data of the table to the process for filtering as in the ets:foldr example. In fact it's even easier to read then the ets:foldr example, as the select call in itself discards anything that doesn't match, while the fun of the foldr call needs to handle both the elements matching and the ones not matching.

It's worth noting in the above ets:fun2ms example that one needs to include ms_transform.hrl in the source code, as this is what triggers the parse transformation of the ets:fun2ms call to a valid match specification. This also implies that the transformation is done at compile time (except when called from the shell of course) and therefore will take no resources at all in runtime. So although you use the more intuitive fun syntax, it gets as efficient in runtime as writing match specifications by hand.

Let's look at some more ets examples. Let's say one wants to get all the employee numbers of any employee hired before the year 2000. Using ets:match isn't an alternative here as relational operators cannot be expressed there. Once again, an ets:foldr could do it (slowly, but correct):

```
ets:foldr(fun(#emp{empno = E, empyear = Y},Acc) when Y < 2000 -> [E | Acc];
                  (_,Acc) -> Acc
          end,
          [],
          emp_tab).
```

The result will be ["052341","076324","535216","789789","989891"], as expected. Now the equivalent expression using a handwritten match specification would look something like this:

```
ets:select(emp_tab,[{#emp{empno = '$1', empyear = '$2', _='_'},
                     [{'<', '$2', 2000}],
                     ['$1']}]).
```

This gives the same result, the [{'<', '$2', 2000}] is in the guard part and therefore discards anything that does not have a empyear (bound to '$2' in the head) less than 2000, just as the guard in the foldl example. Lets jump on to writing it using ets:fun2ms

```
-include_lib("stdlib/include/ms_transform.hrl").

% ...

ets:select(emp_tab, ets:fun2ms(
                      fun(#emp{empno = E, empyear = Y}) when Y < 2000 ->
                              E
                      end)).
```

Obviously readability is gained by using the parse transformation.

I'll show some more examples without the tiresome comparing-to-alternatives stuff. Let's say we'd want the whole object matching instead of only one element. We could of course assign a variable to every part of the record and build it up once again in the body of the `fun`, but it's easier to do like this:

```
ets:select(emp_tab, ets:fun2ms(
                      fun(Obj = #emp{empno = E, empyear = Y})
                         when Y < 2000 ->
                              Obj
                      end)).
```

Just as in ordinary Erlang matching, you can bind a variable to the whole matched object using a "match in then match", i.e. a `=`. Unfortunately this is not general in `fun's` translated to match specifications, only on the "top level", i.e. matching the *whole* object arriving to be matched into a separate variable, is it allowed. For the one's used to writing match specifications by hand, I'll have to mention that the variable A will simply be translated into '$_'. It's not general, but it has very common usage, why it is handled as a special, but useful, case. If this bothers you, the pseudo function `object` also returns the whole matched object, see the part about caveats and limitations below.

Let's do something in the `fun`'s body too: Let's say that someone realizes that there are a few people having an employee number beginning with a zero (`0`), which shouldn't be allowed. All those should have their numbers changed to begin with a one (`1`) instead and one wants the list `[{<Old empno>,<New empno>}]` created:

```
ets:select(emp_tab, ets:fun2ms(
                      fun(#emp{empno = [$0 | Rest] }) ->
                              {[$0|Rest],[$1|Rest]}
                      end)).
```

As a matter of fact, this query hit's the feature of partially bound keys in the table type `ordered_set`, so that not the whole table need be searched, only the part of the table containing keys beginning with `0` is in fact looked into.

The fun of course can have several clauses, so that if one could do the following: For each employee, if he or she is hired prior to 1997, return the tuple `{inventory, <employee number>}`, for each hired 1997 or later, but before 2001, return `{rookie, <employee number>}`, for all others return `{newbie, <employee number>}`. All except for the ones named `Smith` as they would be affronted by anything other than the tag `guru` and that is also what's returned for their numbers; `{guru, <employee number>}`:

```
ets:select(emp_tab, ets:fun2ms(
                      fun(#emp{empno = E, surname = "Smith" }) ->
                              {guru,E};
                         (#emp{empno = E, empyear = Y}) when Y < 1997  ->
                              {inventory, E};
```

```
                    (#emp{empno = E, empyear = Y}) when Y > 2001  ->
                         {newbie, E};
                    (#emp{empno = E, empyear = Y}) -> % 1997 -- 2001
                         {rookie, E}
                end)).
```

The result will be:

```
[{rookie,"011103"},
 {rookie,"041231"},
 {guru,"052341"},
 {guru,"076324"},
 {newbie,"122334"},
 {rookie,"535216"},
 {inventory,"789789"},
 {newbie,"963721"},
 {rookie,"989891"}]
```

and so the Smith's will be happy...

So, what more can you do? Well, the simple answer would be; look in the documentation of match specifications in ERTS users guide. However let's briefly go through the most useful "built in functions" that you can use when the `fun` is to be translated into a match specification by `ets:fun2ms` (it's worth mentioning, although it might be obvious to some, that calling other functions than the one's allowed in match specifications cannot be done. No "usual" Erlang code can be executed by the `fun` being translated by `fun2ms`, the `fun` is after all limited exactly to the power of the match specifications, which is unfortunate, but the price one has to pay for the execution speed of an `ets:select` compared to `ets:foldl/foldr`).

The head of the `fun` is obviously a head matching (or mismatching) *one* parameter, one object of the table we `select` from. The object is always a single variable (can be `_`) or a tuple, as that's what's in `ets, dets` and `mnesia` tables (the match specification returned by `ets:fun2ms` can of course be used with `dets:select` and `mnesia:select` as well as with `ets:select`). The use of `=` in the head is allowed (and encouraged) on the top level.

The guard section can contain any guard expression of Erlang. Even the "old" type test are allowed on the toplevel of the guard (`integer(X)` instead of `is_integer(X)`). As the new type tests (the `is_` tests) are in practice just guard bif's they can also be called from within the body of the fun, but so they can in ordinary Erlang code. Also arithmetics is allowed, as well as ordinary guard bif's. Here's a list of bif's and expressions:

- The type tests: is_atom, is_constant, is_float, is_integer, is_list, is_number, is_pid, is_port, is_reference, is_tuple, is_binary, is_function, is_record
- The boolean operators: not, and, or, andalso, orelse
- The relational operators: >, >=, <, =<, =:=, ==, =/=, /=
- Arithmetics: +, -, *, div, rem
- Bitwise operators: band, bor, bxor, bnot, bsl, bsr
- The guard bif's: abs, element, hd, length, node, round, size, tl, trunc, self
- The obsolete type test (only in guards): atom, constant, float, integer, list, number, pid, port, reference, tuple, binary, function, record

Contrary to the fact with “handwritten” match specifications, the `is_record` guard works as in ordinary Erlang code.

Semicolons (`;`) in guards are allowed, the result will be (as expected) one “match_spec-clause” for each semicolon-separated part of the guard. The semantics beeing identical to the Erlang semantics.

The body of the `fun` is used to construct the resulting value. When selecting from tables one usually just construct a suiting term here, using ordinary Erlang term construction, like tuple parentheses, list brackets and variables matched out in the head, possibly in conjunction with the occasional constant. Whatever expressions are allowed in guards are also allowed here, but there are no special functions except `object` and `bindings` (see further down), which returns the whole matched object and all known variable bindings respectively.

The `dbg` variants of match specifications have an imperative approach to the match specification body, the ets dialect hasn't. The fun body for `ets:fun2ms` returns the result without side effects, and as matching (`=`) in the body of the match specifications is not allowed (for performance reasons) the only thing left, more or less, is term construction...

Let's move on to the `dbg` dialect, the slightly different match specifications translated by `dbg:fun2ms`.

The same reasons for using the parse transformation applies to `dbg`, maybe even more so as filtering using Erlang code is simply not a good idea when tracing (except afterwards, if you trace to file). The concept is similar to that of `ets:fun2ms` except that you usually use it directly from the shell (which can also be done with `ets:fun2ms`).

Let's manufacture a toy module to trace on

```
-module(toy).

-export([start/1, store/2, retrieve/1]).

start(Args) ->
    toy_table = ets:new(toy_table,Args).

store(Key, Value) ->
    ets:insert(toy_table,{Key,Value}).

retrieve(Key) ->
    [{Key, Value}] = ets:lookup(toy_table,Key),
    Value.
```

During model testing, the first test bails out with a `{badmatch,16}` in `{toy,start,1}`, why?

We suspect the ets call, as we match hard on the return value, but want only the particular `new` call with `toy_table` as first parameter. So we start a default tracer on the node:

```
1> dbg:tracer().
{ok,<0.88.0>}
```

And so we turn on call tracing for all processes, we are going to make a pretty restrictive trace pattern, so there's no need to call trace only a few processes (it usually isn't):

```
2> dbg:p(all,call).
{ok,[{matched,nonode@nohost,25}]}
```

It's time to specify the filter. We want to view calls that resemble `ets:new(toy_table,<something>)`:

```
3> dbg:tp(ets,new,dbg:fun2ms(fun([toy_table,_]) -> true end)).
{ok,[{matched,nonode@nohost,1},{saved,1}]}
```

As can be seen, the `fun`'s used with `dbg:fun2ms` takes a single list as parameter instead of a single tuple. The list matches a list of the parameters to the traced function. A single variable may also be used of course. The body of the fun expresses in a more imperative way actions to be taken if the fun head (and the guards) matches. I return `true` here, but it's only because the body of a fun cannot be empty, the return value will be discarded.

When we run the test of our module now, we get the following trace output:

```
(<0.86.0>) call ets:new(toy_table,[ordered_set])
```

Let's play we haven't spotted the problem yet, and want to see what `ets:new` returns. We do a slightly different trace pattern:

```
4> dbg:tp(ets,new,dbg:fun2ms(fun([toy_table,_]) -> return_trace() end)).
```

Resulting in the following trace output when we run the test:

```
(<0.86.0>) call ets:new(toy_table,[ordered_set])
(<0.86.0>) returned from ets:new/2 -> 24
```

The call to `return_trace`, makes a trace message appear when the function returns. It applies only to the specific function call triggering the match specification (and matching the head/guards of the match specification). This is the by far the most common call in the body of a `dbg` match specification.

As the test now fails with `{badmatch,24}`, it's obvious that the badmatch is because the atom `toy_table` does not match the number returned for an unnamed table. So we spotted the problem, the table should be named and the arguments supplied by our test program does not include `named_table`. We rewrite the start function to:

```
start(Args) ->
    toy_table = ets:new(toy_table,[named_table |Args]).
```

And with the same tracing turned on, we get the following trace output:

```
(<0.86.0>) call ets:new(toy_table,[named_table,ordered_set])
(<0.86.0>) returned from ets:new/2 -> toy_table
```

Very well. Let's say the module now passes all testing and goes into the system. After a while someone realizes that the table `toy_table` grows while the system is running and that for some reason there are a lot of elements with atom's as keys. You had expected only integer keys and so does the rest of the system. Well, obviously not all of the system. You turn on call tracing and try to see calls to your module with an atom as the key:

```
1> dbg:tracer().
{ok,<0.88.0>}
2> dbg:p(all,call).
{ok,[{matched,nonode@nohost,25}]}
3> dbg:tpl(toy,store,dbg:fun2ms(fun([A,_]) when is_atom(A) -> true end)).
{ok,[{matched,nonode@nohost,1},{saved,1}]}
```

We use dbg:tpl here to make sure to catch local calls (let's say the module has grown since the smaller version and we're not sure this inserting of atoms is not done locally...). When in doubt always use local call tracing.

Let's say nothing happens when we trace in this way. Our function is never called with these parameters. We make the conclusion that someone else (some other module) is doing it and we realize that we must trace on ets:insert and want to see the calling function. The calling function may be retrieved using the match specification function caller and to get it into the trace message, one has to use the match spec function message. The filter call looks like this (looking for calls to ets:insert):

```
4> dbg:tpl(ets,insert,dbg:fun2ms(fun([toy_table,{A,_}]) when is_atom(A) ->
                                    message(caller())
                                end)).
{ok,[{matched,nonode@nohost,1},{saved,2}]}
```

The caller will now appear in the "additional message" part of the trace output, and so after a while, the following output comes:

```
(<0.86.0>) call ets:insert(toy_table,{garbage,can}) ({evil_mod,evil_fun,2})
```

You have found out that the function evil_fun of the module evil_mod, with arity 2, is the one causing all this trouble.

This was just a toy example, but it illustrated the most used calls in match specifications for dbg The other, more esotherical calls are listed and explained in the *Users guide of the ERTS application*, they really are beyond the scope of this document.

To end this chatty introduction with something more precise, here follows some parts about caveats and restrictions concerning the fun's used in conjunction with ets:fun2ms and dbg:fun2ms:

Warning:
To use the pseudo functions triggering the translation, one *has to* include the header file ms_transform.hrl in the source code. Failure to do so will possibly result in runtime errors rather than compile time, as the expression may be valid as a plain Erlang program without translation.

Warning:
The fun has to be literally constructed inside the parameter list to the pseudo functions. The fun cannot be bound to a variable first and then passed to ets:fun2ms or dbg:fun2ms, i.e this will work: `ets:fun2ms(fun(A) -> A end)` but not this: `F = fun(A) -> A end, ets:fun2ms(F)`. The later will result in a compile time error if the header is included, otherwise a runtime error. Even if the later construction would ever appear to work, it really doesn't, so don't ever use it.

Several restrictions apply to the fun that is being translated into a match_spec. To put it simple you cannot use anything in the fun that you cannot use in a match_spec. This means that, among others, the following restrictions apply to the fun itself:

- Functions written in Erlang cannot be called, neither local functions, global functions or real fun's

- Everything that is written as a function call will be translated into a match_spec call to a builtin function, so that the call `is_list(X)` will be translated to `{'is_list', '$1'}` (`'$1'` is just an example, the numbering may vary). If one tries to call a function that is not a match_spec builtin, it will cause an error.
- Variables occurring in the head of the `fun` will be replaced by match_spec variables in the order of occurrence, so that the fragment `fun({A,B,C})` will be replaced by `{'$1', '$2', '$3'}` etc. Every occurrence of such a variable later in the match_spec will be replaced by a match_spec variable in the same way, so that the fun `fun({A,B}) when is_atom(A) -> B end` will be translated into `[{{'$1','$2'},[{is_atom,'$1'}],['$2']}]`.
- Variables that are not appearing in the head are imported from the environment and made into match_spec `const` expressions. Example from the shell:

```
1> X = 25.
25
2> ets:fun2ms(fun({A,B}) when A > X -> B end).
[{{'$1','$2'},[{'>','$1',{const,25}}],['$2']}]
```

- Matching with = cannot be used in the body. It can only be used on the top level in the head of the fun. Example from the shell again:

```
1> ets:fun2ms(fun({A,[B|C]} = D) when A > B -> D end).
[{{'$1',['$2'|'$3']},[{'>','$1','$2'}],['$_']}]
2> ets:fun2ms(fun({A,[B|C]=D}) when A > B -> D end).
Error: fun with head matching ('=' in head) cannot be translated into
match_spec
{error,transform_error}
3> ets:fun2ms(fun({A,[B|C]}) when A > B -> D = [B|C], D end).
Error: fun with body matching ('=' in body) is illegal as match_spec
{error,transform_error}
```

All variables are bound in the head of a match_spec, so the translator can not allow multiple bindings. The special case when matching is done on the top level makes the variable bind to `'$_'` in the resulting match_spec, it is to allow a more natural access to the whole matched object. The pseudo function `object()` could be used instead, see below. The following expressions are translated equally:

```
ets:fun2ms(fun({a,_} = A) -> A end).
ets:fun2ms(fun({a,_}) -> object() end).
```

- The special match_spec variables `'$_'` and `'$*'` can be accessed through the pseudo functions `object()` (for `'$_'`) and `bindings()` (for `'$*'`). as an example, one could translate the following `ets:match_object/2` call to a `ets:select` call:

```
ets:match_object(Table, {'$1',test,'$2'}).
```

...is the same as...

```
ets:select(Table, ets:fun2ms(fun({A,test,B}) -> object() end)).
```

(This was just an example, in this simple case the former expression is probably preferable in terms of readability). The `ets:select/2` call will conceptually look like this in the resulting code:

```
ets:select(Table, [{{'$1',test,'$2'},[],['$_']}]).
```

Matching on the top level of the fun head might feel like a more natural way to access `'$_'`, see above.

- Term constructions/literals are translated as much as is needed to get them into valid match_specs, so that tuples are made into match_spec tuple constructions (a one element tuple containing the tuple) and constant expressions are used when importing variables from the environment. Records are also translated into plain tuple constructions, calls to element etc. The guard test `is_record/2` is translated into match_spec code using the three parameter version that's built into match_specs, so that `is_record(A,t)` is translated into `{is_record,'$1',t,5}` given that the record size of record type `t` is 5.
- Language constructions like `case`, `if`, `catch` etc that are not present in match_specs are not allowed.
- If the header file `ms_transform.hrl` is not included, the fun won't be translated, which may result in a *runtime error* (depending on if the fun is valid in a pure Erlang context). Be absolutely sure that the header is included when using `ets` and `dbg:fun2ms/1` in compiled code.
- If the pseudo function triggering the translation is `ets:fun2ms/1`, the fun's head must contain a single variable or a single tuple. If the pseudo function is `dbg:fun2ms/1` the fun's head must contain a single variable or a single list.

The translation from fun's to match_specs is done at compile time, so runtime performance is not affected by using these pseudo functions. The compile time might be somewhat longer though.

For more information about match_specs, please read about them in *ERTS users guide*.

Exports

`parse_transform(Forms,_Options) -> Forms`

Types:

- Forms = Erlang abstract code format, see the erl_parse module description
- _Options = Option list, required but not used

Implements the actual transformation at compile time. This function is called by the compiler to do the source code transformation if and when the `ms_transform.hrl` header file is included in your source code. See the `ets` and `dbg:fun2ms/1` function manual pages for documentation on how to use this parse_transform, see the `match_spec` chapter in `ERTS` users guide for a description of match specifications.

`transform_from_shell(Dialect,Clauses,BoundEnvironment) -> term()`

Types:

- Dialect = ets | dbg
- Clauses = Erlang abstract form for a single fun
- BoundEnvironment = [{atom(), term()}, ...], list of variable bindings in the shell environment

Implements the actual transformation when the `fun2ms` functions are called from the shell. In this case the abstract form is for one single fun (parsed by the Erlang shell), and all imported variables should be in the key-value list passed as `BoundEnvironment`. The result is a term, normalized, i.e. not in abstract format.

`format_error(Errcode) -> ErrMessage`

Types:

- Errcode = term()
- ErrMessage = string()

Takes an error code returned by one of the other functions in the module and creates a textual description of the error. Fairly uninteresting function actually.

orddict

Erlang Module

`Orddict` implements a `Key - Value` dictionary. An `orddict` is a representation of a dictionary, where a list of pairs is used to store the keys and values. The list is ordered after the keys.

This module provides exactly the same interface as the module `dict` but with a defined representation. One difference is that while `dict` considers two keys as different if they do not match (`=:=`), this module considers two keys as different if and only if they do not compare equal (`==`).

DATA TYPES

```
ordered_dictionary()
  as returned by new/0
```

Exports

`append(Key, Value, Orddict1) -> Orddict2`

Types:

- Key = Value = term()
- Orddict1 = Orddict2 = ordered_dictionary()

This function appends a new `Value` to the current list of values associated with `Key`. An exception is generated if the initial value associated with `Key` is not a list of values.

`append_list(Key, ValList, Orddict1) -> Orddict2`

Types:

- ValList = [Value]
- Key = Value = term()
- Orddict1 = Orddict2 = ordered_dictionary()

This function appends a list of values `ValList` to the current list of values associated with `Key`. An exception is generated if the initial value associated with `Key` is not a list of values.

`erase(Key, Orddict1) -> Orddict2`

Types:

- Key = term()
- Orddict1 = Orddict2 = ordered_dictionary()

This function erases all items with a given key from a dictionary.

`fetch(Key, Orddict) -> Value`

Types:

- Key = Value = term()
- Orddict = ordered_dictionary()

This function returns the value associated with `Key` in the dictionary `Orddict`. `fetch` assumes that the `Key` is present in the dictionary and an exception is generated if `Key` is not in the dictionary.

`fetch_keys(Orddict) -> Keys`

Types:

- Orddict = ordered_dictionary()
- Keys = [term()]

This function returns a list of all keys in the dictionary.

`filter(Pred, Orddict1) -> Orddict2`

Types:

- Pred = fun(Key, Value) -> bool()
- Key = Value = term()
- Orddict1 = Orddict2 = ordered_dictionary()

`Orddict2` is a dictionary of all keys and values in `Orddict1` for which `Pred(Key, Value)` is `true`.

`find(Key, Orddict) -> {ok, Value} | error`

Types:

- Key = Value = term()
- Orddict = ordered_dictionary()

This function searches for a key in a dictionary. Returns `{ok, Value}` where `Value` is the value associated with `Key`, or `error` if the key is not present in the dictionary.

`fold(Fun, Acc0, Orddict) -> Acc1`

Types:

- Fun = fun(Key, Value, AccIn) -> AccOut
- Key = Value = term()
- Acc0 = Acc1 = AccIn = AccOut = term()
- Orddict = ordered_dictionary()

Calls `Fun` on successive keys and values of `Orddict` together with an extra argument `Acc` (short for accumulator). `Fun` must return a new accumulator which is passed to the next call. `Acc0` is returned if the list is empty. The evaluation order is undefined.

`from_list(List) -> Orddict`

Types:

- List = [{Key, Value}]

- Orddict = ordered_dictionary()

This function converts the key/value list `List` to a dictionary.

`is_key(Key, Orddict) -> bool()`

Types:

- Key = term()
- Orddict = ordered_dictionary()

This function tests if `Key` is contained in the dictionary `Orddict`.

`map(Fun, Orddict1) -> Orddict2`

Types:

- Fun = fun(Key, Value1) -> Value2
- Key = Value1 = Value2 = term()
- Orddict1 = Orddict2 = ordered_dictionary()

`map` calls `Func` on successive keys and values of `Orddict` to return a new value for each key. The evaluation order is undefined.

`merge(Fun, Orddict1, Orddict2) -> Orddict3`

Types:

- Fun = fun(Key, Value1, Value2) -> Value
- Key = Value1 = Value2 = Value3 = term()
- Orddict1 = Orddict2 = Orddict3 = ordered_dictionary()

`merge` merges two dictionaries, `Orddict1` and `Orddict2`, to create a new dictionary. All the `Key` - `Value` pairs from both dictionaries are included in the new dictionary. If a key occurs in both dictionaries then `Fun` is called with the key and both values to return a new value. `merge` could be defined as:

```
merge(Fun, D1, D2) ->
    fold(fun (K, V1, D) ->
                 update(K, fun (V2) -> Fun(K, V1, V2) end, V1, D)
         end, D2, D1).
```

but is faster.

`new() -> ordered_dictionary()`

This function creates a new dictionary.

`size(Orddict) -> int()`

Types:

- Orddict = ordered_dictionary()

Returns the number of elements in an `Orddict`.

`store(Key, Value, Orddict1) -> Orddict2`

Types:

- Key = Value = term()

- Orddict1 = Orddict2 = ordered_dictionary()

This function stores a `Key` - `Value` pair in a dictionary. If the `Key` already exists in `Orddict1`, the associated value is replaced by `Value`.

`to_list(Orddict) -> List`

Types:

- Orddict = ordered_dictionary()
- List = [{Key, Value}]

This function converts the dictionary to a list representation.

`update(Key, Fun, Orddict1) -> Orddict2`

Types:

- Key = term()
- Fun = fun(Value1) -> Value2
- Value1 = Value2 = term()
- Orddict1 = Orddict2 = ordered_dictionary()

Update the a value in a dictionary by calling `Fun` on the value to get a new value. An exception is generated if `Key` is not present in the dictionary.

`update(Key, Fun, Initial, Orddict1) -> Orddict2`

Types:

- Key = Initial = term()
- Fun = fun(Value1) -> Value2
- Value1 = Value2 = term()
- Orddict1 = Orddict2 = ordered_dictionary()

Update the a value in a dictionary by calling `Fun` on the value to get a new value. If `Key` is not present in the dictionary then `Initial` will be stored as the first value. For example `append/3` could be defined as:

```
append(Key, Val, D) ->
    update(Key, fun (Old) -> Old ++ [Val] end, [Val], D).
```

`update_counter(Key, Increment, Orddict1) -> Orddict2`

Types:

- Key = term()
- Increment = number()
- Orddict1 = Orddict2 = ordered_dictionary()

Add `Increment` to the value associated with `Key` and store this value. If `Key` is not present in the dictionary then `Increment` will be stored as the first value.

This could be defined as:

```
update_counter(Key, Incr, D) ->
    update(Key, fun (Old) -> Old + Incr end, Incr, D).
```

but is faster.

Notes

The functions `append` and `append_list` are included so we can store keyed values in a list *accumulator*. For example:

```
> D0 = orddict:new(),
  D1 = orddict:store(files, [], D0),
  D2 = orddict:append(files, f1, D1),
  D3 = orddict:append(files, f2, D2),
  D4 = orddict:append(files, f3, D3),
  orddict:fetch(files, D4).
[f1,f2,f3]
```

This saves the trouble of first fetching a keyed value, appending a new value to the list of stored values, and storing the result.

The function `fetch` should be used if the key is known to be in the dictionary, otherwise `find`.

See Also

dict(3) [page 97], gb_trees(3) [page 183]

ordsets

Erlang Module

Sets are collections of elements with no duplicate elements. An `ordset` is a representation of a set, where an ordered list is used to store the elements of the set. An ordered list is more efficient than an unordered list.

This module provides exactly the same interface as the module `sets` but with a defined representation. One difference is that while `sets` considers two elements as different if they do not match (`=:=`), this module considers two elements as different if and only if they do not compare equal (`==`).

DATA TYPES

```
ordered_set()
  as returned by new/0
```

Exports

`new() -> Ordset`

Types:

- Ordset = ordered_set()

Returns a new empty ordered set.

`is_set(Ordset) -> bool()`

Types:

- Ordset = term()

Returns `true` if `Ordset` is an ordered set of elements, otherwise `false`.

`size(Ordset) -> int()`

Types:

- Ordset = term()

Returns the number of elements in `Ordset`.

`to_list(Ordset) -> List`

Types:

- Ordset = ordered_set()
- List = [term()]

Returns the elements of `Ordset` as a list.

`from_list(List) -> Ordset`

Types:

- List = [term()]
- Ordset = ordered_set()

Returns an ordered set of the elements in `List`.

`is_element(Element, Ordset) -> bool()`

Types:

- Element = term()
- Ordset = ordered_set()

Returns `true` if `Element` is an element of `Ordset`, otherwise `false`.

`add_element(Element, Ordset1) -> Ordset2`

Types:

- Element = term()
- Ordset1 = Ordset2 = ordered_set()

Returns a new ordered set formed from `Ordset1` with `Element` inserted.

`del_element(Element, Ordset1) -> Ordset2`

Types:

- Element = term()
- Ordset1 = Ordset2 = ordered_set()

Returns `Ordset1`, but with `Element` removed.

`union(Ordset1, Ordset2) -> Ordset3`

Types:

- Ordset1 = Ordset2 = Ordset3 = ordered_set()

Returns the merged (union) set of `Ordset1` and `Ordset2`.

`union(OrdsetList) -> Ordset`

Types:

- OrdsetList = [ordered_set()]
- Ordset = ordered_set()

Returns the merged (union) set of the list of sets.

`intersection(Ordset1, Ordset2) -> Ordset3`

Types:

- Ordset1 = Ordset2 = Ordset3 = ordered_set()

Returns the intersection of `Ordset1` and `Ordset2`.

`intersection(OrdsetList) -> Ordset`

Types:

- OrdsetList = [ordered_set()]
- Ordset = ordered_set()

Returns the intersection of the non-empty list of sets.

`subtract(Ordset1, Ordset2) -> Ordset3`

Types:

- Ordset1 = Ordset2 = Ordset3 = ordered_set()

Returns only the elements of `Ordset1` which are not also elements of `Ordset2`.

`is_subset(Ordset1, Ordset2) -> bool()`

Types:

- Ordset1 = Ordset2 = ordered_set()

Returns `true` when every element of `Ordset1` is also a member of `Ordset2`, otherwise `false`.

`fold(Function, Acc0, Ordset) -> Acc1`

Types:

- Function = fun (E, AccIn) -> AccOut
- Acc0 = Acc1 = AccIn = AccOut = term()
- Ordset = ordered_set()

Fold `Function` over every element in `Ordset` returning the final value of the accumulator.

`filter(Pred, Ordset1) -> Set2`

Types:

- Pred = fun (E) -> bool()
- Set1 = Set2 = ordered_set()

Filter elements in `Set1` with boolean function `Fun`.

See Also

gb_sets(3) [page 177], sets(3) [page 354]

pg

Erlang Module

This (experimental) module implements process groups. A process group is a group of processes that can be accessed by a common name. For example, a group named `foobar` can include a set of processes as members of this group and they can be located on different nodes.

When messages are sent to the named group, all members of the group receive the message. The messages are serialized. If the process `P1` sends the message `M1` to the group, and process `P2` simultaneously sends message `M2`, then all members of the group receive the two messages in the same order. If members of a group terminate, they are automatically removed from the group.

This module is not complete. The module is inspired by the ISIS system and the causal order protocol of the ISIS system should also be implemented. At the moment, all messages are serialized by sending them through a group master process.

Exports

`create(PgName) -> ok | {error, Reason}`

Types:

- PgName = term()
- Reason = already_created | term()

Creates an empty group named `PgName` on the current node.

`create(PgName, Node) -> ok | {error, Reason}`

Types:

- PgName = term()
- Node = node()
- Reason = already_created | term()

Creates an empty group named `PgName` on the node `Node`.

`join(PgName, Pid) -> Members`

Types:

- PgName = term()
- Pid = pid()
- Members = [pid()]

Joins the pid `Pid` to the process group `PgName`. Returns a list of all old members of the group.

`send(PgName, Msg) -> void()`

Types:

- PgName = Msg = term()

Sends the tuple {`pg_message, From, PgName, Msg`} to all members of the process group `PgName`.

Failure: {`badarg, {PgName, Msg}`} if `PgName` is not a process group (a globally registered name).

`esend(PgName, Msg) -> void()`

Types:

- PgName = Msg = term()

Sends the tuple {`pg_message, From, PgName, Msg`} to all members of the process group `PgName`, except ourselves.

Failure: {`badarg, {PgName, Msg}`} if `PgName` is not a process group (a globally registered name).

`members(PgName) -> Members`

Types:

- PgName = term()
- Members = [pid()]

Returns a list of all members of the process group `PgName`.

pool

Erlang Module

pool can be used to run a set of Erlang nodes as a pool of computational processors. It is organized as a master and a set of slave nodes and includes the following features:

- The slave nodes send regular reports to the master about their current load.
- Queries can be sent to the master to determine which node will have the least load.

The BIF statistics(run_queue) is used for estimating future loads. It returns the length of the queue of ready to run processes in the Erlang runtime system.

The slave nodes are started with the slave module. This effects, tty IO, file IO, and code loading.

If the master node fails, the entire pool will exit.

Exports

```
start(Name) ->
start(Name, Args) -> Nodes
```

Types:

- Name = atom()
- Args = string()
- Nodes = [node()]

Starts a new pool. The file .hosts.erlang is read to find host names where the pool nodes can be started. See section Files [page 276] below. The start-up procedure fails if the file is not found.

The slave nodes are started with slave:start/2,3, passing along Name and, if provided, Args. Name is used as the first part of the node names, Args is used to specify command line arguments. See slave(3) [page 369].

Access rights must be set so that all nodes in the pool have the authority to access each other.

The function is synchronous and all the nodes, as well as all the system servers, are running when it returns a value.

```
attach(Node) -> allready_attached | attached
```

Types:

- Node = node()

This function ensures that a pool master is running and includes Node in the pool master's pool of nodes.

`stop() -> stopped`

Stops the pool and kills all the slave nodes.

`get_nodes() -> Nodes`

Types:

- Nodes = [node()]

Returns a list of the current member nodes of the pool.

`pspawn(Mod, Fun, Args) -> pid()`

Types:

- Mod = Fun = atom()
- Args = [term()]

Spawns a process on the pool node which is expected to have the lowest future load.

`pspawn_link(Mod, Fun, Args) -> pid()`

Types:

- Mod = Fun = atom()
- Args = [term()]

Spawn links a process on the pool node which is expected to have the lowest future load.

`get_node() -> node()`

Returns the node with the expected lowest future load.

Files

`.hosts.erlang` is used to pick hosts where nodes can be started. See [net_adm(3)] for information about format and location of this file.

`$HOME/.erlang.slave.out.HOST` is used for all additional IO that may come from the slave nodes on standard IO. If the start-up procedure does not work, this file may indicate the reason.

proc_lib

Erlang Module

This module is used to start processes adhering to the [OTP Design Principles]. Specifically, the functions in this module are used by the OTP standard behaviors (`gen_server`, `gen_fsm`, ...) when starting new processes. The functions can also be used to start *special processes*, user defined processes which comply to the OTP design principles. See [Sys and Proc_Lib] in OTP Design Principles for an example.

Some useful information is initialized when a process starts. The registered names, or the process identifiers, of the parent process, and the parent ancestors, are stored together with information about the function initially called in the process.

While in "plain Erlang" a process is said to terminate normally only for the exit reason `normal`, a process started using `proc_lib` is also said to terminate normally if it exits with reason `shutdown`. This is the reason used when an application (supervision tree) is stopped.

When a process started using `proc_lib` terminates abnormally – that is, with another exit reason than `normal` or `shutdown` – a *crash report* is generated, which is written to terminal by the default SASL event handler. That is, the crash report is normally only visible if the SASL application is started. See [sasl(6)] and [SASL User's Guide].

The crash report contains the previously stored information such as ancestors and initial function, the termination reason, and information regarding other processes which terminate as a result of this process terminating.

Exports

```
spawn(Fun) -> pid()
spawn(Node, Fun) -> pid()
spawn(Module, Function, Args) -> pid()
spawn(Node, Module, Function, Args) -> pid()
```

Types:

- Node = node()
- Fun = fun() -> void()
- Module = Function = atom()
- Args = [term()]

Spawns a new process and initializes it as described above. The process is spawned using the [spawn] BIFs.

```
spawn_link(Fun) -> pid()
spawn_link(Node, Fun) -> pid()
spawn_link(Module, Function, Args) -> pid()
```

```
spawn_link(Node, Module, Function, Args) -> pid()
```

Types:

- Node = node()
- Fun = fun() -> void()
- Module = Function = atom()
- Args = [term()]

Spawns a new process and initializes it as described above. The process is spawned using the [spawn_link] BIFs.

```
spawn_opt(Fun, SpawnOpts) -> pid()
spawn_opt(Node, Fun, SpawnOpts) -> pid()
spawn_opt(Module, Function, Args, SpawnOpts) -> pid()
spawn_opt(Node, Module, Func, Args, SpawnOpts) -> pid()
```

Types:

- Node = node()
- Fun = fun() -> void()
- Module = Function = atom()
- Args = [term()]
- SpawnOpts – see erlang:spawn_opt/2,3,4,5

Spawns a new process and initializes it as described above. The process is spawned using the [spawn_opt] BIFs.

> **Note:**
> Using the spawn option `monitor` is currently not allowed, but will cause the function to fail with reason `badarg`.

```
start(Module, Function, Args) -> Ret
start(Module, Function, Args, Time) -> Ret
start(Module, Function, Args, Time, SpawnOpts) -> Ret
start_link(Module, Function, Args) -> Ret
start_link(Module, Function, Args, Time) -> Ret
start_link(Module, Function, Args, Time, SpawnOpts) -> Ret
```

Types:

- Module = Function = atom()
- Args = [term()]
- Time = int() >= 0 | infinity
- SpawnOpts – see erlang:spawn_opt/2,3,4,5
- Ret = term() | {error, Reason}

Starts a new process synchronously. Spawns the process and waits for it to start. When the process has started, it *must* call init_ack(Parent,Ret) [page 279] or init_ack(Ret) [page 279], where `Parent` is the process that evaluates this function. At this time, `Ret` is returned.

If the `start_link/3,4,5` function is used and the process crashes before it has called `init_ack/1,2`, {`error, Reason`} is returned if the calling process traps exits.

If `Time` is specified as an integer, this function waits for `Time` milliseconds for the new process to call `init_ack`, or {`error, timeout`} is returned, and the process is killed.

The `SpawnOpts` argument, if given, will be passed as the last argument to the `spawn_opt/2,3,4,5` BIF.

> **Note:**
> Using the spawn option `monitor` is currently not allowed, but will cause the function to fail with reason `badarg`.

```
init_ack(Parent, Ret) -> void()
init_ack(Ret) -> void()
```

Types:

- Parent = pid()
- Ret = term()

This function must used by a process that has been started by a start[_link]/3,4,5 [page 278] function. It tells `Parent` that the process has initialized itself, has started, or has failed to initialize itself.

The `init_ack/1` function uses the parent value previously stored by the start function used.

If this function is not called, the start function will return an error tuple (if a link and/or a timeout is used) or hang otherwise.

The following example illustrates how this function and `proc_lib:start_link/3` are used.

```
-module(my_proc).
-export([start_link/0]).
-export([init/1]).

start_link() ->
    proc_lib:start_link(my_proc, init, [self()]).

init(Parent) ->
    case do_initialization() of
        ok ->
            proc_lib:init_ack(Parent, {ok, self()});
        {error, Reason} ->
            exit(Reason)
    end,
    loop().

...
```

`format(CrashReport) -> string()`

Types:

- CrashReport = term()

This function can be used by a user defined event handler to format a crash report. The crash report is sent using `error_logger:error_report(crash_report, CrashReport)`. That is, the event to be handled is of the format `{error_report, GL, {Pid, crash_report, CrashReport}}` where `GL` is the group leader pid of the process `Pid` which sent the crash report.

`initial_call(Process) -> {Module,Function,Args} | Fun | false`

Types:

- Process = pid() | {X,Y,Z} | ProcInfo
- X = Y = Z = int()
- ProcInfo = term()
- Module = Function = atom()
- Args = [term()]
- Fun = fun() -> void()

Extracts the initial call of a process that was started using one of the spawn or start functions described above. `Process` can either be a pid, an integer tuple (from which a pid can be created), or the process information of a process `Pid` fetched through an `erlang:process_info(Pid)` function call.

`translate_initial_call(Process) -> {Module,Function,Arity} | Fun`

Types:

- Process = pid() | {X,Y,Z} | ProcInfo
- X = Y = Z = int()
- ProcInfo = term()
- Module = Function = atom()
- Arity = int()
- Fun = fun() -> void()

This function is used by the `c:i/0` and `c:regs/0` functions in order to present process information.

Extracts the initial call of a process that was started using one of the spawn or start functions described above, and translates it to more useful information. `Process` can either be a pid, an integer tuple (from which a pid can be created), or the process information of a process `Pid` fetched through an `erlang:process_info(Pid)` function call.

If the initial call is to one of the system defined behaviors such as `gen_server` or `gen_event`, it is translated to more useful information. If a `gen_server` is spawned, the returned `Module` is the name of the callback module and `Function` is `init` (the function that initiates the new server).

A `supervisor` and a `supervisor_bridge` are also `gen_server` processes. In order to return information that this process is a supervisor and the name of the call-back module, `Module` is `supervisor` and `Function` is the name of the supervisor callback module. `Arity` is `1` since the `init/1` function is called initially in the callback module.

By default, {proc_lib,init_p,5} is returned if no information about the initial call can be found. It is assumed that the caller knows that the process has been spawned with the proc_lib module.

`hibernate(Module, Function, Args)`

Types:

- Module = Function = atom()
- Args = [term()]

This function does the same as (and does call) the BIF [hibernate/3], but ensures that exception handling and logging continues to work as expected when the process wakes up. Always use this function instead of the BIF for processes started using proc_lib functions.

SEE ALSO

[error_logger(3)]

proplists

Erlang Module

Property lists are ordinary lists containing entries in the form of either tuples, whose first elements are keys used for lookup and insertion, or atoms, which work as shorthand for tuples {Atom, true}. (Other terms are allowed in the lists, but are ignored by this module.) If there is more than one entry in a list for a certain key, the first occurrence normally overrides any later (irrespective of the arity of the tuples).

Property lists are useful for representing inherited properties, such as options passed to a function where a user may specify options overriding the default settings, object properties, annotations, etc.

Exports

`append_values(Key, List) -> List`

Types:

- Key = term()
- List = [term()]

Similar to `get_all_values/2`, but each value is wrapped in a list unless it is already itself a list, and the resulting list of lists is concatenated. This is often useful for "incremental" options; e.g., `append_values(a, [{a, [1,2]}, {b, 0}, {a, 3}, {c, -1}, {a, [4]}])` will return the list `[1,2,3,4]`.

`compact(List) -> List`

Types:

- List = [term()]

Minimizes the representation of all entries in the list. This is equivalent to `[property(P) || P <- List]`.

See also: `property/1`, `unfold/1`.

`delete(Key, List) -> List`

Types:

- Key = term()
- List = [term()]

Deletes all entries associated with `Key` from `List`.

`expand(Expansions, List) -> List`

Types:

- Key = term()
- Expansions = [{Property,[term()]}]
- Property = atom() | tuple()

Expands particular properties to corresponding sets of properties (or other terms). For each pair {`Property, Expansion`} in `Expansions`, if `E` is the first entry in `List` with the same key as `Property`, and `E` and `Property` have equivalent normal forms, then `E` is replaced with the terms in `Expansion`, and any following entries with the same key are deleted from `List`.

For example, the following expressions all return `[fie, bar, baz, fum]`:

```
expand([{foo, [bar, baz]}],
[fie, foo, fum])
expand([{{foo, true}, [bar, baz]}],
[fie, foo, fum])
expand([{{foo, false}, [bar, baz]}],
[fie, {foo, false}, fum])
```

However, no expansion is done in the following call:

```
expand([{{foo, true}, [bar, baz]}],
[{foo, false}, fie, foo, fum])
```

because {`foo, false`} shadows `foo`.

Note that if the original property term is to be preserved in the result when expanded, it must be included in the expansion list. The inserted terms are not expanded recursively. If `Expansions` contains more than one property with the same key, only the first occurrance is used.

See also: `normalize/2`.

`get_all_values(Key, List) -> [term()]`

Types:

- Key = term()
- List = [term()]

Similar to `get_value/2`, but returns the list of values for *all* entries {`Key, Value`} in `List`. If no such entry exists, the result is the empty list.

See also: `get_value/2`.

`get_bool(Key, List) -> bool()`

Types:

- Key = term()
- List = [term()]

Returns the value of a boolean key/value option. If `lookup(Key, List)` would yield {`Key, true`}, this function returns `true`; otherwise `false` is returned.

See also: `get_value/2`, `lookup/2`.

`get_keys(List) -> [term()]`

Types:

- List = [term()]

Returns an unordered list of the keys used in `List`, not containing duplicates.

`get_value(Key, List) -> term()`

Types:

- Key = term()
- List = [term()]

Equivalent to `get_value(Key, List, undefined)`.

`get_value(Key, List, Default) -> term()`

Types:

- Key = term()
- Default = term()
- List = [term()]

Returns the value of a simple key/value property in `List`. If `lookup(Key, List)` would yield `{Key, Value}`, this function returns the corresponding `Value`, otherwise `Default` is returned.

See also: `get_all_values/2`, `get_bool/2`, `get_value/1`, `lookup/2`.

`is_defined(Key, List) -> bool()`

Types:

- Key = term()
- List = [term()]

Returns `true` if `List` contains at least one entry associated with `Key`, otherwise `false` is returned.

`lookup(Key, List) -> none | tuple()`

Types:

- Key = term()
- List = [term()]

Returns the first entry associated with `Key` in `List`, if one exists, otherwise returns `none`. For an atom `A` in the list, the tuple `{A, true}` is the entry associated with `A`.

See also: `get_bool/2`, `get_value/2`, `lookup_all/2`.

`lookup_all(Key, List) -> [tuple()]`

Types:

- Key = term()
- List = [term()]

Returns the list of all entries associated with `Key` in `List`. If no such entry exists, the result is the empty list.

See also: `lookup/2`.

`normalize(List, Stages) -> List`

Types:

- List = [term()]
- Stages = [Operation]
- Operation = {aliases, Aliases} | {negations, Negations} | {expand, Expansions}
- Aliases = [{Key, Key}]
- Negations = [{Key, Key}]
- Key = term()
- Expansions = [{Property, [term()]}]
- Property = atom() | tuple()

Passes `List` through a sequence of substitution/expansion stages. For an `aliases` operation, the function `substitute_aliases/2` is applied using the given list of aliases; for a `negations` operation, `substitute_negations/2` is applied using the given negation list; for an `expand` operation, the function `expand/2` is applied using the given list of expansions. The final result is automatically compacted (cf. `compact/1`).

Typically you want to substitute negations first, then aliases, then perform one or more expansions (sometimes you want to pre-expand particular entries before doing the main expansion). You might want to substitute negations and/or aliases repeatedly, to allow such forms in the right-hand side of aliases and expansion lists.

See also: `compact/1`, `expand/2`, `substitute_aliases/2`, `substitute_negations/2`.

`property(Property) -> Property`

Types:

- Property = atom() | tuple()

Creates a normal form (minimal) representation of a property. If `Property` is {`Key`, `true`} where `Key` is an atom, this returns `Key`, otherwise the whole term `Property` is returned.

See also: `property/2`.

`property(Key, Value) -> Property`

Types:

- Key = term()
- Value = term()
- Property = atom() | tuple()

Creates a normal form (minimal) representation of a simple key/value property. Returns `Key` if `Value` is `true` and `Key` is an atom, otherwise a tuple {`Key`, `Value`} is returned.

See also: `property/1`.

`split(List, Keys) -> {Lists, Rest}`

Types:

- List = [term()]
- Keys = [term()]
- Lists = [[term()]]
- Rest = [term()]

Partitions `List` into a list of sublists and a remainder. `Lists` contains one sublist for each key in `Keys`, in the corresponding order. The relative order of the elements in each sublist is preserved from the original `List`. `Rest` contains the elements in `List` that are not associated with any of the given keys, also with their original relative order preserved.

Example: split([{c, 2}, {e, 1}, a, {c, 3, 4}, d, {b, 5}, b], [a, b, c])

returns

{[[a], [{b, 5}, b],[{c, 2}, {c, 3, 4}]], [{e, 1}, d]}

`substitute_aliases(Aliases, List) -> List`

Types:

- Aliases = [{Key, Key}]
- Key = term()
- List = [term()]

Substitutes keys of properties. For each entry in `List`, if it is associated with some key `K1` such that {`K1, K2`} occurs in `Aliases`, the key of the entry is changed to `Key2`. If the same `K1` occurs more than once in `Aliases`, only the first occurrence is used.

Example: `substitute_aliases([{color, colour}], L)` will replace all tuples {`color, ...`} in `L` with {`colour, ...`}, and all atoms `color` with `colour`.

See also: `normalize/2`, `substitute_negations/2`.

`substitute_negations(Negations, List) -> List`

Types:

- Negations = [{Key, Key}]
- Key = term()
- List = [term()]

Substitutes keys of boolean-valued properties and simultaneously negates their values. For each entry in `List`, if it is associated with some key `K1` such that {`K1, K2`} occurs in `Negations`, then if the entry was {`K1, true`} it will be replaced with {`K2, false`}, otherwise it will be replaced with {`K2, true`}, thus changing the name of the option and simultaneously negating the value given by `get_bool(List)`. If the same `K1` occurs more than once in `Negations`, only the first occurrence is used.

Example: `substitute_negations([{no_foo, foo}], L)` will replace any atom `no_foo` or tuple {`no_foo, true`} in `L` with {`foo, false`}, and any other tuple {`no_foo, ...`} with {`foo, true`}.

See also: `get_bool/2`, `normalize/2`, `substitute_aliases/2`.

`unfold(List) -> List`

Types:

- List = [term()]

Unfolds all occurences of atoms in `List` to tuples {`Atom, true`}.

qlc

Erlang Module

The qlc module provides a query interface to Mnesia, ETS, Dets and other data structures that implement an iterator style traversal of objects.

Overview

The qlc module implements a query interface to *QLC tables*. Typical QLC tables are ETS, Dets, and Mnesia tables. There is also support for user defined tables, see the Implementing a QLC table [page 292] section. A *query* is stated using *Query List Comprehensions* (QLCs). The answers to a query are determined by data in QLC tables that fulfill the constraints expressed by the QLCs of the query. QLCs are similar to ordinary list comprehensions as described in the Erlang Reference Manual and Programming Examples except that variables introduced in patterns cannot be used in list expressions. In fact, in the absence of optimizations and options such as cache and unique (see below), every QLC free of QLC tables evaluates to the same list of answers as the identical ordinary list comprehension.

While ordinary list comprehensions evaluate to lists, calling qlc:q/1,2 [page 298] returns a *Query Handle*. To obtain all the answers to a query, qlc:eval/1,2 [page 295] should be called with the query handle as first argument. Query handles are essentially functional objects ("funs") created in the module calling q/1,2. As the funs refer to the module's code, one should be careful not to keep query handles too long if the module's code is to be replaced. Code replacement is described in the [Erlang Reference Manual]. The list of answers can also be traversed in chunks by use of a *Query Cursor*. Query cursors are created by calling qlc:cursor/1,2 [page 294] with a query handle as first argument. Query cursors are essentially Erlang processes. One answer at a time is sent from the query cursor process to the process that created the cursor.

Syntax

Syntactically QLCs have the same parts as ordinary list comprehensions:

```
[Expression || Qualifier1, Qualifier2, ...]
```

Expression (the *template*) is an arbitrary Erlang expression. Qualifiers are either *filters* or *generators*. Filters are Erlang expressions returning bool(). Generators have the form Pattern <- ListExpression, where ListExpression is an expression evaluating to a query handle or a list. Query handles are returned from qlc:table/2, qlc:append/1,2, qlc:sort/1,2, qlc:keysort/2,3, qlc:q/1,2, and qlc:string_to_handle/1,2,3.

Evaluation

The evaluation of a query handle begins by the inspection of options and the collection of information about tables. As a result qualifiers are modified during the optimization phase. Next all list expressions are evaluated. If a cursor has been created evaluation takes place in the cursor process. For those list expressions that are QLCs, the list expressions of the QLCs' generators are evaluated as well. One has to be careful if list expressions have side effects since the order in which list expressions are evaluated is unspecified. Finally the answers are found by evaluating the qualifiers from left to right, backtracking when some filter returns `false`, or collecting the template when all filters return `true`.

Filters that do not return `bool()` but fail are handled differently depending on their syntax: if the filter is a guard it returns `false`, otherwise the query evaluation fails. This behavior makes it possible for the `qlc` module to do some optimizations without affecting the meaning of a query. For example, when some position of a table is compared to one or more constants, only the objects with matching values are candidates for further evaluation. The other objects are guaranteed to make the filter return `false`, but never fail. The (small) set of candidate objects can often be found by looking up some key values of the table or by traversing the table using a match specification. It is necessary to place the guard filters immediately after the table's generator, otherwise the candidate objects will not be restricted to a small set. The reason is that objects that could make the query evaluation fail must not be excluded by looking up a key or running a match specification.

Join

The `qlc` module supports fast join of two query handles. Fast join is possible if some position (`P1`) of one query handler is compared to or matched against some position (`P2`) of another query handle. Two fast join methods have been implemented:

- Lookup join traverses all objects of one query handle and finds objects of the other handle (a QLC table) such that the values at `P1` and `P2` match. The `qlc` module does not create any indices but looks up values using the key position and the indexed positions of the QLC table.
- Merge join sorts the objects of each query handle if necessary and filters out objects where the values at `P1` and `P2` do not compare equal. If there are many objects with the same value of `P2` a temporary file will be used for the equivalence classes.

The `qlc` module warns at compile time if a QLC combines query handles in such a way that more than one join is possible. In other words, there is no query planner that can choose a good order between possible join operations. It is up to the user to order the joins by introducing query handles.

The join is to be expressed as a guard filter. The filter must be placed immediately after the two joined generators, possibly after guard filters that use variables from no other generators but the two joined generators. The `qlc` module inspects the operands of `=:=/2`, `==/2`, `is_record/2`, `element/2`, and logical operators (`and/2`, `or/2`, `andalso/2`, `orelse/2`, `xor/2`) when determining which joins to consider.

Common options

The following options are accepted by `cursor/2`, `eval/2`, `fold/4`, and `info/2`:

- {`cache_all, Cache`} where `Cache` is equal to `ets` or `list` adds a {`cache,Cache`} option to every list expression of the query except tables and lists. Default is {`cache_all,no`}. The option `cache_all` is equivalent to {`cache_all,ets`}.
- {`max_list_size, MaxListSize`} where `MaxListSize` is the size in bytes of terms on the external format. If the accumulated size of collected objects exceeds `MaxListSize` the objects are written onto a temporary file. This option is used by the {`cache,list`} option as well as by the merge join method. Default is 512*1024 bytes.
- {`tmpdir_usage, TmpFileUsage`} determines the action taken when `qlc` is about to create temporary files on the directory set by the `tmpdir` option. If the value is `not_allowed` an error tuple is returned, otherwise temporary files are created as needed. Default is `allowed` which means that no further action is taken. The values `info_msg`, `warning_msg`, and `error_msg` mean that the function with the corresponding name in the module `error_logger` is called for printing some information (currently the stacktrace).
- {`tmpdir, TempDirectory`} sets the directory used by merge join for temporary files and by the {`cache,list`} option. The option also overrides the `tmpdir` option of `keysort/3` and `sort/2`. The default value is `""` which means that the directory returned by `file:get_cwd()` is used.
- {`unique_all, true`} adds a {`unique,true`} option to every list expression of the query. Default is {`unique_all,false`}. The option `unique_all` is equivalent to {`unique_all,true`}.

Common data types

- `QueryCursor = {qlc_cursor, term()}`
- `QueryHandle = {qlc_handle, term()}`
- `QueryHandleOrList = QueryHandle | list()`
- `Answers = [Answer]`
- `Answer = term()`
- `AbstractExpression` = -parse trees for Erlang expressions, see the [abstract format] documentation in the ERTS User's Guide-
- `MatchExpression` = -matchspecifications, see the [match specification] documentation in the ERTS User's Guide and ms_transform(3) [page 254]-
- `SpawnOptions = default | spawn_options()`
- `SortOptions = [SortOption] | SortOption`
- `SortOption = {compressed, bool()} | {no_files, NoFiles} | {order, Order} | {size, Size} | {tmpdir, TempDirectory} | {unique, bool()}` -see file_sorter(3) [page 163]-
- `Order = ascending | descending | OrderFun`
- `OrderFun = fun(term(), term()) -> bool()`
- `TempDirectory = "" | filename()`

- `Size = int() > 0`
- `NoFiles = int() > 1`
- `KeyPos = int() > 0 | [int() > 0]`
- `MaxListSize = int() >= 0`
- `bool() = true | false`
- `Cache = ets | list | no`
- `TmpFileUsage = allowed | not_allowed | info_msg | warning_msg | error_msg`
- `filename()` = -see filename(3) [page 171]-
- `spawn_options()` = -see [erlang(3)]-

Getting started

As already mentioned queries are stated in the list comprehension syntax as described in the [Erlang Reference Manual]. In the following some familiarity with list comprehensions is assumed. There are examples in [Programming Examples] that can get you started. It should be stressed that list comprehensions do not add any computational power to the language; anything that can be done with list comprehensions can also be done without them. But they add a syntax for expressing simple search problems which is compact and clear once you get used to it.

Many list comprehension expressions can be evaluated by the `qlc` module. Exceptions are expressions such that variables introduced in patterns (or filters) are used in some generator later in the list comprehension. As an example consider an implementation of lists:append(L): `[X ||Y <- L, X <- Y]`. Y is introduced in the first generator and used in the second. The ordinary list comprehension is normally to be preferred when there is a choice as to which to use. One difference is that `qlc:eval/1,2` collects answers in a list which is finally reversed, while list comprehensions collect answers on the stack which is finally unwound.

What the `qlc` module primarily adds to list comprehensions is that data can be read from QLC tables in small chunks. A QLC table is created by calling `qlc:table/2`. Usually `qlc:table/2` is not called directly from the query but via an interface function of some data structure. There are a few examples of such functions in Erlang/OTP: `mnesia:table/1,2`, `ets:table/1,2`, and `dets:table/1,2`. For a given data structure there can be several functions that create QLC tables, but common for all these functions is that they return a query handle created by `qlc:table/2`. Using the QLC tables provided by OTP is probably sufficient in most cases, but for the more advanced user the section Implementing a QLC table [page 292] describes the implementation of a function calling `qlc:table/2`.

Besides `qlc:table/2` there are other functions that return query handles. They might not be used as often as tables, but are useful from time to time. `qlc:append` traverses objects from several tables or lists after each other. If, for instance, you want to traverse all answers to a query QH and then finish off by a term `{finished}`, you can do that by calling `qlc:append(QH, [{finished}])`. `append` first returns all objects of QH, then `{finished}`. If there is one tuple `{finished}` among the answers to QH it will be returned twice from `append`.

As another example, consider concatenating the answers to two queries QH1 and QH2 while removing all duplicates. The means to accomplish this is to use the `unique` option:

```
qlc:q([X || X <- qlc:append(QH1, QH2)], {unique, true})
```

The cost is substantial: every returned answer will be stored in an ETS table. Before returning an answer it is looked up in the ETS table to check if it has already been returned. Without the `unique` options all answers to QH1 would be returned followed by all answers to QH2. The `unique` options keeps the order between the remaining answers.

If the order of the answers is not important there is the alternative to sort the answers uniquely:

```
qlc:sort(qlc:q([X || X <- qlc:append(QH1, QH2)], {unique, true})).
```

This query also removes duplicates but the answers will be sorted. If there are many answers temporary files will be used. Note that in order to get the first unique answer all answers have to be found and sorted.

To return just a few answers cursors can be used. The following code returns no more than five answers using an ETS table for storing the unique answers:

```
C = qlc:cursor(qlc:q([X || X <- qlc:append(QH1, QH2)],{unique,true})),
R = qlc:next_answers(C, 5),
ok = qlc:delete_cursor(C),
R.
```

Query list comprehensions are convenient for stating constraints on data from two or more tables. An example that does a natural join on two query handles on position 2:

```
qlc:q([{X1,X2,X3,Y1} ||
          {X1,X2,X3} <- QH1,
          {Y1,Y2} <- QH2,
          X2 =:= Y2])
```

The `qlc` module will evaluate this differently depending on the query handles `QH1` and `QH2`. If, for example, `X2` is matched against the key of a QLC table the lookup join method will traverse the objects of `QH2` while looking up key values in the table. On the other hand, if neither `X2` nor `Y2` is matched against the key or an indexed position of a QLC table, the merge join method will make sure that `QH1` and `QH2` are both sorted on position 2 and next do the join by traversing the objects one by one.

The `join` option can be used to force the `qlc` module to use a certain join method. For the rest of this section it is assumed that the excessively slow join method called “nested loop” has been chosen:

```
qlc:q([{X1,X2,X3,Y1} ||
          {X1,X2,X3} <- QH1,
          {Y1,Y2} <- QH2,
          X2 =:= Y2],
      {join, nested_loop})
```

In this case the filter will be applied to every possible pair of answers to QH1 and QH2, one at a time. If there are M answers to QH1 and N answers to QH2 the filter will be run M*N times.

If QH2 is a call to the function for `gb_trees` as defined in the Implementing a QLC table [page 292] section, `gb_table:table/1`, the iterator for the gb-tree will be initiated for each answer to QH1 after which the objects of the gb-tree will be returned one by one. This is probably the most efficient way of traversing the table in that case since it takes minimal computational power to get the following object. But if QH2 is not a table but a more complicated QLC, it can be more efficient use some RAM

memory for collecting the answers in a cache, particularly if there are only a few answers. It must then be assumed that evaluating QH2 has no side effects so that the meaning of the query does not change if QH2 is evaluated only once. One way of caching the answers is to evaluate QH2 first of all and substitute the list of answers for QH2 in the query. Another way is to use the `cache` option. It is stated like this:

```
QH2' = qlc:q([X || X <- QH2], {cache, ets})
```

or just

```
QH2' = qlc:q([X || X <- QH2], cache)
```

The effect of the `cache` option is that when the generator QH2' is run the first time every answer is stored in an ETS table. When next answer of QH1 is tried, answers to QH2' are copied from the ETS table which is very fast. As for the `unique` option the cost is a possibly substantial amount of RAM memory. The {`cache`,`list`} option offers the possibility to store the answers in a list on the process heap. While this has the potential of being faster than ETS tables since there is no need to copy answers from the table it can often result in slower evaluation due to more garbage collections of the process' heap as well as increased RAM memory consumption due to larger heaps. Another drawback with cache lists is that if the size of the list exceeds a limit a temporary file will be used. Reading the answers from a file is very much slower than copying them from an ETS table. But if the available RAM memory is scarce setting the limit [page 289] to some low value is an alternative.

There is an option `cache_all` that can be set to `ets` or `list` when evaluating a query. It adds a `cache` or {`cache`,`list`} option to every list expression except QLC tables and lists on all levels of the query. This can be used for testing if caching would improve efficiency at all. If the answer is yes further testing is needed to pinpoint the generators that should be cached.

Implementing a QLC table

As an example of how to use the qlc:table/2 [page 298] function the implementation of a QLC table for the gb_trees [page 183] module is given:

```
-module(gb_table).

-export([table/1]).

table(T) ->
    TF = fun() -> qlc_next(gb_trees:next(gb_trees:iterator(T))) end,
    InfoFun = fun(num_of_objects) -> gb_trees:size(T);
                 (keypos) -> 1;
                 (_) -> undefined
              end,
    LookupFun =
        fun(1, Ks) ->
                lists:flatmap(fun(K) ->
                                  case gb_trees:lookup(K, T) of
                                      {value, V} -> [{K,V}];
                                      none -> []
                                  end
                              end, Ks)
        end,
```

```
    FormatFun =
        fun({all, NElements, ElementFun}) ->
                Vals = gb_nodes(T, NElements, ElementFun),
                {gb_trees, from_orddict, [Vals]};
           ({lookup, 1, KeyValues, _NElements, ElementFun}) ->
                ValsS = io_lib:format("gb_trees:from_orddict(~w)",
                                      [gb_nodes(T, infinity, ElementFun)]),
                io_lib:format("lists:flatmap(fun(K) -> "
                              "case gb_trees:lookup(K, ~s) of "
                              "{value, V} -> [{K,V}];none -> [] end "
                              "end, ~w)",
                              [ValsS, [ElementFun(KV) || KV <- KeyValues]])
        end,
    qlc:table(TF, [{info_fun, InfoFun}, {format_fun, FormatFun},
                   {lookup_fun, LookupFun}]).

qlc_next({X, V, S}) ->
    [{X,V} | fun() -> qlc_next(gb_trees:next(S)) end];
qlc_next(none) ->
    [].

gb_nodes(T, infinity, ElementFun) ->
    gb_nodes(T, -1, ElementFun);
gb_nodes(T, NElements, ElementFun) ->
    gb_iter(gb_trees:iterator(T), NElements, ElementFun).

gb_iter(_I, 0, _EFun) ->
    '...';
gb_iter(I0, N, EFun) ->
    case gb_trees:next(I0) of
        {X, V, I} ->
            [EFun({X,V}) | gb_iter(I, N-1, EFun)];
        none ->
            []
    end.
```

TF is the traversal function. The qlc module requires that there is a way of traversing all objects of the data structure; in gb_trees there is an iterator function suitable for that purpose. Note that for each object returned a new fun is created. As long as the list is not terminated by [] it is assumed that the tail of the list is a nullary function and that calling the function returns further objects (and functions).

The lookup function is optional. It is assumed that the lookup function always finds values much faster than it would take to traverse the table. The first argument is the position of the key. Since qlc_next returns the objects as {Key,Value} pairs the position is 1. Note that the lookup function should return {Key,Value} pairs, just as the traversal function does.

The format function is also optional. It is called by qlc:info to give feedback at runtime of how the query will be evaluated. One should try to give as good feedback as possible without showing too much details. In the example at most 7 objects of the table are shown. The format function handles two cases: all means that all objects of the table will be traversed; {lookup,1,KeyValues} means that the lookup function will be used for looking up key values.

Whether the whole table will be traversed or just some keys looked up depends on how the query is stated. If the query has the form

```
qlc:q([T || P <- LE, F])
```

and P is a tuple, the `qlc` module analyzes P and F in compile time to find positions of the tuple P that are matched or compared to constants. If such a position at runtime turns out to be the key position, the lookup function can be used, otherwise all objects of the table have to be traversed. It is the info function `InfoFun` that returns the key position. There can be indexed positions as well, also returned by the info function. An index is an extra table that makes lookup on some position fast. Mnesia maintains indices upon request, thereby introducing so called secondary keys. The `qlc` module prefers to look up objects using the key before secondary keys regardless of the number of constants to look up.

Exports

`append(QHL) -> QH`

Types:

- QHL = [QueryHandleOrList]
- QH = QueryHandle

Returns a query handle. When evaluating the query handle `QH` all answers to the first query handle in `QHL` is returned followed by all answers to the rest of the query handles in `QHL`.

`append(QH1, QH2) -> QH3`

Types:

- QH1 = QH2 = QueryHandleOrList
- QH3 = QueryHandle

Returns a query handle. When evaluating the query handle `QH3` all answers to `QH1` are returned followed by all answers to `QH2`.

`append(QH1,QH2)` is equivalent to `append([QH1,QH2])`.

`cursor(QueryHandleOrList [, Options]) -> QueryCursor`

Types:

- Options = [Option] | Option
- Option = {cache_all, Cache} | cache_all | {max_list_size, MaxListSize} | {spawn_options, SpawnOptions} | {tmpdir_usage, TmpFileUsage} | {tmpdir, TempDirectory} | {unique_all, bool()} | unique_all

Creates a query cursor and makes the calling process the owner of the cursor. The cursor is to be used as argument to `next_answers/1,2` and (eventually) `delete_cursor/1`. Calls `erlang:spawn_opt` to spawn and link a process which will evaluate the query handle. The value of the option `spawn_options` is used as last argument when calling `spawn_opt`. The default value is `[link]`.

```
1> QH = qlc:q([{X,Y} || X <- [a,b], Y <- [1,2]]),
QC = qlc:cursor(QH),
qlc:next_answers(QC, 1).
[{a,1}]
2> qlc:next_answers(QC, 1).
[{a,2}]
3> qlc:next_answers(QC, all_remaining).
[{b,1},{b,2}]
4> qlc:delete_cursor(QC).
ok
```

`delete_cursor(QueryCursor) -> ok`

Deletes a query cursor. Only the owner of the cursor can delete the cursor.

`eval(QueryHandleOrList [, Options]) -> Answers | Error`
`e(QueryHandleOrList [, Options]) -> Answers`

Types:

- Options = [Option] | Option
- Option = {cache_all, Cache} | cache_all | {max_list_size, MaxListSize} | {tmpdir_usage, TmpFileUsage} | {tmpdir, TempDirectory} | {unique_all, bool()} | unique_all
- Error = {error, module(), Reason}
- Reason =-as returned by file_sorter(3)-

Evaluates a query handle in the calling process and collects all answers in a list.

```
1> QH = qlc:q([{X,Y} || X <- [a,b], Y <- [1,2]]),
qlc:eval(QH).
[{a,1},{a,2},{b,1},{b,2}]
```

`fold(Function, Acc0, QueryHandleOrList [, Options]) -> Acc1 | Error`

Types:

- Function = fun(Answer, AccIn) -> AccOut
- Acc0 = Acc1 = AccIn = AccOut = term()
- Options = [Option] | Option
- Option = {cache_all, Cache} | cache_all | {max_list_size, MaxListSize} | {tmpdir_usage, TmpFileUsage} | {tmpdir, TempDirectory} | {unique_all, bool()} | unique_all
- Error = {error, module(), Reason}
- Reason =-as returned by file_sorter(3)-

Calls `Function` on successive answers to the query handle together with an extra argument `AccIn`. The query handle and the function are evaluated in the calling process. `Function` must return a new accumulator which is passed to the next call. `Acc0` is returned if there are no answers to the query handle.

```
1> QH = [1,2,3,4,5,6],
qlc:fold(fun(X, Sum) -> X + Sum end, 0, QH).
21
```

`format_error(Error) -> Chars`

Types:

- Error = {error, module(), term()}
- Chars = [char() | Chars]

Returns a descriptive string in English of an error tuple returned by some of the functions of the `qlc` module or the parse transform. This function is mainly used by the compiler invoking the parse transform.

`info(QueryHandleOrList [, Options]) -> Info`

Types:

- Options = [Option] | Option
- Option = EvalOption | ReturnOption
- EvalOption = {cache_all, Cache} | cache_all | {max_list_size, MaxListSize} | {tmpdir_usage, TmpFileUsage} | {tmpdir, TempDirectory} | {unique_all, bool()} | unique_all
- ReturnOption = {depth, Depth} | {flat, bool()} | {format, Format} | {n_elements, NElements}
- Depth = infinity | int() >= 0
- Format = abstract_code | string
- NElements = infinity | int() > 0
- Info = AbstractExpression | string()

Returns information about a query handle. The information describes the simplifications and optimizations that are the results of preparing the query for evaluation. This function is probably useful mostly during debugging.

The information has the form of an Erlang expression where QLCs most likely occur. Depending on the format functions of mentioned QLC tables it may not be absolutely accurate.

The default is to return a sequence of QLCs in a block, but if the option {`flat`,`false`} is given, one single QLC is returned. The default is to return a string, but if the option {`format`,`abstract_code`} is given, abstract code is returned instead. In the abstract code port identifiers, references, and pids are represented by strings. The default is to return all elements in lists, but if the {`n_elements`,`NElements`} option is given, only a limited number of elements are returned. The default is to show all of objects and match specifications, but if the {`depth`,`Depth`} option is given, parts of terms below a certain depth are replaced by `'...'`.

```
1> QH = qlc:q([{X,Y} || X <- [x,y], Y <- [a,b]]),
io:format("~s~n", [qlc:info(QH, unique_all)]).
begin
    V1 =
        qlc:q([SQV ||
                   SQV <- [x,y]],
              [{unique,true}]),
    V2 =
        qlc:q([SQV ||
                   SQV <- [a,b]],
              [{unique,true}]),
    qlc:q([{X,Y} ||
               X <- V1,
```

```
                    Y <- V2],
              [{unique,true}])
end
```

In this example two simple QLCs have been inserted just to hold the {unique,true} option.

```
1> E1 = ets:new(e1, []),
E2 = ets:new(e2, []),
true = ets:insert(E1, [{1,a},{2,b}]),
true = ets:insert(E2, [{a,1},{b,2}]),
Q = qlc:q([{X,Z,W} ||
{X, Z} <- ets:table(E1),
{W, Y} <- ets:table(E2),
X =:= Y]),
io:format("~s~n", [qlc:info(Q)]).
begin
    V1 =
        qlc:q([P0 ||
                   P0 = {W,Y} <- ets:table(17)]),
    V2 =
        qlc:q([[G1|G2] ||
                   G2 <- V1,
                   G1 <- ets:table(16),
                   element(2, G1) =:= element(1, G2)],
              [{join,lookup}]),
    qlc:q([{X,Z,W} ||
               [{X,Z}|{W,Y}] <- V2,
               X =:= Y])
end
```

In this example the query list comprehension V2 has been inserted to show the joined generators and the join method chosen. A convention is used for lookup join: the first generator (G2) is the one traversed, the second one (G1) is the table where constants are looked up.

`keysort(KeyPos, QH1 [, SortOptions]) -> QH2`

Types:

- QH1 = QueryHandleOrList
- QH2 = QueryHandle

Returns a query handle. When evaluating the query handle QH2 the answers to the query handle QH1 are sorted by file_sorter:keysort/4 [page 163] according to the options.

The sorter will use temporary files only if QH1 does not evaluate to a list and the size of the binary representation of the answers exceeds Size bytes, where Size is the value of the size option.

`next_answers(QueryCursor [, NumberOfAnswers]) -> Answers | Error`

Types:

- NumberOfAnswers = all_remaining | int() > 0
- Error = {error, module(), Reason}
- Reason =-as returned by file_sorter(3)-

Returns some or all of the remaining answers to a query cursor. Only the owner of `Cursor` can retrieve answers.

The optional argument `NumberOfAnswers`determines the maximum number of answers returned. The default value is `10`. If less than the requested number of answers is returned, subsequent calls to `next_answers` will return `[]`.

`q(QueryListComprehension [, Options]) -> QueryHandle`

Types:

- QueryListComprehension = -literal query listcomprehension-
- Options = [Option] | Option
- Option = {max_lookup, MaxLookup} | {cache, Cache} | cache | {join, Join} | {lookup, Lookup} | {unique, bool()} | unique
- MaxLookup = int() >= 0 | infinity
- Join = any | lookup | merge | nested_loop
- Lookup = bool() | any

Returns a query handle for a query list comprehension. The query list comprehension must be the first argument to `qlc:q/1,2` or it will be evaluated as an ordinary list comprehension. It is also necessary to add the line

```
-include_lib("stdlib/include/qlc.hrl").
```

to the source file. This causes a parse transform to substitute a fun for the query list comprehension. The (compiled) fun will be called when the query handle is evaluated.

When calling `qlc:q/1,2` from the Erlang shell the parse transform is automatically called. When this happens the fun substituted for the query list comprehension is not compiled but will be evaluated by `erl_eval(3)`. This is also true when expressions are evaluated by means of `file:eval/1,2` or in the debugger.

To be very explicit, this will not work:

```
...
A = [X || {X} <- [{1},{2}]],
QH = qlc:q(A),
...
```

The variable `A` will be bound to the evaluated value of the list comprehension (`[1,2]`). The compiler complains with an error message (“argument is not a query list comprehension”); the shell process stops with a `badarg` reason.

The {`cache,ets`} option can be used to cache the answers to a query list comprehension. The answers are stored in one ETS table for each cached query list comprehension. When a cached query list comprehension is evaluated again, answers are fetched from the table without any further computations. As a consequence, when all answers to a cached query list comprehension have been found, the ETS tables used for caching answers to the query list comprehension’s qualifiers can be emptied. The option `cache` is equivalent to {`cache,ets`}.

The {`cache,list`} option can be used to cache the answers to a query list comprehension just like {`cache,ets`}. The difference is that the answers are kept in a list (on the process heap). If the answers would occupy more than a certain amount of RAM memory a temporary file is used for storing the answers. The option `max_list_size` sets the limit in bytes and the temporary file is put on the directory set by the `tmpdir` option.

The cache option has no effect if it is known that the query list comprehension will be evaluated at most once. This is always true for the top-most query list comprehension and also for the list expression of the first generator in a list of qualifiers. Note that in the presence of side effects in filters or callback functions the answers to query list comprehensions can be affected by the cache option.

The {unique,true} option can be used to remove duplicate answers to a query list comprehension. The unique answers are stored in one ETS table for each query list comprehension. The table is emptied every time it is known that there are no more answers to the query list comprehension. The option unique is equivalent to {unique,true}. If the unique option is combined with the {cache,ets} option, two ETS tables are used, but the full answers are stored in one table only. If the unique option is combined with the {cache,list} option the answers are sorted twice using keysort/3; once to remove duplicates, and once to restore the order.

The cache and unique options apply not only to the query list comprehension itself but also to the results of looking up constants, running match specifications, and joining handles.

```
1> Q = qlc:q([{A,X,Z,W} ||
A <- [a,b,c],
{X,Z} <- [{a,1},{b,4},{c,6}],
{W,Y} <- [{2,a},{3,b},{4,c}],
X =:= Y],
{cache, list}),
io:format("~s~n", [qlc:info(Q)]).
begin
    V1 =
        qlc:q([P0 ||
                   P0 = {X,Z} <- qlc:keysort(1, [{a,1},{b,4},{c,6}], [])]),
    V2 =
        qlc:q([P0 ||
                   P0 = {W,Y} <- qlc:keysort(2, [{2,a},{3,b},{4,c}], [])]),
    V3 =
        qlc:q([[G1|G2] ||
                   G1 <- V1,
                   G2 <- V2,
                   element(1, G1) == element(2, G2)],
              [{join,merge},{cache,list}]),
    qlc:q([{A,X,Z,W} ||
               A <- [a,b,c],
               [{X,Z}|{W,Y}] <- V3,
               X =:= Y])
end
```

In this example the cached results of the merge join are traversed for each value of A. Note that without the cache option the join would have been carried out three times, once for each value of A

sort/1,2 and keysort/2,3 can also be used for caching answers and for removing duplicates. When sorting answers are cached in a list, possibly stored on a temporary file, and no ETS tables are used.

Sometimes (see qlc:table/2 [page 303] below) traversal of tables can be done by looking up key values, which is assumed to be fast. Under certain (rare) circumstances it could happen that there are too many key values to look up. The {max_lookup,MaxLookup}

option can then be used to limit the number of lookups: if more than MaxLookup lookups would be required no lookups are done but the table traversed instead. The default value is infinity which means that there is no limit on the number of keys to look up.

```
1> T = gb_trees:empty(),
QH = qlc:q([X || {{X,Y},_} <- gb_table:table(T),
((X =:= 1) or (X =:= 2)),
((Y =:= a) or (Y =:= b) or (Y =:= c))]),
io:format("~s~n", [qlc:info(QH)]).
ets:match_spec_run(
       lists:flatmap(fun(K) ->
                           case
                               gb_trees:lookup(K,
                                               gb_trees:from_orddict([]))
                           of
                               {value,V} ->
                                   [{K,V}];
                               none ->
                                   []
                           end
                     end,
                     [{1,a},{1,b},{1,c},{2,a},{2,b},{2,c}]),
       ets:match_spec_compile([{{{'$1','$2'},'_'},
                                [{'andalso',
                                  {'or',
                                   {'=:=','$1',1},
                                   {'=:=','$1',2}},
                                  {'or',
                                   {'or',
                                    {'=:=','$2',a},
                                    {'=:=','$2',b}},
                                   {'=:=','$2',c}}}],
                                ['$1']}]))
```

In this example using the gb_table module from the Implementing a QLC table [page 292] section there are six keys to look up: {1,a}, {1,b}, {1,c}, {2,a}, {2,b}, and {2,c}. The reason is that the two elements of the key {X,Y} are matched separately.

The {lookup,true} option can be used to ensure that the qlc module will look up constants in some QLC table. If there are more than one QLC table among the generators' list expressions, constants have to be looked up in at least one of the tables. The evaluation of the query fails if there are no constants to look up. This option is useful in situations when it would be unacceptable to traverse all objects in some table. Setting the lookup option to false ensures that no constants will be looked up ({max_lookup,0} has the same effect). The default value is any which means that constants will be looked up whenever possible.

The {join,Join} option can be used to ensure that a certain join method will be used: {join,lookup} invokes the lookup join method; {join,merge} invokes the merge join method; and {join,nested_loop} invokes the method of matching every pair of objects from two handles. The last method is mostly very slow. The evaluation of the query fails if the qlc module cannot carry out the chosen join method. The default value is any which means that some fast join method will be used if possible.

`sort(QH1 [, SortOptions]) -> QH2`

Types:

- QH1 = QueryHandleOrList
- QH2 = QueryHandle

Returns a query handle. When evaluating the query handle `QH2` the answers to the query handle `QH1` are sorted by file_sorter:sort/3 [page 163] according to the options.

The sorter will use temporary files only if `QH1` does not evaluate to a list and the size of the binary representation of the answers exceeds `Size` bytes, where `Size` is the value of the `size` option.

`string_to_handle(QueryString [, Options [, Bindings]]) -> QueryHandle | Error`

Types:

- QueryString = string()
- Options = [Option] | Option
- Option = {max_lookup, MaxLookup} | {cache, Cache} | cache | {join, Join} | {lookup, Lookup} | {unique, bool()} | unique
- MaxLookup = int() >= 0 | infinity
- Join = any | lookup | merge | nested_loop
- Lookup = bool() | any
- Bindings =-as returned by erl_eval:bindings/1-
- Error = {error, module(), Reason}
- Reason = -ErrorInfo as returned by erl_scan:string/1 or erl_parse:parse_exprs/1-

A string version of `qlc:q/1,2`. When the query handle is evaluated the fun created by the parse transform is interpreted by `erl_eval(3)`. The query string is to be one single query list comprehension terminated by a period.

```
1> L = [1,2,3],
Bs = erl_eval:add_binding('L', L, erl_eval:new_bindings()),
QH = qlc:string_to_handle("[X+1 || X <- L].", [], Bs),
qlc:eval(QH).
[2,3,4]
```

This function is probably useful mostly when called from outside of Erlang, for instance from a driver written in C.

`table(TraverseFun, Options) -> QueryHandle`

Types:

- TraverseFun = TraverseFun0 | TraverseFun1
- TraverseFun0 = fun() -> TraverseResult
- TraverseFun1 = fun(MatchExpression) -> TraverseResult
- TraverseResult = Objects | term()
- Objects = [] | [term() | ObjectList]
- ObjectList = TraverseFun0 | Objects
- Options = [Option] | Option
- Option = {format_fun, FormatFun} | {info_fun, InfoFun} | {lookup_fun, LookupFun} | {parent_fun, ParentFun} | {post_fun, PostFun} | {pre_fun, PreFun}
- FormatFun = undefined | fun(SelectedObjects) -> FormatedTable

- SelectedObjects = all | {all, NElements, DepthFun} | {match_spec, MatchExpression} | {lookup, Position, Keys} | {lookup, Position, Keys, NElements, DepthFun}
- NElements = infinity | int() > 0
- DepthFun = fun(term()) -> term()
- FormatedTable = {Mod, Fun, Args} | AbstractExpression | character_list()
- InfoFun = undefined | fun(InfoTag) -> InfoValue
- InfoTag = indices | is_unique_objects | keypos | num_of_objects
- InfoValue = undefined | term()
- LookupFun = undefined | fun(Position, Keys) -> LookupResult
- LookupResult = [term()] | term()
- ParentFun = undefined | fun() -> ParentFunValue
- PostFun = undefined | fun() -> void()
- PreFun = undefined | fun([PreArg]) -> void()
- PreArg = {parent_value, ParentFunValue} | {stop_fun, StopFun}
- ParentFunValue = undefined | term()
- StopFun = undefined | fun() -> void()
- Position = int() > 0
- Keys = [term()]
- Mod = Fun = atom()
- Args = [term()]

Returns a query handle for a QLC table. In Erlang/OTP there is support for ETS, Dets and Mnesia tables, but it is also possible to turn many other data structures into QLC tables. The way to accomplish this is to let function(s) in the module implementing the data structure create a query handle by calling `qlc:table/2`. The different ways to traverse the table as well as properties of the table are handled by callback functions provided as options to `qlc:table/2`.

The callback function `TraverseFun` is used for traversing the table. It is to return a list of objects terminated by either `[]` or a nullary fun to be used for traversing the not yet traversed objects of the table. Any other return value is immediately returned as value of the query evaluation. Unary `TraverseFuns` are to accept a match specification as argument. The match specification is created by the parse transform by analyzing the pattern of the generator calling `qlc:table/2` and filters using variables introduced in the pattern. If the parse transform cannot find a match specification equivalent to the pattern and filters, `TraverseFun` will be called with a match specification returning every object. Modules that can utilize match specifications for optimized traversal of tables should call `qlc:table/2` with a unary `TraverseFun` while other modules can provide a nullary `TraverseFun`. `ets:table/2` is an example of the former; `gb_table:table/1` in the Implementing a QLC table [page 292] section is an example of the latter.

`PreFun` is a unary callback function that is called once before the table is read for the first time. If the call fails, the query evaluation fails. Similarly, the nullary callback function `PostFun` is called once after the table was last read. The return value, which is caught, is ignored. If `PreFun` has been called for a table, `PostFun` is guaranteed to be called for that table, even if the evaluation of the query fails for some reason. The order in which pre (post) functions for different tables are evaluated is not specified. Other table access than reading, such as calling `InfoFun`, is assumed to be OK at any time. The argument `PreArgs` is a list of tagged values. Currently there are two tags, `parent_value` and `stop_fun`, used by Mnesia for managing transactions. The value of

parent_value is the value returned by ParentFun, or undefined if there is no ParentFun. ParentFun is called once just before the call of PreFun in the context of the process calling eval, fold, or cursor. The value of stop_fun is a nullary fun that deletes the cursor if called from the parent, or undefined if there is no cursor.

The binary callback function LookupFun is used for looking up objects in the table. The first argument Position is the key position or an indexed position and the second argument Keys is a sorted list of unique values. The return value is to be a list of all objects (tuples) such that the element at Position is a member of Keys. Any other return value is immediately returned as value of the query evaluation. LookupFun is called instead of traversing the table if the parse transform at compile time can find out that the filters match and compare the element at Position in such a way that only Keys need to be looked up in order to find all potential answers. The key position is obtained by calling InfoFun(keypos) and the indexed positions by calling InfoFun(indices). If the key position can be used for lookup it is always chosen, otherwise the indexed position requiring the least number of lookups is chosen. If there is a tie between two indexed positions the one occurring first in the list returned by InfoFun is chosen. Positions requiring more than max_lookup [page 299] lookups are ignored.

The unary callback function InfoFun is to return information about the table. undefined should be returned if the value of some tag is unknown:

- indices. Returns a list of indexed positions, a list of positive integers.
- is_unique_objects. Returns true if the objects returned by TraverseFun are unique.
- keypos. Returns the position of the table's key, a positive integer.
- is_sorted_key. Returns true if the objects returned by TraverseFun are sorted on the key.
- num_of_objects. Returns the number of objects in the table, a non-negative integer.

The unary callback function FormatFun is used by qlc:info/1,2 [page 296] for displaying the call that created the table's query handle. The default value, undefined, means that info/1,2 displays a call to '$MOD':'$FUN'/0. It is up to FormatFun to present the selected objects of the table in a suitable way. However, if a character list is chosen for presentation it must be an Erlang expression that can be scanned and parsed (a trailing dot will be added by qlc:info though). FormatFun is called with an argument that describes the selected objects based on optimizations done as a result of analyzing the filters of the QLC where the call to qlc:table/2 occurs. The possible values of the argument are:

- {lookup, Position, Keys, NElements, DepthFun}. LookupFun is used for looking up objects in the table.
- {match_spec, MatchExpression}. No way of finding all possible answers by looking up keys was found, but the filters could be transformed into a match specification. All answers are found by calling TraverseFun(MatchExpression).
- {all, NElements, DepthFun}. No optimization was found. A match specification matching all objects will be used if TraverseFun is unary.

NElements is the value of the info/1,2 option n_elements, and DepthFun is a function that can be used for limiting the size of terms; calling DepthFun(Term) substitutes '...' for parts of Term below the depth specified by the info/1,2 option depth. If calling FormatFun with an argument including NElements and DepthFun fails, FormatFun is called once again with an argument excluding NElements and DepthFun ({lookup,Position,Keys} or all).

See ets(3) [page 160], dets(3) [page 95] and [mnesia(3)] for the various options recognized by table/1,2 in respective module.

See Also

dets(3) [page 81], [Erlang Reference Manual], erl_eval(3) [page 116], [erlang(3)], ets(3) [page 139], [file(3)], [error_logger(3)], file_sorter(3) [page 163], [mnesia(3)], [Programming Examples], shell(3) [page 357]

queue

Erlang Module

This module implements (double ended) FIFO queues in an efficient manner.

All functions fail with reason `badarg` if arguments are of wrong type, for example queue arguments are not queues, indexes are not integers, list arguments are not lists. Improper lists cause internal crashes. An index out of range for a queue also causes a failure with reason `badarg`.

Some functions, where noted, fail with reason `empty` for an empty queue.

All operations has an amortized O(1) running time, except `len/1`, `join/2`, `split/2` and `filter/2` that are O(n). To minimize the size of a queue minimizing the amount of garbage built by queue operations, the queues do not contain explicit length information, and that is why `len/1` is O(n). If better performance for this particular operation is essential, it is easy for the caller to keep track of the length.

Queues are double ended. The mental picture of a queue is a line of people (items) waiting for their turn. The queue front is the end with the item that has waited the longest. The queue rear is the end an item enters when it starts to wait. If instead using the mental picture of a list, the front is called head and the rear is called tail.

Entering at the front and exiting at the rear are reverse operations on the queue.

The module has several sets of interface functions. The “Original API”, the “Extended API” and the “Okasaki API”.

The “Original API” and the “Extended API” both use the mental picture of a waiting line of items. Both also have reverse operations suffixed “_r”.

The “Original API” item removal functions return compound terms with both the removed item and the resulting queue. The “Extended API” contain alternative functions that build less garbage as well as functions for just inspecting the queue ends. Also the “Okasaki API” functions build less garbage.

The “Okasaki API” is inspired by “Purely Functional Data structures” by Chris Okasaki. It regards queues as lists. The API is by many regarded as strange and avoidable. For example many reverse operations have lexically reversed names, some with more readable but perhaps less understandable aliases.

Original API

Exports

`new() -> Q`

Types:

- Q = queue()

Returns an empty queue.

`is_queue(Term) -> true | false`

Types:

- Term = term()

Tests if `Q` is a queue and returns `true` if so and `false` otherwise.

`is_empty(Q) -> true | false`

Types:

- Q = queue()

Tests if `Q` is empty and returns `true` if so and `false` otherwise.

`len(Q) -> N`

Types:

- Q = queue()
- N = integer()

Calculates and returns the length of queue `Q`.

`in(Item, Q1) -> Q2`

Types:

- Item = term()
- Q1 = Q2 = queue()

Inserts `Item` at the rear of queue `Q1`. Returns the resulting queue `Q2`.

`in_r(Item, Q1) -> Q2`

Types:

- Item = term()
- Q1 = Q2 = queue()

Inserts `Item` at the front of queue `Q1`. Returns the resulting queue `Q2`.

`out(Q1) -> Result`

Types:

- Result = {{value, Item}, Q2} | {empty, Q1}
- Q1 = Q2 = queue()

Removes the item at the front of queue `Q1`. Returns the tuple {{`value, Item`}, `Q2`}, where `Item` is the item removed and `Q2` is the resulting queue. If `Q1` is empty, the tuple {`empty, Q1`} is returned.

`out_r(Q1) -> Result`

Types:

- Result = {{value, Item}, Q2} | {empty, Q1}
- Q1 = Q2 = queue()

Removes the item at the rear of the queue `Q1`. Returns the tuple {{`value, Item`}, `Q2`}, where `Item` is the item removed and `Q2` is the new queue. If `Q1` is empty, the tuple {`empty, Q1`} is returned.

`from_list(L) -> queue()`

Types:

- L = list()

Returns a queue containing the items in `L` in the same order; the head item of the list will become the front item of the queue.

`to_list(Q) -> list()`

Types:

- Q = queue()

Returns a list of the items in the queue in the same order; the front item of the queue will become the head of the list.

`reverse(Q1) -> Q2`

Types:

- Q1 = Q2 = queue()

Returns a queue `Q2` that contains the items of `Q1` in the reverse order.

`split(N, Q1) -> {Q2,Q3}`

Types:

- N = integer()
- Q1 = Q2 = Q3 = queue()

Splits `Q1` in two. The `N` front items are put in `Q2` and the rest in `Q3`

`join(Q1, Q2) -> Q3`

Types:

- Q1 = Q2 = Q3 = queue()

Returns a queue `Q3` that is the result of joining `Q1` and `Q2` with `Q1` in front of `Q2`.

`filter(Fun, Q1) -> Q2`

Types:

- Fun = fun(Item) -> bool() | list()

- Q1 = Q2 = queue()

Returns a queue `Q2` that is the result of calling `Fun(Item)` on all items in `Q1`, in order from front to rear.

If `Fun(Item)` returns `true`, `Item` is copied to the result queue. If it returns `false`, `Item` is not copied. If it returns a list the list elements are inserted instead of `Item` in the result queue.

So, `Fun(Item)` returning `[Item]` is thereby semantically equivalent to returning `true`, just as returning `[]` is semantically equivalent to returning `false`. But returning a list builds more garbage than returning an atom.

Extended API

Exports

`get(Q) -> Item`

Types:

- Item = term()
- Q = queue()

Returns `Item` at the front of queue `Q`.

Fails with reason `empty` if `Q` is empty.

`get_r(Q) -> Item`

Types:

- Item = term()
- Q = queue()

Returns `Item` at the rear of queue `Q`.

Fails with reason `empty` if `Q` is empty.

`drop(Q1) -> Q2`

Types:

- Item = term()
- Q1 = Q2 = queue()

Returns a queue `Q2` that is the result of removing the front item from `Q1`.

Fails with reason `empty` if `Q1` is empty.

`drop_r(Q1) -> Q2`

Types:

- Item = term()
- Q1 = Q2 = queue()

Returns a queue `Q2` that is the result of removing the rear item from `Q1`.

Fails with reason `empty` if `Q1` is empty.

`peek(Q) -> {value,Item} | empty`

Types:

- Item = term()
- Q = queue()

Returns the tuple `{value, Item}` where `Item` is the front item of `Q`, or `empty` if `Q1` is empty.

`peek_r(Q) -> {value,Item} | empty`

Types:

- Item = term()
- Q = queue()

Returns the tuple `{value, Item}` where `Item` is the rear item of `Q`, or `empty` if `Q1` is empty.

Okasaki API

Exports

`cons(Item, Q1) -> Q2`

Types:

- Item = term()
- Q1 = Q2 = queue()

Inserts `Item` at the head of queue `Q1`. Returns the new queue `Q2`.

`head(Q) -> Item`

Types:

- Item = term()
- Q = queue()

Returns `Item` from the head of queue `Q`.

Fails with reason `empty` if `Q` is empty.

`tail(Q1) -> Q2`

Types:

- Item = term()
- Q1 = Q2 = queue()

Returns a queue `Q2` that is the result of removing the head item from `Q1`.

Fails with reason `empty` if `Q1` is empty.

`snoc(Q1, Item) -> Q2`

Types:

- Item = term()

- Q1 = Q2 = queue()

Inserts `Item` as the tail item of queue `Q1`. Returns the new queue `Q2`.

```
daeh(Q) -> Item
last(Q) -> Item
```

Types:

- Item = term()
- Q = queue()

Returns the tail item of queue `Q`.

Fails with reason `empty` if `Q` is empty.

```
liat(Q1) -> Q2
init(Q1) -> Q2
lait(Q1) -> Q2
```

Types:

- Item = term()
- Q1 = Q2 = queue()

Returns a queue `Q2` that is the result of removing the tail item from `Q1`.

Fails with reason `empty` if `Q1` is empty.

The name `lait/1` is a misspelling - do not use it anymore.

random

Erlang Module

Random number generator. The method is attributed to B.A. Wichmann and I.D.Hill, in 'An efficient and portable pseudo-random number generator', Journal of Applied Statistics. AS183. 1982. Also Byte March 1987.

The current algorithm is a modification of the version attributed to Richard A O'Keefe in the standard Prolog library.

Every time a random number is requested, a state is used to calculate it, and a new state produced. The state can either be implicit (kept in the process dictionary) or be an explicit argument and return value. In this implementation, the state (the type `ran()`) consists of a tuple of three integers.

Exports

`seed() -> ran()`

Seeds random number generation with default (fixed) values in the process dictionary, and returns the old state.

`seed(A1, A2, A3) -> ran()`

Types:

- A1 = A2 = A3 = int()

Seeds random number generation with integer values in the process dictionary, and returns the old state.

One way of obtaining a seed is to use the BIF `now/0`:

```
...
{A1,A2,A3} = now(),
random:seed(A1, A2, A3),
...
```

`seed0() -> ran()`

Returns the default state.

`uniform()-> float()`

Returns a random float uniformly distributed between `0.0` and `1.0`, updating the state in the process dictionary.

`uniform(N) -> int()`

Types:

- N = int()

Given an integer `N >= 1`, `uniform/1` returns a random integer uniformly distributed between `1` and `N`, updating the state in the process dictionary.

`uniform_s(State0) -> {float(), State1}`

Types:

- State0 = State1 = ran()

Given a state, `uniform_s/1`returns a random float uniformly distributed between `0.0` and `1.0`, and a new state.

`uniform_s(N, State0) -> {int(), State1}`

Types:

- N = int()
- State0 = State1 = ran()

Given an integer `N >= 1` and a state, `uniform_s/2` returns a random integer uniformly distributed between `1` and `N`, and a new state.

Note

Some of the functions use the process dictionary variable `random_seed` to remember the current seed.

If a process calls `uniform/0` or `uniform/1` without setting a seed first, `seed/0` is called automatically.

re

Erlang Module

> **Warning:**
> This is an experimental module and the interface is subject to change. The purpose is to be a reference implementation of what's described in the EEP "Built in regular expressions in Erlang", available at www.erlang.org for review. No production code should be based on this module until the draft is accepted and the module corrected accordingly.

This module contains functions for regular expression matching for strings and binaries.

The regular expression syntax and semantics resemble that of Perl. This library in many ways replaces the old regexp library written purely in Erlang, as it has a richer syntax as well as many more options. The library is also many times faster than the pure Erlang implementation.

Although the library's matching algorithms are currently based on the PCRE library, it is not to be viewed as an Erlang to PCRE mapping. Only parts of the PCRE library is interfaced and the re library in some ways extend PCRE. The PCRE documentation contains many parts of no interest to the Erlang programmer, why only the relevant part of the documentation is included here. There should bee no need to go directly to the PCRE library documentation.

DATA TYPES

```
iodata() = iolist() | binary()
iolist() = [char() | binary() | iolist()]
  - a binary is allowed as the tail of the list
```

```
mp() = Opaque datatype containing a compiled regular expression.
```

Exports

`compile(Regexp) -> {ok, MP} | {error, ErrSpec}`

Types:

- Regexp = iodata()

The same as `compile(Regexp,[])`

`compile(Regexp,Options) -> {ok, MP} | {error, ErrSpec}`

Types:

- Regexp = iodata()
- Options = [Option]
- Option = anchored | caseless | dollar_endonly | dotall | extended | firstline | multiline | no_auto_capture | dupnames | ungreedy | {newline, NLSpec}
- NLSpec = cr | crlf | lf | anycrlf
- MP = mp()
- ErrSpec = {ErrString, Position}
- ErrString = string()
- Position = int()

This function compiles a regular expression with the syntax described below into an internal format to be used later as a parameter to the run/2,3 functions.

Compiling the regular expression before matching is useful if the same expression is to be used in matching against multiple subjects during the program's lifetime. Compiling once and executing many times is far more efficient than compiling each time one wants to match.

The options have the following meanings:

`anchored` The pattern is forced to be "anchored", that is, it is constrained to match only at the first matching point in the string that is being searched (the "subject string"). This effect can also be achieved by appropriate constructs in the pattern itself.

`caseless` Letters in the pattern match both upper and lower case letters. It is equivalent to Perl's /i option, and it can be changed within a pattern by a (?i) option setting. Uppercase and lowercase letters are defined as in the ISO-8859-1 character set.

`dollar_endonly` A dollar metacharacter in the pattern matches only at the end of the subject string. Without this option, a dollar also matches immediately before a newline at the end of the string (but not before any other newlines). The `dollar_endonly` option is ignored if `multiline` is given. There is no equivalent option in Perl, and no way to set it within a pattern.

`dotall` A dot maturate in the pattern matches all characters, including those that indicate newline. Without it, a dot does not match when the current position is at a newline. This option is equivalent to Perl's /s option, and it can be changed within a pattern by a (?s) option setting. A negative class such as [^a] always matches newline characters, independent of the setting of this option.

`extended` Whitespace data characters in the pattern are ignored except when escaped or inside a character class. Whitespace does not include the VT character (ASCII 11). In addition, characters between an unescaped # outside a character class and the next newline, inclusive, are also ignored. This is equivalent to Perl's /x option, and it can be changed within a pattern by a (?x) option setting. This option makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. Whitespace characters may never appear within special character sequences in a pattern, for example within the sequence `(?(` which introduces a conditional subpattern.

`firstline` An unanchored pattern is required to match before or at the first newline in the subject string, though the matched text may continue over the newline.

`multiline` By default, PCRE treats the subject string as consisting of a single line of characters (even if it actually contains newlines). The "start of line" metacharacter (^) matches only at the start of the string, while the "end of line" metacharacter ($) matches only at the end of the string, or before a terminating newline (unless `dollar_endonly` is given). This is the same as Perl.

When `multiline` it is given, the "start of line" and "end of line" constructs match immediately following or immediately before internal newlines in the subject string, respectively, as well as at the very start and end. This is equivalent to Perl's /m option, and it can be changed within a pattern by a (?m) option setting. If there are no newlines in a subject string, or no occurrences of ^ or $ in a pattern, setting `multiline` has no effect.

`no_auto_capture` Disables the use of numbered capturing parentheses in the pattern. Any opening parenthesis that is not followed by ? behaves as if it were followed by ?: but named parentheses can still be used for capturing (and they acquire numbers in the usual way). There is no equivalent of this option in Perl.

`dupnames` Names used to identify capturing subpatterns need not be unique. This can be helpful for certain types of pattern when it is known that only one instance of the named subpattern can ever be matched. There are more details of named subpatterns below

`ungreedy` This option inverts the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by "?". It is not compatible with Perl. It can also be set by a (?U) option setting within the pattern.

`{newline, NLSpec}` Override the default definition of a newline in the subject string, which is LF (ASCII 10) in Erlang.

`cr` Newline is indicated by a single character CR (ASCII 13)

`lf` Newline is indicated by a single character LF (ASCII 10), the default

`crlf` Newline is indicated by the two-character CRLF (ASCII 13 followed by ASCII 10) sequence.

`anycrlf` Any of the three preceding sequences should be recognized.

`run(Subject,RE) -> {match, Captured} | nomatch | {error, ErrSpec}`

Types:

- RE = mp() | iodata()
- Subject = iodata()
- Captured = [CaptureData]
- CaptureData = {int(),int()} | string() | binary()
- ErrSpec = {ErrString, Position}

- ErrString = string()
- Position = int()

The same as `run(RE,Subject,[])`.

`run(Subject,RE,Options) -> {match, Captured} | nomatch | {error, ErrSpec}`

Types:

- RE = mp() | iodata()
- Subject = iodata()
- Options = [Option]
- Option = anchored | notbol | noteol | notempty | {offset, int()} | {newline, NLSpec} | {capture, ValueSpec} | {capture, ValueSpec, Type} | CompileOpt
- Type = index | list | binary
- ValueSpec = all | all_but_first | first | ValueList
- ValueList = [ValueID]
- ValueID = int() | string() | atom()
- CompileOpt = see compile/2 above
- NLSpec = cr | crlf | lf | anycrlf
- Captured = [CaptureData]
- CaptureData = {int(),int()} | string() | binary()
- ErrSpec = {ErrString, Position}
- ErrString = string()
- Position = int()

Executes a regexp matching, returning `{match, Captured}` or `nomatch`. The regular expression can be given either as `iodata()` in which case it is automatically compiled (as by `re:compile/2`) and executed, or as a pre compiled `mp()` in which case it is executed against the subject directly.

When compilation is involved, the function may return compilation errors as when compiling separately (`{error, {string(),int()}}`); when only matching, no errors are returned.

The option list can only contain the options `anchored`, `notbol`, `noteol`, `notempty`, `{offset, int()}`, `{newline, NLSpec}` and `{capture, ValueSpec}`/`{capture, ValueSpec, Type}` if the regular expression is previously compiled, otherwise all options valid for the `re:compile/2` function are allowed as well. Options allowed both for compilation and execution of a match, namely `anchored` and `{newline, NLSpec}`, will affect both the compilation and execution if present together with a non pre-compiled regular expression.

The `{capture, ValueSpec}`/`{capture, ValueSpec, Type}` defines what to return from the function upon successful matching. The `capture` tuple may contain both a value specification telling which of the captured substrings are to be returned, and a type specification, telling how captured substrings are to be returned (as index tuples, lists or binaries). The `capture` option makes the function quite flexible and powerful. The different options are described in detail below

If the capture options describe that no substring capturing at all is to be done (`{capture, none}`), the function will return the single atom `match` upon successful matching, otherwise the tuple `{match, ValueList}` is returned. Disabling capturing can be done either by specifying `none` or an empty list as `ValueSpec`.

A description of all the options relevant for execution follows:

`anchored` Limits `re:run/3` to matching at the first matching position. If a pattern was compiled with `anchored`, or turned out to be anchored by virtue of its contents, it cannot be made unachored at matching time, hence there is no `unanchored` option.

`notempty` An empty string is not considered to be a valid match if this option is given. If there are alternatives in the pattern, they are tried. If all the alternatives match the empty string, the entire match fails. For example, if the pattern

```
a?b?
```

is applied to a string not beginning with “a” or “b”, it matches the empty string at the start of the subject. With `notempty` given, this match is not valid, so re:run/3 searches further into the string for occurrences of “a” or “b”.

Perl has no direct equivalent of `notempty`, but it does make a special case of a pattern match of the empty string within its split() function, and when using the /g modifier. It is possible to emulate Perl’s behavior after matching a null string by first trying the match again at the same offset with `notempty` and `anchored`, and then if that fails by advancing the starting offset (see below) and trying an ordinary match again.

`notbol` This option specifies that the first character of the subject string is not the beginning of a line, so the circumflex metacharacter should not match before it. Setting this without `multiline` (at compile time) causes circumflex never to match. This option affects only the behavior of the circumflex metacharacter. It does not affect \A.

`noteol` This option specifies that the end of the subject string is not the end of a line, so the dollar metacharacter should not match it nor (except in multiline mode) a newline immediately before it. Setting this without `multiline` (at compile time) causes dollar never to match. This option affects only the behavior of the dollar metacharacter. It does not affect \Z or \z.

`{offset, int()}` Start matching at the offset (position) given in the subject string. The offset is zero-based, so that the default is `{offset,0}` (all of the subject string).

`{newline, NLSpec}` Override the default definition of a newline in the subject string, which is LF (ASCII 10) in Erlang.

`cr` Newline is indicated by a single character CR (ASCII 13)

`lf` Newline is indicated by a single character LF (ASCII 10), the default

`crlf` Newline is indicated by the two-character CRLF (ASCII 13 followed by ASCII 10) sequence.

`anycrlf` Any of the three preceding sequences should be recognized.

`{capture, ValueSpec}`/`{capture, ValueSpec, Type}` Specifies which captured substrings are returned and in what format. By default, `re:run/3` captures all of the matching part of the substring as well as all capturing subpatterns (all of the pattern is automatically captured). The default return type is (zero-based) indexes of the captured parts of the string, given as `{Offset,Length}` pairs (the `index` `Type` of capturing).

As an example of the default behavior, the following call:

```
re:run("ABCabcdABC","abcd",[]).
```

returns, as first and only captured string the matching part of the subject (“abcd” in the middle) as a index pair `{3,4}`, where character positions are zero based, just as in offsets. The return value of the call above would then be:

```
{match,[{3,4}]}
```

Another (and quite common) case is where the regular expression matches all of the subject, as in:

```
re:run("ABCabcdABC",".*abcd.*",[]).
```

where the return value correspondingly will point out all of the string, beginning at index 0 and being 10 characters long:

```
{match,[{0,10}]}
```

If the regular expression contains capturing subpatterns, like in the following case:

```
re:run("ABCabcdABC",".*(abcd).*",[]).
```

all of the matched subject is captured, as well as the captured substrings:

```
{match,[{0,10},{3,4}]}
```

the complete matching pattern always giving the first return value in the list and the rest of the subpatterns being added in the order they occurred in the regular expression.

The capture tuple is built up as follows:

`ValueSpec` Specifies which captured (sub)patterns are to be returned. The ValueSpec can either be an atom describing a predefined set of return values, or a list containing either the indexes or the names of specific subpatterns to return.

The predefined sets of subpatterns are:

`all` All captured subpatterns including the complete matching string. This is the default.

`first` Only the first captured subpattern, which is always the complete matching part of the subject. All explicitly captured subpatterns are discarded.

`all_but_first` All but the first matching subpattern, i.e. all explicitly captured subpatterns, but not the complete matching part of the subject string. This is useful if the regular expression as a whole matches a large part of the subject, but the part you're interested in is in an explicitly captured subpattern. If the return type is `list` or `binary`, not returning subpatterns you're not interested in is a good way to optimize.

`none` Do not return matching subpatterns at all, yielding the single atom `match` as the return value of the function when matching successfully instead of the {`match, list()`} return. Specifying an empty list gives the same behavior.

The value list is a list of indexes for the subpatterns to return, where index 0 is for all of the pattern, and 1 is for the first explicit capturing subpattern in the regular expression, and so forth. When using named captured subpatterns (see below) in the regular expression, one can use `atom()`'s or `string()`'s to specify the subpatterns to be returned. This deserves an example, consider the following regular expression:

```
".*(abcd).*"
```

matched against the string “"ABCabcdABC“, capturing only the "abcd“ part (the first explicit subpattern):

```
re:run("ABCabcdABC",".*(abcd).*",[{capture,[1]}]).
```

The call will yield the following result:

```
{match,[{3,4}]}
```

as the first explicitly captured subpattern is “(abcd)”, matching “abcd” in the subject, at (zero-based) position 3, of length 4.
Now consider the same regular expression, but with the subpattern explicitly named ’FOO’:

```
".*(?<FOO>abcd).*"
```

With this expression, we could still give the index of the subpattern with the following call:

```
re:run("ABCabcdABC",".*(?<FOO>abcd).*",[{capture,[1]}]).
```

giving the same result as before. But as the subpattern is named, we can also give its name in the value list:

```
re:run("ABCabcdABC",".*(?<FOO>abcd).*",[{capture,['FOO']}]).
```

which would yield the same result as the earlier examples, namely:

```
{match,[{3,4}]}
```

The values list might specify indexes or names not present in the regular expression, in which case the return values vary depending on the type. If the type is `index`, the tuple `{-1,0}` is returned for values having no corresponding subpattern in the regexp, but for the other types (`binary` and `list`), the values are the empty binary or list respectively. This makes it impossible to differentiate between a empty matching subpattern and an invalid subpattern name in the return values for those types. If that differentiation is necessary, use the `type index` and do the conversion to the final type in Erlang code.

In general, subpatterns that got assigned no value in the match are returned as the tuple `{-1,0}` when `type` is `index`. Unasigned subpatterns are returned as the empty binary or list respectively for other return types. Consider the regular expression:

```
".*((?<FOO>abdd)|a(..d)).*"
```

There are three explicitly capturing subpatterns, where the opening parenthesis position determines the order in the result, hence `((?<FOO>abdd)|a(..d))` is subpattern index 1, `(?<FOO>abdd)` is subpattern index 2 and `(..d)` is subpattern index 3. When matched against the following string:

```
"ABCabcdABC"
```

the subpattern at index 2 won’t match, as “abdd” is not present in the string, but the complete pattern matches (due to the alternative `a(..d)`. The subpattern at index 2 is therefore unassigned and the default return value will be:

```
{match,[{0,10},{3,4},{-1,0},{4,3}]}
```

Setting the capture `Type` to `binary` would give the following:

```
{match,[<<"ABCabcdABC">>,<<"abcd">>,<<>>,<<"bcd">>]}
```

where the empty binary (`<<>>`) represents the unassigned subpattern. In the `binary` case, some information about the matching is therefore lost, the `<<>>` might just as well be an empty string captured.

`Type` Optionally specifies how captured substrings are to be returned. If omitted, the default of `index` is used. The `Type` can be one of the following:

`index` Return captured substrings as pairs of byte indexes into the subject string and length of the matching string in the subject (as if the subject string was flattened with iolist_to_binary prior to matching). This is the default.

`list` Return matching substrings as lists of characters (Erlang `string()`'s).

`binary` Return matching substrings as binaries.

The options solely affecting the compilation step are described in the `re:compile/2` function.

PERL LIKE REGULAR EXPRESSIONS SYNTAX

The following sections contain reference material for the regular expressions used by this module. The regular expression reference is taken from the PCRE documentation, but converted as needed.

The documentation is altered where appropriate and where the re module behaves differently than the PCRE library.

PCRE regular expression details

The syntax and semantics of the regular expressions that are supported by PCRE are described in detail below. Perl's regular expressions are described in its own documentation, and regular expressions in general are covered in a number of books, some of which have copious examples. Jeffrey Friedl's "Mastering Regular Expressions", published by O'Reilly, covers regular expressions in great detail. This description of PCRE's regular expressions is intended as reference material.

The reference material is divided into the following sections:

Newline conventions

PCRE supports four different conventions for indicating line breaks in strings: a single CR (carriage return) character, a single LF (linefeed) character, the two-character sequence CRLF or any of the three preceding.

It is also possible to specify a newline convention by starting a pattern string with one of the following five sequences:

(*CR) carriage return

(*LF) linefeed

(*CRLF) carriage return, followed by linefeed

(*ANYCRLF) any of the three above

These override the default and the options given to `re:compile/2`. For example, the pattern:

```
(*CR)a.b
```

changes the convention to CR. That pattern matches “a\nb” because LF is no longer a newline. Note that these special settings, which are not Perl-compatible, are recognized only at the very start of a pattern, and that they must be in upper case. If more than one of them is present, the last one is used.

Characters and metacharacters

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

```
The quick brown fox
```

matches a portion of a subject string that is identical to itself. When caseless matching is specified (the `caseless` option), letters are matched independently of case.

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of *metacharacters*, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of metacharacters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized within square brackets. Outside square brackets, the metacharacters are as follows:

\ general escape character with several uses

^ assert start of string (or line, in multiline mode)

$ assert end of string (or line, in multiline mode)

. match any character except newline (by default)

[start character class definition

| start of alternative branch

(start subpattern

) end subpattern

? extends the meaning of (, also 0 or 1 quantifier, also quantifier minimizer

* 0 or more quantifier

+ 1 or more quantifier, also “possessive quantifier”

{ start min/max quantifier

Part of a pattern that is in square brackets is called a “character class”. In a character class the only metacharacters are:

\ general escape character

^ negate the class, but only if the first character

- indicates character range

[POSIX character class (only if followed by POSIX syntax)

] terminates the character class

The following sections describe the use of each of the metacharacters.

Backslash

The backslash character has several uses. Firstly, if it is followed by a non-alphanumeric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a * character, you write * in the pattern. This escaping action applies whether or not the following character would otherwise be interpreted as a metacharacter, so it is always safe to precede a non-alphanumeric with backslash to specify that it stands for itself. In particular, if you want to match a backslash, you write \\.

If a pattern is compiled with the `extended` option, whitespace in the pattern (other than in a character class) and characters between a # outside a character class and the next newline are ignored. An escaping backslash can be used to include a whitespace or # character as part of the pattern.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between \Q and \E. This is different from Perl in that $ and @ are handled as literals in \Q...\E sequences in PCRE, whereas in Perl, $ and @ cause variable interpolation. Note the following examples:

```
Pattern           PCRE matches   Perl matches

\Qabc$xyz\E       abc$xyz        abc followed by the contents of $xyz
\Qabc\$xyz\E      abc\$xyz       abc\$xyz
\Qabc\E\$\Qxyz\E  abc$xyz        abc$xyz
```

The \Q...\E sequence is recognized both inside and outside character classes.

Non-printing characters

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

\a alarm, that is, the BEL character (hex 07)

\cx "control-x", where x is any character

\e escape (hex 1B)

\f formfeed (hex 0C)

\n linefeed (hex 0A)

\r carriage return (hex 0D)

\t tab (hex 09)

\ddd character with octal code ddd, or backreference

\xhh character with hex code hh

\x{hhh..} character with hex code hhh..

The precise effect of \cx is as follows: if x is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus \cz becomes hex 1A, but \c{ becomes hex 3B, while \c; becomes hex 7B.

After \x, from zero to two hexadecimal digits are read (letters can be in upper or lower case). Any number of hexadecimal digits may appear between \x{ and }, but the value of the character code must be less than 256.

If characters other than hexadecimal digits appear between \x{ and }, or if there is no terminating }, this form of escape is not recognized. Instead, the initial \x will be interpreted as a basic hexadecimal escape, with no following digits, giving a character whose value is zero.

Characters whose value is less than 256 can be defined by either of the two syntaxes for \x. There is no difference in the way they are handled. For example, \xdc is exactly the same as \x{dc}.

After \0 up to two further octal digits are read. If there are fewer than two digits, just those that are present are used. Thus the sequence \0\x\07 specifies two binary zeros followed by a BEL character (code value 7). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a *back reference*. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and uses them to generate a data character. Any subsequent digits stand for themselves. The value of a character specified in octal must be less than \400. For example:

\040 is another way of writing a space

\40 is the same, provided there are fewer than 40 previous capturing subpatterns

\7 is always a back reference

\11 might be a back reference, or another way of writing a tab

\011 is always a tab

\0113 is a tab followed by the character “3”

\113 might be a back reference, otherwise the character with octal code 113

\377 might be a back reference, otherwise the byte consisting entirely of 1 bits

\81 is either a back reference, or a binary zero followed by the two characters “8” and “1”

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single character value can be used both inside and outside character classes. In addition, inside a character class, the sequence \b is interpreted as the backspace character (hex 08), and the sequences \R and \X are interpreted as the characters “R” and “X”, respectively. Outside a character class, these sequences have different meanings (see below).

Absolute and relative back references

The sequence \g followed by an unsigned or a negative number, optionally enclosed in braces, is an absolute or relative back reference. A named back reference can be coded as \g{name}. Back references are discussed later, following the discussion of parenthesized subpatterns.

Generic character types

Another use of backslash is for specifying generic character types. The following are always recognized:

\d any decimal digit

\D any character that is not a decimal digit

\h any horizontal whitespace character

\H any character that is not a horizontal whitespace character

\s any whitespace character

\S any character that is not a whitespace character

\v any vertical whitespace character

\V any character that is not a vertical whitespace character

\w any “word” character

\W any “non-word” character

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

For compatibility with Perl, \s does not match the VT character (code 11). This makes it different from the the POSIX “space” class. The \s characters are HT (9), LF (10), FF

(12), CR (13), and space (32). If “use locale;” is included in a Perl script, \s may match the VT character. In PCRE, it never does.

A “word” character is an underscore or any character less than 256 that is a letter or digit. The definition of letters and digits is controlled by PCRE’s low-valued character tables, which are always ISO-8859-1.

Newline sequences

The \R sequence is equivalent to the following:

```
(?>\r\n|\n|\x0b|\f|\r|\x85)
```

This is an example of an “atomic group”, details of which are given below.

This particular group matches either the two-character sequence CR followed by LF, or one of the single characters LF (linefeed, U+000A), VT (vertical tab, U+000B), FF (formfeed, U+000C), CR (carriage return, U+000D), or NEL (next line, U+0085). The two-character sequence is treated as a single unit that cannot be split.

Inside a character class, \R matches the letter “R”.

Resetting the match start

The escape sequence \K, which is a Perl 5.10 feature, causes any previously matched characters not to be included in the final matched sequence. For example, the pattern:

```
foo\Kbar
```

matches “foobar”, but reports that it has matched “bar”. This feature is similar to a lookbehind assertion (described below). However, in this case, the part of the subject before the real match does not have to be of fixed length, as lookbehind assertions do. The use of \K does not interfere with the setting of captured substrings. For example, when the pattern

```
(foo)\Kbar
```

matches “foobar”, the first substring is still set to “foo”.

Simple assertions

The final use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described below. The backslashed assertions are:

\b matches at a word boundary

\B matches when not at a word boundary

\A matches at the start of the subject

\Z matches at the end of the subject also matches before a newline at the end of the subject

\z matches only at the end of the subject

\G matches at the first matching position in the subject

These assertions may not appear in character classes (but note that \b has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match \w or \W (i.e. one matches \w and the other matches \W), or the start or end of the string if the first or last character matches \w, respectively.

The \A, \Z, and \z assertions differ from the traditional circumflex and dollar (described in the next section) in that they only ever match at the very start and end of the subject string, whatever options are set. Thus, they are independent of multiline mode. These three assertions are not affected by the `notbol` or `noteol` options, which affect only the behaviour of the circumflex and dollar metacharacters. However, if the *startoffset* argument of `re:run/3` is non-zero, indicating that matching is to start at a point other than the beginning of the subject, \A can never match. The difference between \Z and \z is that \Z matches before a newline at the end of the string as well as at the very end, whereas \z matches only at the end.

The \G assertion is true only when the current matching position is at the start point of the match, as specified by the *startoffset* argument of `re:run/3`. It differs from \A when the value of *startoffset* is non-zero. By calling `re:run/3` multiple times with appropriate arguments, you can mimic Perl's /g option, and it is in this kind of implementation where \G can be useful.

Note, however, that PCRE's interpretation of \G, as the start of the current match, is subtly different from Perl's, which defines it as the end of the previous match. In Perl, these can be different when the previously matched string was empty. Because PCRE does just one match at a time, it cannot reproduce this behaviour.

If all the alternatives of a pattern begin with \G, the expression is anchored to the starting match position, and the "anchored" flag is set in the compiled regular expression.

Circumflex and dollar

Outside a character class, in the default matching mode, the circumflex character is an assertion that is true only if the current matching point is at the start of the subject string. If the *startoffset* argument of `re:run/3` is non-zero, circumflex can never match if the `multiline` option is unset. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character is an assertion that is true only if the current matching point is at the end of the subject string, or immediately before a newline at the end of the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meaning of dollar can be changed so that it matches only at the very end of the string, by setting the `dollar_endonly` option at compile time. This does not affect the \Z assertion.

The meanings of the circumflex and dollar characters are changed if the `multiline` option is set. When this is the case, a circumflex matches immediately after internal newlines as well as at the start of the subject string. It does not match after a newline that ends the string. A dollar matches before any newlines in the string, as well as at the very end, when `multiline` is set. When newline is specified as the two-character sequence CRLF, isolated CR and LF characters do not indicate newlines.

For example, the pattern /^abc$/ matches the subject string “def\nabc” (where \n represents a newline) in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with `^` are not anchored in multiline mode, and a match for circumflex is possible when the *startoffset* argument of `re:run/3` is non-zero. The `dollar_endonly` option is ignored if `multiline` is set.

Note that the sequences \A, \Z, and \z can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with \A it is always anchored, whether or not `multiline` is set.

Full stop (period, dot)

Outside a character class, a dot in the pattern matches any one character in the subject string except (by default) a character that signifies the end of a line.

When a line ending is defined as a single character, dot never matches that character; when the two-character sequence CRLF is used, dot does not match CR if it is immediately followed by LF, but otherwise it matches all characters (including isolated CRs and LFs).

The behaviour of dot with regard to newlines can be changed. If the `dotall` option is set, a dot matches any one character, without exception. If the two-character sequence CRLF is present in the subject string, it takes two dots to match it.

The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newlines. Dot has no special meaning in a character class.

Square brackets and character classes

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class [aeiou] matches any lower case vowel, while [^aeiou] matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters that are in the class by enumerating those that are not. A class that starts with a circumflex is not an assertion: it still consumes a character from the subject string, and therefore it fails if the current pointer is at the end of the string.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless [aeiou] matches “A” as well as “a”, and a caseless [^aeiou] does not match “A”, whereas a caseful version would. Caselessness is always in regard to the ISO-8859-1 character set in Erlang.

Characters that might indicate line breaks are never treated in any special way when matching character classes, whatever line-ending sequence is in use, and whatever setting of the `dotall` and `multiline` options is used. A class such as [^a] always matches one of these characters.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, [d-m] matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character “]” as the end character of a range. A pattern such as [W-]46] is interpreted as a class of two characters (“W” and “-”) followed by a literal string “46]”, so it would match “W46]” or “-46]”. However, if the “]” is escaped with a backslash it is interpreted as the end of range, so [W-\]46] is interpreted as a class containing a range followed by two other characters. The octal or hexadecimal representation of “]” can also be used to end a range.

Ranges operate in the collating sequence of character values. They can also be used for characters specified numerically, for example [\000-\037].

If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, [W-c] is equivalent to [][\\^_‘wxyzabc], matched caselessly .

The character types \d, \D, \p, \P, \s, \S, \w, and \W may also appear in a character class, and add the characters that they match to the class. For example, [\dABCDEF] matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class [^\W_] matches any letter or digit, but not underscore.

The only metacharacters that are recognized in character classes are backslash, hyphen (only where it can be interpreted as specifying a range), circumflex (only at the start), opening square bracket (only when it can be interpreted as introducing a POSIX class name - see the next section), and the terminating closing square bracket. However, escaping other non-alphanumeric characters does no harm.

Posix character classes

Perl supports the POSIX notation for character classes. This uses names enclosed by [: and :] within the enclosing square brackets. PCRE also supports this notation. For example,

```
[01[:alpha:]%]
```

matches “0”, “1”, any alphabetic character, or “%”. The supported class names are

alnum letters and digits

alpha letters

ascii character codes 0 - 127

blank space or tab only

cntrl control characters

digit decimal digits (same as \d)

graph printing characters, excluding space

lower lower case letters

print printing characters, including space

punct printing characters, excluding letters and digits

space whitespace (not quite the same as \s)

upper upper case letters

word "word" characters (same as \w)

xdigit hexadecimal digits

The "space" characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). Notice that this list includes the VT character (code 11). This makes "space" different to \s, which does not include VT (for Perl compatibility).

The name "word" is a Perl extension, and "blank" is a GNU extension from Perl 5.8. Another Perl extension is negation, which is indicated by a ^ character after the colon. For example,

```
[12[:^digit:]]
```

matches "1", "2", or any non-digit. PCRE (and Perl) also recognize the POSIX syntax [.ch.] and [=ch=] where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

Vertical bar

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|sullivan
```

matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined below), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

Internal option setting

The settings of the `caseless`, `multiline`, `dotall`, and `extended` options (which are Perl-compatible) can be changed from within the pattern by a sequence of Perl option letters enclosed between "(?" and ")". The option letters are

i for `caseless`

m for `multiline`

s for `dotall`

x for `extended`

For example, (?im) sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as (?im-sx), which sets `caseless` and `multiline` while unsetting `dotall` and `extended`, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

The PCRE-specific options `dupnames`, `ungreedy`, and `extra` can be changed in the same way as the Perl-compatible options by using the characters J, U and X respectively.

When an option change occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. If the change is placed right at the start of a pattern, PCRE extracts it into the global options

An option change within a subpattern (see below for a description of subpatterns) affects only that part of the current pattern that follows it, so

```
(a(?i)b)c
```

matches abc and aBc and no other strings (assuming `caseless` is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

```
(a(?i)b|c)
```

matches "ab", "aB", "c", and "C", even though when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

Note: There are other PCRE-specific options that can be set by the application when the compile or match functions are called. In some cases the pattern can contain special leading sequences to override what the application has set or what has been defaulted. Details are given in the section entitled "Newline sequences" above.

Subpatterns

Subpatterns are delimited by parentheses (round brackets), which can be nested. Turning part of a pattern into a subpattern does two things:

1. It localizes a set of alternatives. For example, the pattern

    cat(aract|erpillar|)

matches one of the words “cat”, “cataract”, or “caterpillar”. Without the parentheses, it would match “cataract”, “erpillar” or an empty string.

2. It sets up the subpattern as a capturing subpattern. This means that, when the complete pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller via the return value of `re:run/3`. Opening parentheses are counted from left to right (starting from 1) to obtain numbers for the capturing subpatterns.

For example, if the string “the red king” is matched against the pattern

    the ((red|white) (king|queen))

the captured substrings are “red king”, “red”, and “king”, and are numbered 1, 2, and 3, respectively.

The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by a question mark and a colon, the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string “the white queen” is matched against the pattern

    the ((?:red|white) (king|queen))

the captured substrings are “white queen” and “queen”, and are numbered 1 and 2. The maximum number of capturing subpatterns is 65535.

As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the “?” and the “:”. Thus the two patterns

- (?i:saturday|sunday)
- (?:(?i)saturday|sunday)

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match “SUNDAY” as well as “Saturday”.

Duplicate subpattern numbers

Perl 5.10 introduced a feature whereby each alternative in a subpattern uses the same numbers for its capturing parentheses. Such a subpattern starts with (?| and is itself a non-capturing subpattern. For example, consider this pattern:

```
(?|(Sat)ur|(Sun))day
```

Because the two alternatives are inside a (?| group, both sets of capturing parentheses are numbered one. Thus, when the pattern matches, you can look at captured substring number one, whichever alternative matched. This construct is useful when you want to capture part, but not all, of one of a number of alternatives. Inside a (?| group, parentheses are numbered as usual, but the number is reset at the start of each branch. The numbers of any capturing buffers that follow the subpattern start after the highest number used in any branch. The following example is taken from the Perl documentation. The numbers underneath show in which buffer the captured content will be stored.

```
# before  ---------------branch-reset----------- after
/ ( a )  (?| x ( y ) z | (p (q) r) | (t) u (v) ) ( z ) /x
# 1            2         2  3        2     3     4
```

A backreference or a recursive call to a numbered subpattern always refers to the first one in the pattern with the given number.

An alternative approach to using this “branch reset” feature is to use duplicate named subpatterns, as described in the next section.

Named subpatterns

Identifying capturing parentheses by number is simple, but it can be very hard to keep track of the numbers in complicated regular expressions. Furthermore, if an expression is modified, the numbers may change. To help with this difficulty, PCRE supports the naming of subpatterns. This feature was not added to Perl until release 5.10. Python had the feature earlier, and PCRE introduced it at release 4.0, using the Python syntax. PCRE now supports both the Perl and the Python syntax.

In PCRE, a subpattern can be named in one of three ways: (?<name>...) or (?’name’...) as in Perl, or (?P<name>...) as in Python. References to capturing parentheses from other parts of the pattern, such as backreferences, recursion, and conditions, can be made by name as well as by number.

Names consist of up to 32 alphanumeric characters and underscores. Named capturing parentheses are still allocated numbers as well as names, exactly as if the names were not present. The `capture` specification to `re:run/3` can use named values if they are present in the regular expression.

By default, a name must be unique within a pattern, but it is possible to relax this constraint by setting the `dupnames` option at compile time. This can be useful for patterns where only one instance of the named parentheses can match. Suppose you want to match the name of a weekday, either as a 3-letter abbreviation or as the full name, and in both cases you want to extract the abbreviation. This pattern (ignoring the line breaks) does the job:

```
(?<DN>Mon|Fri|Sun)(?:day)?|
(?<DN>Tue)(?:sday)?|
(?<DN>Wed)(?:nesday)?|
(?<DN>Thu)(?:rsday)?|
(?<DN>Sat)(?:urday)?
```

There are five capturing substrings, but only one is ever set after a match. (An alternative way of solving this problem is to use a “branch reset” subpattern, as described in the previous section.)

In case of capturing named subpatterns which are not unique, the first occurence is returned from `re:exec/3`, if the name is specified int the `values` part of the `capture` statement.

Repetition

Repetition is specified by quantifiers, which can follow any of the following items:

- a literal data character
- the dot metacharacter
- the \C escape sequence
- the \X escape sequence
- the \R escape sequence
- an escape such as \d that matches a single character
- a character class
- a back reference (see next section)
- a parenthesized subpattern (unless it is an assertion)

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

z{2,4}

matches “zz”, “zzz”, or “zzzz”. A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

[aeiou]{3,}

matches at least 3 successive vowels, but may match many more, while

\d{8}

matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, {,6} is not a quantifier, but a literal string of four characters.

The quantifier {0} is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience, the three most common quantifiers have single-character abbreviations:

* is equivalent to {0,}

+ is equivalent to {1,}

? is equivalent to {0,1}

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are “greedy”, that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between /* and */ and within the comment, individual * and / characters may appear. An attempt to match C comments by applying the pattern

```
/\*.*\*/
```

to the string

```
/* first comment */ not comment /* second comment */
```

fails, because it matches the entire string owing to the greediness of the .* item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\*.*?\*/
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the `ungreedy` option is set (an option that is not available in Perl), the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more memory is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with .* or .{0,} and the `dotall` option (equivalent to Perl's /s) is set, thus allowing the dot to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE normally treats such a pattern as though it were preceded by \A.

In cases where it is known that the subject string contains no newlines, it is worth setting `dotall` in order to obtain this optimization, or alternatively using ^ to indicate anchoring explicitly.

However, there is one situation where the optimization cannot be used. When .* is inside capturing parentheses that are the subject of a backreference elsewhere in the pattern, a match at the start may fail where a later one succeeds. Consider, for example:

```
(.*)abc\1
```

If the subject is "xyz123abc123" the match point is the fourth character. For this reason, such a pattern is not implicitly anchored.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
(tweedle[dume]{3}\s*)+
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
/(a|(b))+/
```

matches "aba" the value of the second captured substring is "b".

Atomic grouping and possessive quantifiers

With both maximizing ("greedy") and minimizing ("ungreedy" or "lazy") repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern \d+foo when applied to the subject line

```
123456bar
```

After matching all 6 digits and then failing to match “foo”, the normal action of the matcher is to try again with only 5 digits matching the \d+ item, and then with 4, and so on, before ultimately failing. “Atomic grouping” (a term taken from Jeffrey Friedl’s book) provides the means for specifying that once a subpattern has matched, it is not to be re-evaluated in this way.

If we use atomic grouping for the previous example, the matcher gives up immediately on failing to match “foo” the first time. The notation is a kind of special parenthesis, starting with (?> as in this example:

```
(?>\d+)foo
```

This kind of parenthesis “locks up” the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both \d+ and \d+? are prepared to adjust the number of digits they match in order to make the rest of the pattern match, (?>\d+) can only match an entire sequence of digits.

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and can be nested. However, when the subpattern for an atomic group is just a single repeated item, as in the example above, a simpler notation, called a “possessive quantifier” can be used. This consists of an additional + character following a quantifier. Using this notation, the previous example can be rewritten as

```
\d++foo
```

Note that a possessive quantifier can be used with an entire group, for example:

```
(abc|xyz){2,3}+
```

Possessive quantifiers are always greedy; the setting of the `ungreedy` option is ignored. They are a convenient notation for the simpler forms of atomic group. However, there is no difference in the meaning of a possessive quantifier and the equivalent atomic group, though there may be a performance difference; possessive quantifiers should be slightly faster.

The possessive quantifier syntax is an extension to the Perl 5.8 syntax. Jeffrey Friedl originated the idea (and the name) in the first edition of his book. Mike McCloskey liked it, so implemented it when he built Sun’s Java package, and PCRE copied it from there. It ultimately found its way into Perl at release 5.10.

PCRE has an optimization that automatically “possessifies” certain simple pattern constructs. For example, the sequence A+B is treated as A++B because there is no point in backtracking into a sequence of A’s when B must follow.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of an atomic group is the only way to avoid some failing matches taking a very long time indeed. The pattern

```
(\D+|<\d+>)*[!?]
```

matches an unlimited number of substrings that either consist of non-digits, or digits enclosed in <>, followed by either ! or ?. When it matches, it runs quickly. However, if it is applied to

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the internal \D+ repeat and the external * repeat in a large number of ways, and all have to be tried. (The example uses [!?] rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed so that it uses an atomic group, like this:

```
((?>\D+)|<\d+>)*[!?]
```

sequences of non-digits cannot be broken, and failure happens quickly.

Back references

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. A “forward back reference” of this type can make sense when a repetition is involved and the subpattern to the right has participated in an earlier iteration.

It is not possible to have a numerical “forward back reference” to a subpattern whose number is 10 or more using this syntax because a sequence such as \50 is interpreted as a character defined in octal. See the subsection entitled “Non-printing characters” above for further details of the handling of digits following a backslash. There is no such problem when named parentheses are used. A back reference to any subpattern is possible using named parentheses (see below).

Another way of avoiding the ambiguity inherent in the use of digits following a backslash is to use the \g escape sequence, which is a feature introduced in Perl 5.10. This escape must be followed by an unsigned number or a negative number, optionally enclosed in braces. These examples are all identical:

- (ring), \1
- (ring), \g1
- (ring), \g{1}

An unsigned number specifies an absolute reference without the ambiguity that is present in the older syntax. It is also useful when literal digits follow the reference. A negative number is a relative reference. Consider this example:

```
(abc(def)ghi)\g{-1}
```

The sequence \g{-1} is a reference to the most recently started capturing subpattern before \g, that is, is it equivalent to \2. Similarly, \g{-2} would be equivalent to \1. The use of relative references can be helpful in long patterns, and also in patterns that are created by joining together fragments that contain references within themselves.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (see "Subpatterns as subroutines" below for a way of doing that). So the pattern

```
(sens|respons)e and \1ibility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If caseful matching is in force at the time of the back reference, the case of letters is relevant. For example,

```
((?i)rah)\s+\1
```

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

There are several different ways of writing back references to named subpatterns. The .NET syntax \k{name} and the Perl syntax \k<name> or \k'name' are supported, as is the Python syntax (?P=name). Perl 5.10's unified back reference syntax, in which \g can be used for both numeric and named references, is also supported. We could rewrite the above example in any of the following ways:

- (?<p1>(?i)rah)\s+\k<p1>
- (?'p1'(?i)rah)\s+\k{p1}
- (?P<p1>(?i)rah)\s+(?P=p1)
- (?<p1>(?i)rah)\s+\g{p1}

A subpattern that is referenced by name may appear in the pattern before or after the reference.

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern

```
(a|(bc))\2
```

always fails if it starts to match "a" rather than "bc". Because there may be many capturing parentheses in a pattern, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the `extended` option is set, this can be whitespace. Otherwise an empty comment (see "Comments" below) can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern

```
(a|b\1)+
```

matches any number of "a"s and also "aba", "ababbaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

Assertions

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as \b, \B, \A, \G, \Z, \z, ^ and $ are described above.

More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it. An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed.

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Lookahead assertions

Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example,

```
\w+(?=;)
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and

```
foo(?!bar)
```

matches any occurrence of “foo” that is not followed by “bar”. Note that the apparently similar pattern

```
(?!foo)bar
```

does not find an occurrence of “bar” that is preceded by something other than “foo”; it finds any occurrence of “bar” whatsoever, because the assertion (?!foo) is always true when the next three characters are “bar”. A lookbehind assertion is needed to achieve the other effect.

If you want to force a matching failure at some point in a pattern, the most convenient way to do it is with (?!) because an empty string always matches, so an assertion that requires there not to be an empty string must always fail.

Lookbehind assertions

Lookbehind assertions start with (?<= for positive assertions and (?<! for negative assertions. For example,

```
(?<!foo)bar
```

does find an occurrence of “bar” that is not preceded by “foo”. The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several top-level alternatives, they do not all have to have the same fixed length. Thus

```
(?<=bullock|donkey)
```

is permitted, but

```
(?<!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl (at least for 5.8), which requires all branches to match the same length of string. An assertion such as

```
(?<=ab(c|de))
```

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches:

```
(?<=abc|abde)
```

In some cases, the Perl 5.10 escape sequence \K (see above) can be used instead of a lookbehind assertion; this is not restricted to a fixed-length.

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed length and then try to match. If there are insufficient characters before the current position, the assertion fails.

Possessive quantifiers can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

```
abcd$
```

when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for each “a” in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```
^.*abcd$
```

the initial .* matches the entire string at first, but when this fails (because there is no following “a”), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for “a” covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```
^.*+(?<=abcd)
```

there can be no backtracking for the .*+ item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

Using multiple assertions

Several assertions (of any sort) may occur in succession. For example,

```
(?<=\d{3})(?<!999)foo
```

matches “foo” preceded by three digits that are not “999”. Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not “999”. This pattern does *not* match “foo” preceded by six characters, the first of which are digits and the last three of which are not “999”. For example, it doesn’t match “123abcfoo”. A pattern to do that is

```
(?<=\d{3}...)(?<!999)foo
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not “999”.

Assertions can be nested in any combination. For example,

```
(?<=(?<!foo)bar)baz
```

matches an occurrence of “baz” that is preceded by “bar” which in turn is not preceded by “foo”, while

```
(?<=\d{3}(?!999)...)foo
```

is another pattern that matches “foo” preceded by three digits and any three characters that are not “999”.

Conditional subpatterns

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are

- (?(condition)yes-pattern)
- (?(condition)yes-pattern|no-pattern)

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are four kinds of condition: references to subpatterns, references to recursion, a pseudo-condition called DEFINE, and assertions.

Checking for a used subpattern by number

If the text between the parentheses consists of a sequence of digits, the condition is true if the capturing subpattern of that number has previously matched. An alternative notation is to precede the digits with a plus or minus sign. In this case, the subpattern number is relative rather than absolute. The most recently opened parentheses can be referenced by (?(-1), the next most recent by (?(-2), and so on. In looping constructs it can also make sense to refer to subsequent groups with constructs such as (?(+2).

Consider the following pattern, which contains non-significant whitespace to make it more readable (assume the `extended` option) and to divide it into three parts for ease of discussion:

```
( \( )? [^()]+ (?(1) \) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If you were embedding this pattern in a larger one, you could use a relative reference:

```
...other stuff... ( \( )? [^()]+ (?(-1) \) ) ...
```

This makes the fragment independent of the parentheses in the larger pattern.

Checking for a used subpattern by name

Perl uses the syntax (?(<name>)...) or (?('name')...) to test for a used subpattern by name. For compatibility with earlier versions of PCRE, which had this facility before Perl, the syntax (?(name)...) is also recognized. However, there is a possible ambiguity with this syntax, because subpattern names may consist entirely of digits. PCRE looks first for a named subpattern; if it cannot find one and the name consists entirely of digits, PCRE looks for a subpattern of that number, which must be greater than zero. Using subpattern names that consist entirely of digits is not recommended.

Rewriting the above example to use a named subpattern gives this:

```
(?<OPEN> \( )? [^()]+ (?(<OPEN>) \) )
```

Checking for pattern recursion

If the condition is the string (R), and there is no subpattern with the name R, the condition is true if a recursive call to the whole pattern or any subpattern has been made. If digits or a name preceded by ampersand follow the letter R, for example:

```
(?(R3)...) or (?(R&name)...)
```

the condition is true if the most recent recursion is into the subpattern whose number or name is given. This condition does not check the entire recursion stack.

At “top level”, all these recursion test conditions are false. Recursive patterns are described below.

Defining subpatterns for use by reference only

If the condition is the string (DEFINE), and there is no subpattern with the name DEFINE, the condition is always false. In this case, there may be only one alternative in the subpattern. It is always skipped if control reaches this point in the pattern; the idea of DEFINE is that it can be used to define “subroutines” that can be referenced from elsewhere. (The use of “subroutines” is described below.) For example, a pattern to match an IPv4 address could be written like this (ignore whitespace and line breaks):

```
(?(DEFINE) (?<byte> 2[0-4]\d | 25[0-5] | 1\d\d | [1-9]?\d) ) \b (?&byte)
(\.(?&byte)){3} \b
```

The first part of the pattern is a DEFINE group inside which a another group named "byte" is defined. This matches an individual component of an IPv4 address (a number less than 256). When matching takes place, this part of the pattern is skipped because DEFINE acts like a false condition.

The rest of the pattern uses references to the named group to match the four dot-separated components of an IPv4 address, insisting on a word boundary at each end.

Assertion conditions

If the condition is not in any of the above formats, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant whitespace, and with the two alternatives on the second line:

```
(?(?=[^a-z]*[a-z])
\d{2}-[a-z]{3}-\d{2}  |  \d{2}-\d{2}-\d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

Comments

The sequence (?# marks the start of a comment that continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the `extended` option is set, an unescaped # character outside a character class introduces a comment that continues to immediately after the next newline in the pattern.

Recursive patterns

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth.

For some time, Perl has provided a facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at run time, and the code can refer to the expression itself. A Perl pattern using code interpolation to solve the parentheses problem can be created like this:

```
$re = qr{\( (?: (?>[^()]+) | (?p{$re}) )* \)}x;
```

The (?p{...}) item interpolates Perl code at run time, and in this case refers recursively to the pattern in which it appears.

Obviously, PCRE cannot support the interpolation of Perl code. Instead, it supports special syntax for recursion of the entire pattern, and also for individual subpattern recursion. After its introduction in PCRE and Python, this kind of recursion was introduced into Perl at release 5.10.

A special item that consists of (? followed by a number greater than zero and a closing parenthesis is a recursive call of the subpattern of the given number, provided that it occurs inside that subpattern. (If not, it is a “subroutine” call, which is described in the next section.) The special item (?R) or (?0) is a recursive call of the entire regular expression.

In PCRE (like Python, but unlike Perl), a recursive subpattern call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure.

This PCRE pattern solves the nested parentheses problem (assume the `extended` option is set so that whitespace is ignored):

```
\( ( (?>[^()]+) | (?R) )* \)
```

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of non-parentheses, or a recursive match of the pattern itself (that is, a correctly parenthesized substring). Finally there is a closing parenthesis.

If this were part of a larger pattern, you would not want to recurse the entire pattern, so instead you could use this:

```
( \( ( (?>[^()]+) | (?1) )* \) )
```

We have put the pattern into parentheses, and caused the recursion to refer to them instead of the whole pattern.

In a larger pattern, keeping track of parenthesis numbers can be tricky. This is made easier by the use of relative references. (A Perl 5.10 feature.) Instead of (?1) in the pattern above you can write (?-2) to refer to the second most recently opened parentheses preceding the recursion. In other words, a negative number counts capturing parentheses leftwards from the point at which it is encountered.

It is also possible to refer to subsequently opened parentheses, by writing references such as (?+2). However, these cannot be recursive because the reference is not inside the parentheses that are referenced. They are always “subroutine” calls, as described in the next section.

An alternative approach is to use named parentheses instead. The Perl syntax for this is (?&name); PCRE’s earlier syntax (?P>name) is also supported. We could rewrite the above example as follows:

```
(?<pn> \( ( (?>[^()]+) | (?&pn) )* \) )
```

If there is more than one subpattern with the same name, the earliest one is used.

This particular example pattern that we have been looking at contains nested unlimited repeats, and so the use of atomic grouping for matching strings of non-parentheses is important when applying the pattern to strings that do not match. For example, when this pattern is applied to

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

it yields “no match” quickly. However, if atomic grouping is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject, and all have to be tested before failure can be reported.

At the end of a match, the values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If the pattern above is matched against

(ab(cd)ef)

the value for the capturing parentheses is “ef”, which is the last value taken on at the top level. If additional parentheses are added, giving

```
\( ( ( (?>[^()]+) | (?R) )* ) \)
   ^                        ^
   ^                        ^
```

the string they capture is “ab(cd)ef”, the contents of the top level parentheses.

Do not confuse the (?R) item with the condition (R), which tests for recursion. Consider this pattern, which matches text in angle brackets, allowing for arbitrary nesting. Only digits are allowed in nested brackets (that is, when recursing), whereas any characters are permitted at the outer level.

< (?: (?(R) \d++ | [^<>]*+) | (?R)) * >

In this pattern, (?(R) is the start of a conditional subpattern, with two different alternatives for the recursive and non-recursive cases. The (?R) item is the actual recursive call.

Subpatterns as subroutines

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. The “called” subpattern may be defined before or after the reference. A numbered reference can be absolute or relative, as in these examples:

- (...(absolute)...)...(?2)...
- (...(relative)...)...(?-1)...
- (...(?+1)...(relative)...

An earlier example pointed out that the pattern

(sens|respons)e and \1ibility

matches “sense and sensibility” and “response and responsibility”, but not “sense and responsibility”. If instead the pattern

(sens|respons)e and (?1)ibility

is used, it does match “sense and responsibility” as well as the other two strings. Another example is given in the discussion of DEFINE above.

Like recursive subpatterns, a “subroutine” call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure.

When a subpattern is used as a subroutine, processing options such as case-independence are fixed when the subpattern is defined. They cannot be changed for different calls. For example, consider this pattern:

```
(abc)(?i:(?-1))
```

It matches “abcabc”. It does not match “abcABC” because the change of processing option does not affect the called subpattern.

Backtracking control

Perl 5.10 introduced a number of “Special Backtracking Control Verbs”, which are described in the Perl documentation as “experimental and subject to change or removal in a future version of Perl”. It goes on to say: “Their usage in production code should be noted to avoid problems during upgrades.” The same remarks apply to the PCRE features described in this section.

The new verbs make use of what was previously invalid syntax: an opening parenthesis followed by an asterisk. In Perl, they are generally of the form (*VERB:ARG) but PCRE does not support the use of arguments, so its general form is just (*VERB). Any number of these verbs may occur in a pattern. There are two kinds:

Verbs that act immediately

The following verbs act as soon as they are encountered:

```
(*ACCEPT)
```

This verb causes the match to end successfully, skipping the remainder of the pattern. When inside a recursion, only the innermost pattern is ended immediately. PCRE differs from Perl in what happens if the (*ACCEPT) is inside capturing parentheses. In Perl, the data so far is captured: in PCRE no data is captured. For example:

```
A(A|B(*ACCEPT)|C)D
```

This matches “AB”, “AAD”, or “ACD”, but when it matches “AB”, no data is captured.

```
(*FAIL) or (*F)
```

This verb causes the match to fail, forcing backtracking to occur. It is equivalent to (?!) but easier to read. The Perl documentation notes that it is probably useful only when combined with (?{}) or (??{}). Those are, of course, Perl features that are not present in PCRE. The nearest equivalent is the callout feature, as for example in this pattern:

```
a+(?C)(*FAIL)
```

A match with the string “aaaa” always fails, but the callout is taken before each backtrack happens (in this example, 10 times).

Verbs that act after backtracking

The following verbs do nothing when they are encountered. Matching continues with what follows, but if there is no subsequent match, a failure is forced. The verbs differ in exactly what kind of failure occurs.

(*COMMIT)

This verb causes the whole match to fail outright if the rest of the pattern does not match. Even if the pattern is unanchored, no further attempts to find a match by advancing the start point take place. Once (*COMMIT) has been passed, `re:run/3` is committed to finding a match at the current starting point, or not at all. For example:

a+(*COMMIT)b

This matches “xxaab” but not “aacaab”. It can be thought of as a kind of dynamic anchor, or “I’ve started, so I must finish.”

(*PRUNE)

This verb causes the match to fail at the current position if the rest of the pattern does not match. If the pattern is unanchored, the normal “bumpalong” advance to the next starting character then happens. Backtracking can occur as usual to the left of (*PRUNE), or when matching to the right of (*PRUNE), but if there is no match to the right, backtracking cannot cross (*PRUNE). In simple cases, the use of (*PRUNE) is just an alternative to an atomic group or possessive quantifier, but there are some uses of (*PRUNE) that cannot be expressed in any other way.

(*SKIP)

This verb is like (*PRUNE), except that if the pattern is unanchored, the “bumpalong” advance is not to the next character, but to the position in the subject where (*SKIP) was encountered. (*SKIP) signifies that whatever text was matched leading up to it cannot be part of a successful match. Consider:

a+(*SKIP)b

If the subject is “aaaac...”, after the first match attempt fails (starting at the first character in the string), the starting point skips on to start the next attempt at “c”. Note that a possessive quantifer does not have the same effect in this example; although it would suppress backtracking during the first match attempt, the second attempt would start at the second character instead of skipping on to “c”.

(*THEN)

This verb causes a skip to the next alternation if the rest of the pattern does not match. That is, it cancels pending backtracking, but only within the current alternation. Its name comes from the observation that it can be used for a pattern-based if-then-else block:

(COND1 (*THEN) FOO | COND2 (*THEN) BAR | COND3 (*THEN) BAZ) ...

If the COND1 pattern matches, FOO is tried (and possibly further items after the end of the group if FOO succeeds); on failure the matcher skips to the second alternative and tries COND2, without backtracking into COND1. If (*THEN) is used outside of any alternation, it acts exactly like (*PRUNE).

regexp

Erlang Module

This module contains functions for regular expression matching and substitution.

Exports

`match(String, RegExp) -> MatchRes`

Types:

- String = RegExp = string()
- MatchRes = {match,Start,Length} | nomatch | {error,errordesc()}
- Start = Length = integer()

Finds the first, longest match of the regular expression `RegExp` in `String`. This function searches for the longest possible match and returns the first one found if there are several expressions of the same length. It returns as follows:

{`match`,`Start`,`Length`} if the match succeeded. `Start` is the starting position of the match, and `Length` is the length of the matching string.

`nomatch` if there were no matching characters.

{`error`,`Error`} if there was an error in `RegExp`.

`first_match(String, RegExp) -> MatchRes`

Types:

- String = RegExp = string()
- MatchRes = {match,Start,Length} | nomatch | {error,errordesc()}
- Start = Length = integer()

Finds the first match of the regular expression `RegExp` in `String`. This call is usually faster than `match` and it is also a useful way to ascertain that a match exists. It returns as follows:

{`match`,`Start`,`Length`} if the match succeeded. `Start` is the starting position of the match and `Length` is the length of the matching string.

`nomatch` if there were no matching characters.

{`error`,`Error`} if there was an error in `RegExp`.

`matches(String, RegExp) -> MatchRes`

Types:

- String = RegExp = string()

- MatchRes = {match, Matches} | {error, errordesc()}
- Matches = list()

Finds all non-overlapping matches of the expression `RegExp` in `String`. It returns as follows:

{`match, Matches`} if the regular expression was correct. The list will be empty if there was no match. Each element in the list looks like {`Start, Length`}, where `Start` is the starting position of the match, and `Length` is the length of the matching string.

{`error,Error`} if there was an error in `RegExp`.

`sub(String, RegExp, New) -> SubRes`

Types:

- String = RegExp = New = string()
- SubRes = {ok,NewString,RepCount} | {error,errordesc()}
- RepCount = integer()

Substitutes the first occurrence of a substring matching `RegExp` in `String` with the string `New`. A `&` in the string `New` is replaced by the matched substring of `String`. `\&` puts a literal `&` into the replacement string. It returns as follows:

{`ok,NewString,RepCount`} if `RegExp` is correct. `RepCount` is the number of replacements which have been made (this will be either 0 or 1).

{`error, Error`} if there is an error in `RegExp`.

`gsub(String, RegExp, New) -> SubRes`

Types:

- String = RegExp = New = string()
- SubRes = {ok,NewString,RepCount} | {error,errordesc()}
- RepCount = integer()

The same as `sub`, except that all non-overlapping occurrences of a substring matching `RegExp` in `String` are replaced by the string `New`. It returns:

{`ok,NewString,RepCount`} if `RegExp` is correct. `RepCount` is the number of replacements which have been made.

{`error, Error`} if there is an error in `RegExp`.

`split(String, RegExp) -> SplitRes`

Types:

- String = RegExp = string()
- SubRes = {ok,FieldList} | {error,errordesc()}
- Fieldlist = [string()]

`String` is split into fields (sub-strings) by the regular expression `RegExp`.

If the separator expression is `" "` (a single space), then the fields are separated by blanks and/or tabs and leading and trailing blanks and tabs are discarded. For all other values of the separator, leading and trailing blanks and tabs are not discarded. It returns:

{ok, FieldList} to indicate that the string has been split up into the fields of FieldList.

{error, Error} if there is an error in RegExp.

sh_to_awk(ShRegExp) -> AwkRegExp

Types:

- ShRegExp AwkRegExp = string()
- SubRes = {ok,NewString,RepCount} | {error,errordesc()}
- RepCount = integer()

Converts the sh type regular expression ShRegExp into a full AWK regular expression. Returns the converted regular expression string. sh expressions are used in the shell for matching file names and have the following special characters:

* matches any string including the null string.

? matches any single character.

[...] matches any of the enclosed characters. Character ranges are specified by a pair of characters separated by a -. If the first character after [is a !, then any character not enclosed is matched.

It may sometimes be more practical to use sh type expansions as they are simpler and easier to use, even though they are not as powerful.

parse(RegExp) -> ParseRes

Types:

- RegExp = string()
- ParseRes = {ok,RE} | {error,errordesc()}

Parses the regular expression RegExp and builds the internal representation used in the other regular expression functions. Such representations can be used in all of the other functions instead of a regular expression string. This is more efficient when the same regular expression is used in many strings. It returns:

{ok, RE}if RegExpis correct and REis the internal representation.

{error, Error}if there is an error in RegExpString.

format_error(ErrorDescriptor) -> Chars

Types:

- ErrorDescriptor = errordesc()
- Chars = [char() | Chars]

Returns a string which describes the error ErrorDescriptor returned when there is an error in a regular expression.

Regular Expressions

The regular expressions allowed here is a subset of the set found in egrep and in the AWK programming language, as defined in the book, The AWK Programming Language, by A. V. Aho, B. W. Kernighan, P. J. Weinberger. They are composed of the following characters:

c matches the non-metacharacter c.

\c matches the escape sequence or literal character c.

. matches any character.

^ matches the beginning of a string.

$ matches the end of a string.

[abc...] character class, which matches any of the characters abc... Character ranges are specified by a pair of characters separated by a -.

[^abc...] negated character class, which matches any character except abc....

r1 | r2 alternation. It matches either r1 or r2.

r1r2 concatenation. It matches r1 and then r2.

r+ matches one or more rs.

r* matches zero or more rs.

r? matches zero or one rs.

(r) grouping. It matches r.

The escape sequences allowed are the same as for Erlang strings:

\b backspace

\f form feed

\ newline (line feed)

\r carriage return

\t tab

\e escape

\v vertical tab

\s space

\d delete

\ddd the octal value ddd

\c any other character literally, for example \\ for backslash, \" for “)

To make these functions easier to use, in combination with the function io:get_line which terminates the input line with a new line, the $ characters also matches a string ending with "...\ ". The following examples define Erlang data types:

```
Atoms      [a-z][0-9a-zA-Z_]*

Variables [A-Z_][0-9a-zA-Z_]*

Floats     (\+|-)?[0-9]+\.[0-9]+((E|e)(\+|-)?[0-9]+)?
```

Regular expressions are written as Erlang strings when used with the functions in this module. This means that any \ or " characters in a regular expression string must be written with \ as they are also escape characters for the string. For example, the regular expression string for Erlang floats is:
`"(\\+|-)?[0-9]+\\.[0-9]+((E|e)(\\+|-)?[0-9]+)?"`.

It is not really necessary to have the escape sequences as part of the regular expression syntax as they can always be generated directly in the string. They are included for completeness and can they can also be useful when generating regular expressions, or when they are entered other than with Erlang strings.

sets

Erlang Module

Sets are collections of elements with no duplicate elements. The representation of a set is not defined.

This module provides exactly the same interface as the module `ordsets` but with a defined representation. One difference is that while this module considers two elements as different if they do not match (`=:=`), `ordsets` considers two elements as different if and only if they do not compare equal (`==`).

DATA TYPES

```
set()
  as returned by new/0
```

Exports

`new() -> Set`

Types:

- Set = set()

Returns a new empty set.

`is_set(Set) -> bool()`

Types:

- Set = term()

Returns `true` if `Set` is a set of elements, otherwise `false`.

`size(Set) -> int()`

Types:

- Set = term()

Returns the number of elements in `Set`.

`to_list(Set) -> List`

Types:

- Set = set()
- List = [term()]

Returns the elements of `Set` as a list.

`from_list(List) -> Set`

Types:

- List = [term()]
- Set = set()

Returns an set of the elements in `List`.

`is_element(Element, Set) -> bool()`

Types:

- Element = term()
- Set = set()

Returns `true` if `Element` is an element of `Set`, otherwise `false`.

`add_element(Element, Set1) -> Set2`

Types:

- Element = term()
- Set1 = Set2 = set()

Returns a new set formed from `Set1` with `Element` inserted.

`del_element(Element, Set1) -> Set2`

Types:

- Element = term()
- Set1 = Set2 = set()

Returns `Set1`, but with `Element` removed.

`union(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = set()

Returns the merged (union) set of `Set1` and `Set2`.

`union(SetList) -> Set`

Types:

- SetList = [set()]
- Set = set()

Returns the merged (union) set of the list of sets.

`intersection(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = set()

Returns the intersection of `Set1` and `Set2`.

`intersection(SetList) -> Set`

Types:

- SetList = [set()]
- Set = set()

Returns the intersection of the non-empty list of sets.

`subtract(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = set()

Returns only the elements of `Set1` which are not also elements of `Set2`.

`is_subset(Set1, Set2) -> bool()`

Types:

- Set1 = Set2 = set()

Returns `true` when every element of `Set1` is also a member of `Set2`, otherwise `false`.

`fold(Function, Acc0, Set) -> Acc1`

Types:

- Function = fun (E, AccIn) -> AccOut
- Acc0 = Acc1 = AccIn = AccOut = term()
- Set = set()

Fold `Function` over every element in `Set` returning the final value of the accumulator.

`filter(Pred, Set1) -> Set2`

Types:

- Pred = fun (E) -> bool()
- Set1 = Set2 = set()

Filter elements in `Set1` with boolean function `Fun`.

See Also

ordsets(3) [page 270], gb_sets(3) [page 177]

shell

Erlang Module

The module `shell` implements an Erlang shell.

The shell is a user interface program for entering expression sequences. The expressions are evaluated and a value is returned. A history mechanism saves previous commands and their values, which can then be incorporated in later commands. How many commands and results to save can be determined by the user, either interactively, by calling `shell:history/1` and `shell:results/1`, or by setting the application configuration parameters `shell_history_length` and `shell_saved_results` for the application STDLIB.

The shell uses a helper process for evaluating commands in order to protect the history mechanism from exceptions. By default the evaluator process is killed when an exception occurs, but by calling `shell:catch_exception/1` or by setting the application configuration parameter `shell_catch_exception` for the application STDLIB this behavior can be changed. See also the example below.

Variable bindings, and local process dictionary changes which are generated in user expressions are preserved, and the variables can be used in later commands to access their values. The bindings can also be forgotten so the variables can be re-used.

The special shell commands all have the syntax of (local) function calls. They are evaluated as normal function calls and many commands can be used in one expression sequence.

If a command (local function call) is not recognized by the shell, an attempt is first made to find the function in the module `user_default`, where customized local commands can be placed. If found, then the function is evaluated. Otherwise, an attempt is made to evaluate the function in the module `shell_default`. The module `user_default` must be explicitly loaded.

The shell also permits the user to start multiple concurrent jobs. A job can be regarded as a set of processes which can communicate with the shell.

There is some support for reading and printing records in the shell. During compilation record expressions are translated to tuple expressions. In runtime it is not known whether a tuple actually represents a record. Nor are the record definitions used by compiler available at runtime. So in order to read the record syntax and print tuples as records when possible, record definitions have to be maintained by the shell itself. The shell commands for reading, defining, forgetting, listing, and printing records are described below. Note that each job has its own set of record definitions. To facilitate matters record definitions in the modules `shell_default` and `user_default` (if loaded) are read each time a new job is started. For instance, adding the line

```
-include_lib("kernel/include/file.hrl").
```

to `user_default` makes the definition of `file_info` readily available in the shell.

The shell runs in two modes:

- Normal (possibly restricted) mode, in which commands can be edited and expressions evaluated.
- Job Control Mode JCL, in which jobs can be started, killed, detached and connected.

Only the currently connected job can 'talk' to the shell.

Shell Commands

b() Prints the current variable bindings.

f() Removes all variable bindings.

f(X) Removes the binding of variable X.

h() Prints the history list.

history(N) Sets the number of previous commands to keep in the history list to N. The previous number is returned. The default number is 20.

results(N) Sets the number of results from previous commands to keep in the history list to N. The previous number is returned. The default number is 20.

e(N) Repeats the command N, if N is positive. If it is negative, the Nth previous command is repeated (i.e., e(-1) repeats the previous command).

v(N) Uses the return value of the command N in the current command, if N is positive. If it is negative, the return value of the Nth previous command is used (i.e., v(-1) uses the value of the previous command).

help() Evaluates shell_default:help().

c(File) Evaluates shell_default:c(File). This compiles and loads code in File and purges old versions of code, if necessary. Assumes that the file and module names are the same.

catch_exception(Bool) Sets the exception handling of the evaluator process. The previous exception handling is returned. The default (false) is to kill the evaluator process when an exception occurs, which causes the shell to create a new evaluator process. When the exception handling is set to true the evaluator process lives on which means that for instance ports and ETS tables as well as processes linked to the evaluator process survive the exception.

rd(RecordName, RecordDefinition) Defines a record in the shell. RecordName is an atom and RecordDefinition lists the field names and the default values. Usually record definitions are made known to the shell by use of the rr commands described below, but sometimes it is handy to define records on the fly.

rf() Removes all record definitions, then reads record definitions from the modules shell_default and user_default (if loaded). Returns the names of the records defined.

rf(RecordNames) Removes selected record definitions. RecordNames is a record name or a list of record names. Use '_' to remove all record definitions.

rl() Prints all record definitions.

rl(RecordNames) Prints selected record definitions. RecordNames is a record name or a list of record names.

rp(Term) Prints a term using the record definitions known to the shell. All of Term is printed; the depth is not limited as is the case when a return value is printed.

`rr(Module)` Reads record definitions from a module's BEAM file. If there are no record definitions in the BEAM file, the source file is located and read instead. Returns the names of the record definitions read. `Module` is an atom.

`rr(Wildcard)` Reads record definitions from files. Existing definitions of any of the record names read are replaced. `Wildcard` is a wildcard string as defined in `filelib(3)` but not an atom.

`rr(WildcardOrModule, RecordNames)` Reads record definitions from files but discards record names not mentioned in `RecordNames` (a record name or a list of record names).

`rr(WildcardOrModule, RecordNames, Options)` Reads record definitions from files. The compiler options `{i,Dir}`, `{d,Macro}`, and `{d,Macro,Value}` are recognized and used for setting up the include path and macro definitions. Use `'_'` as value of `RecordNames` to read all record definitions.

Example

The following example is a long dialogue with the shell. Commands starting with > are inputs to the shell. All other lines are output from the shell. All commands in this example are explained at the end of the dialogue. .

```
strider 1> erl
Erlang (BEAM) emulator version 5.3 [hipe] [threads:0]

Eshell V5.3  (abort with ^G)
1>Str = "abcd".
"abcd"
2> L = length(Str).
4
3> Descriptor = {L, list_to_atom(Str)}.
{4,abcd}
4> L.
4
5> b().
Descriptor = {4,abcd}
L = 4
Str = "abcd"
ok
6> f(L).
ok
7> b().
Descriptor = {4,abcd}
Str = "abcd"
ok
8> f(L).
ok
9> {L, _} = Descriptor.
{4,abcd}
10> L.
4
11> {P, Q, R} = Descriptor.
** exception error: no match of right hand side value {4,abcd}
```

```
12> P.
* 1: variable 'P' is unbound **
13> Descriptor.
{4,abcd}
14>{P, Q} = Descriptor.
{4,abcd}
15> P.
4
16> f().
ok
17> put(aa, hello).
undefined
18> get(aa).
hello
19> Y = test1:demo(1).
11
20> get().
[{aa,worked}]
21> put(aa, hello).
worked
22> Z = test1:demo(2).
** exception error: no match of right hand side value 1
     in function  test1:demo/1
23> Z.
* 1: variable 'Z' is unbound **
24> get(aa).
hello
25> erase(), put(aa, hello).
undefined
26> spawn(test1, demo, [1]).
<0.57.0>
27> get(aa).
hello
28> io:format("hello hello\ ").
hello hello ok
29> e(28).
hello hello ok
30> v(28).
ok
31> c(ex).
{ok,ex}
32> rr(ex).
[rec]
33> rl(rec).
-record(rec,{a,b = val()}).
ok
34> #rec{}.
** exception error: undefined function shell_default:val/0
35> #rec{b = 3}.
#rec{a = undefined,b = 3}
36> rp(v(-1)).
#rec{a = undefined,b = 3}
ok
```

```
37> rd(rec, {f = orddict:new()}).
rec
38> #rec{}.
#rec{f = []}
ok
39> rd(rec, {c}), A.
* 1: variable 'A' is unbound **
40> #rec{}.
#rec{c = undefined}
ok
41> test1:loop(0).
Hello Number: 0
Hello Number: 1
Hello Number: 2
Hello Number: 3

User switch command
 --> i
 --> c
.
.
.
Hello Number: 3374
Hello Number: 3375
Hello Number: 3376
Hello Number: 3377
Hello Number: 3378
** exception exit: killed
42> E = ets:new(t, []).
17
43> ets:insert({d,1,2}).
** exception error: undefined function ets:insert/1
44> ets:insert(E, {d,1,2}).
** exception error: argument is of wrong type
     in function  ets:insert/2
        called as ets:insert(16,{d,1,2})
45> f(E).
ok
46> catch_exception(true).
false
47> E = ets:new(t, []).
18
48> ets:insert({d,1,2}).
* exception error: undefined function ets:insert/1
49> ets:insert(E, {d,1,2}).
true
50> halt().
strider 2>
```

Comments

Command 1 sets the variable `Str` to the string `"abcd"`.

Command 2 sets L to the length of the string evaluating the BIF atom_to_list.

Command 3 builds the tuple Descriptor.

Command 4 prints the value of the variable L.

Command 5 evaluates the internal shell command b(), which is an abbreviation of "bindings". This prints the current shell variables and their bindings. The ok at the end is the return value of the b() function.

Command 6 f(L) evaluates the internal shell command f(L) (abbreviation of "forget"). The value of the variable L is removed.

Command 7 prints the new bindings.

Command 8 has no effect since L has no value.

Command 9 performs a pattern matching operation on Descriptor, binding a new value to L.

Command 10 prints the current value of L.

Command 11 tries to match {P, Q, R} against Descriptor which is {4, abc}. The match fails and none of the new variables become bound. The printout starting with "** exception error:" is not the value of the expression (the expression had no value because its evaluation failed), but rather a warning printed by the system to inform the user that an error has occurred. The values of the other variables (L, Str, etc.) are unchanged.

Commands 12 and 13 show that P is unbound because the previous command failed, and that Descriptor has not changed.

Commands 14 and 15 show a correct match where P and Q are bound.

Command 16 clears all bindings.

The next few commands assume that test1:demo(X) is defined in the following way:

```
demo(X) ->
    put(aa, worked),
    X = 1,
    X + 10.
```

Commands 17 and 18 set and inspect the value of the item aa in the process dictionary.

Command 19 evaluates test1:demo(1). The evaluation succeeds and the changes made in the process dictionary become visible to the shell. The new value of the dictionary item aa can be seen in command 20.

Commands 21 and 22 change the value of the dictionary item aa to hello and call test1:demo(2). Evaluation fails and the changes made to the dictionary in test1:demo(2), before the error occurred, are discarded.

Commands 23 and 24 show that Z was not bound and that the dictionary item aa has retained its original value.

Commands 25, 26 and 27 show the effect of evaluating test1:demo(1) in the background. In this case, the expression is evaluated in a newly spawned process. Any changes made in the process dictionary are local to the newly spawned process and therefore not visible to the shell.

Commands 28, 29 and 30 use the history facilities of the shell.

Command 29 is e(28). This re-evaluates command 28. Command 30 is v(28). This uses the value (result) of command 28. In the cases of a pure function (a function with

no side effects), the result is the same. For a function with side effects, the result can be different.

The next few commands show some record manipulation. It is assumed that `ex.erl` defines a record like this:

```
-record(rec, {a, b = val()}).

val() ->
    3.
```

Commands 31 and 32 compiles the file `ex.erl` and reads the record definitions in `ex.beam`. If the compiler did not output any record definitions on the BEAM file, `rr(ex)` tries to read record definitions from the source file instead.

Command 33 prints the definition of the record named `rec`.

Command 34 tries to create a `rec` record, but fails since the function `val/0` is undefined. Command 35 shows the workaround: explicitly assign values to record fields that cannot otherwise be initialized.

Command 36 prints the newly created record using record definitions maintained by the shell.

Command 37 defines a record directly in the shell. The definition replaces the one read from the file `ex.beam`.

Command 38 creates a record using the new definition, and prints the result.

Command 39 and 40 show that record definitions are updated as side effects. The evaluation of the command fails but the definition of `rec` has been carried out.

For the next command, it is assumed that `test1:loop(N)` is defined in the following way:

```
loop(N) ->
    io:format("Hello Number: ~w~n", [N]),
    loop(N+1).
```

Command 41 evaluates `test1:loop(0)`, which puts the system into an infinite loop. At this point the user types `Control G`, which suspends output from the current process, which is stuck in a loop, and activates `JCL` mode. In `JCL` mode the user can start and stop jobs.

In this particular case, the `i` command (“interrupt”) is used to terminate the looping program, and the `c` command is used to connect to the shell again. Since the process was running in the background before we killed it, there will be more printouts before the “`** exception exit: killed`” message is shown.

Command 42 creates an ETS table.

Command 43 tries to insert a tuple into the ETS table but the first argument (the table) is missing. The exception kills the evaluator process.

Command 44 corrects the mistake, but the ETS table has been destroyed since it was owned by the killed evaluator process.

Command 46 sets the exception handling of the evaluator process to `true`. The exception handling can also be set when starting Erlang, like this: `erl -stdlib shell_catch_exception true`.

Command 48 makes the same mistake as in command 43, but this time the evaluator process lives on. The single star at the beginning of the printout signals that the exception has been caught.

Command 49 successfully inserts the tuple into the ETS table.

The `halt()` command exits the Erlang runtime system.

JCL Mode

When the shell starts, it starts a single evaluator process. This process, together with any local processes which it spawns, is referred to as a `job`. Only the current job, which is said to be `connected`, can perform operations with standard IO. All other jobs, which are said to be `detached`, are `blocked` if they attempt to use standard IO.

All jobs which do not use standard IO run in the normal way.

The shell escape key `^G` (Control G) detaches the current job and activates `JCL` mode. The `JCL` mode prompt is `"-->"`. If `"?"` is entered at the prompt, the following help message is displayed:

```
--> ?
c [nn]   - connect to job
i [nn]   - interrupt job
k [nn]   - kill job
j        - list all jobs
s        - start local shell
r [node] - start remote shell
q        - quit Erlang
?  | h   - this message
```

The `JCL` commands have the following meaning:

`c [nn]` Connects to job number <`nn`> or the current job. The standard shell is resumed. Operations which use standard IO by the current job will be interleaved with user inputs to the shell.

`i [nn]` Stops the current evaluator process for job number `nn` or the current job, but does not kill the shell process. Accordingly, any variable bindings and the process dictionary will be preserved and the job can be connected again. This command can be used to interrupt an endless loop.

`k [nn]` Kills job number `nn` or the current job. All spawned processes in the job are killed, provided they have not evaluated the `group_leader/1` BIF and are located on the local machine. Processes spawned on remote nodes will not be killed.

`j` Lists all jobs. A list of all known jobs is printed. The current job name is prefixed with '*'.

`s` Starts a new job. This will be assigned the new index `[nn]` which can be used in references.

`r [node]` Starts a remote job on `node`. This is used in distributed Erlang to allow a shell running on one node to control a number of applications running on a network of nodes.

`q` Quits Erlang. Note that this option is disabled if Erlang is started with the ignore break, `+Bi`, system flag (which may be useful e.g. when running a restricted shell, see below).

`?` Displays this message.

It is possible to alter the behavior of shell escape by means of the STDLIB application variable `shell_esc`. The value of the variable can be either `jcl` (`erl -stdlib shell_esc jcl`) or `abort` (`erl -stdlib shell_esc abort`). The first option sets ^G to activate JCL mode (which is also default behavior). The latter sets ^G to terminate the current shell and start a new one. JCL mode cannot be invoked when `shell_esc` is set to `abort`.

If you want an Erlang node to have a remote job active from the start (rather than the default local job), you start Erlang with the `-remsh` flag. Example: `erl -sname this_node -remsh other_node@other_host`

Restricted Shell

The shell may be started in a restricted mode. In this mode, the shell evaluates a function call only if allowed. This feature makes it possible to, for example, prevent a user from accidentally calling a function from the prompt that could harm a running system (useful in combination with the the system flag `+Bi`).

When the restricted shell evaluates an expression and encounters a function call or an operator application, it calls a callback function (with information about the function call in question). This callback function returns `true` to let the shell go ahead with the evaluation, or `false` to abort it. There are two possible callback functions for the user to implement:

`local_allowed(Func, ArgList, State) -> {true,NewState} | {false,NewState}`

to determine if the call to the local function `Func` with arguments `ArgList` should be allowed.

`non_local_allowed(FuncSpec, ArgList, State) -> {true,NewState} | {false,NewState} | {{redirect,NewFuncSpec,NewArgList},NewState}`

to determine if the call to non-local function `FuncSpec` (`{Module,Func}` or a fun) with arguments `ArgList` should be allowed. The return value `{redirect,NewFuncSpec,NewArgList}` can be used to let the shell evaluate some other function than the one specified by `FuncSpec` and `ArgList`.

These callback functions are in fact called from local and non-local evaluation function handlers, described in the erl_eval [page 116] manual page. (Arguments in `ArgList` are evaluated before the callback functions are called).

The `State` argument is a tuple `{ShellState,ExprState}`. The return value `NewState` has the same form. This may be used to carry a state between calls to the callback functions. Data saved in `ShellState` lives through an entire shell session. Data saved in `ExprState` lives only through the evaluation of the current expression.

There are two ways to start a restricted shell session:

- Use the STDLIB application variable `restricted_shell` and specify, as its value, the name of the callback module. Example (with callback functions implemented in callback_mod.erl): `$ erl -stdlib restricted_shell callback_mod`
- From a normal shell session, call function `shell:start_restricted/1`. This exits the current evaluator and starts a new one in restricted mode.

Notes:

- When restricted shell mode is activated or deactivated, new jobs started on the node will run in restricted or normal mode respectively.
- If restricted mode has been enabled on a particular node, remote shells connecting to this node will also run in restricted mode.
- The callback functions cannot be used to allow or disallow execution of functions called from compiled code (only functions called from expressions entered at the shell prompt).

Errors when loading the callback module is handled in different ways depending on how the restricted shell is activated:

- If the restricted shell is activated by setting the kernel variable during emulator startup and the callback module cannot be loaded, a default restricted shell allowing only the commands `q()` and `init:stop()` is used as fallback.
- If the restricted shell is activated using `shell:start_restricted/1` and the callback module cannot be loaded, an error report is sent to the error logger and the call returns `{error,Reason}`.

Exports

`history(N) -> integer()`

Types:

- N = integer()

Sets the number of previous commands to keep in the history list to `N`. The previous number is returned. The default number is 20.

`results(N) -> integer()`

Types:

- N = integer()

Sets the number of results from previous commands to keep in the history list to `N`. The previous number is returned. The default number is 20.

`catch_exception(Bool) -> Bool`

Types:

- Bool = bool()

Sets the exception handling of the evaluator process. The previous exception handling is returned. The default (`false`) is to kill the evaluator process when an exception occurs, which causes the shell to create a new evaluator process. When the exception handling is set to `true` the evaluator process lives on which means that for instance ports and ETS tables as well as processes linked to the evaluator process survive the exception.

`start_restricted(Module) -> ok | {error, Reason}`

Types:

- Module = atom()
- Reason = atom()

Exits a normal shell and starts a restricted shell. `Module` specifies the callback module for the functions `local_allowed/3` and `non_local_allowed/3`. The function is meant to be called from the shell.

If the callback module cannot be loaded, an error tuple is returned. The `Reason` in the error tuple is the one returned by the code loader when trying to load the code of the callback module.

`stop_restricted() -> ok`

Exits a restricted shell and starts a normal shell. The function is meant to be called from the shell.

shell_default

Erlang Module

The functions in shell_default are called when no module name is given in a shell command.

Consider the following shell dialogue:

```
1 > lists:reverse("abc").
"cba"
2 > c(foo).
{ok, foo}
```

In command one, the module lists is called. In command two, no module name is specified. The shell searches the modules user_default followed by shell_default for the function foo/1.

shell_default is intended for "system wide" customizations to the shell. user_default is intended for "local" or individual user customizations.

Hint

To add your own commands to the shell, create a module called user_default and add the commands you want. Then add the following line as the *first* line in your .erlang file in your home directory.

```
code:load_abs("$PATH/user_default").
```

$PATH is the directory where your user_default module can be found.

slave

Erlang Module

This module provides functions for starting Erlang slave nodes. All slave nodes which are started by a master will terminate automatically when the master terminates. All TTY output produced at the slave will be sent back to the master node. File I/O is done via the master.

Slave nodes on other hosts than the current one are started with the program `rsh`. The user must be allowed to `rsh` to the remote hosts without being prompted for a password. This can be arranged in a number of ways (refer to the `rsh` documentation for details). A slave node started on the same host as the master inherits certain environment values from the master, such as the current directory and the environment variables. For what can be assumed about the environment when a slave is started on another host, read the documentation for the `rsh` program.

An alternative to the `rsh` program can be specified on the command line to `erl` as follows: `-rsh Program`.

The slave node should use the same file system at the master. At least, Erlang/OTP should be installed in the same place on both computers and the same version of Erlang should be used.

Currently, a node running on Windows NT can only start slave nodes on the host on which it is running.

The master node must be alive.

Exports

```
start(Host) ->
start(Host, Name) ->
start(Host, Name, Args) -> {ok, Node} | {error, Reason}
```

Types:

- Host = Name = atom()
- Args = string()
- Node = node()
- Reason = timeout | no_rsh | {already_running, Node}

Starts a slave node on the host `Host`. Host names need not necessarily be specified as fully qualified names; short names can also be used. This is the same condition that applies to names of distributed Erlang nodes.

The name of the started node will be `Name@Host`. If no name is provided, the name will be the same as the node which executes the call (with the exception of the host name part of the node name).

The slave node resets its `user` process so that all terminal I/O which is produced at the slave is automatically relayed to the master. Also, the file process will be relayed to the master.

The `Args` argument is used to set `erl` command line arguments. If provided, it is passed to the new node and can be used for a variety of purposes. See [erl(1)]

As an example, suppose that we want to start a slave node at host `H` with the node name `Name@H`, and we also want the slave node to have the following properties:

- directory `Dir` should be added to the code path;
- the Mnesia directory should be set to `M`;
- the unix `DISPLAY` environment variable should be set to the display of the master node.

The following code is executed to achieve this:

```
E = " -env DISPLAY " ++ net_adm:localhost() ++ ":0 ",
Arg = "-mnesia_dir " ++ M ++ " -pa " ++ Dir ++ E,
slave:start(H, Name, Arg).
```

If successful, the function returns {`ok, Node`}, where `Node` is the name of the new node. Otherwise it returns {`error, Reason`}, where `Reason` can be one of:

`timeout` The master node failed to get in contact with the slave node. This can happen in a number of circumstances:

- Erlang/OTP is not installed on the remote host
- the file system on the other host has a different structure to the the master
- the Erlang nodes have different cookies.

`no_rsh` There is no `rsh` program on the computer.

{`already_running, Node`} A node with the name `Name@Host` already exists.

```
start_link(Host) ->
start_link(Host, Name) ->
start_link(Host, Name, Args) -> {ok, Node} | {error, Reason}
```

Types:

- Host = Name = atom()
- Args = string()
- Node = node()
- Reason = timeout | no_rsh | {already_running, Node}

Starts a slave node in the same way as `start/1,2,3`, except that the slave node is linked to the currently executing process. If that process terminates, the slave node also terminates.

See `start/1,2,3` for a description of arguments and return values.

`stop(Node) -> ok`

Types:

- Node = node()

Stops (kills) a node.

`pseudo([Master | ServerList]) -> ok`

Types:

- Master = node()
- ServerList = [atom()]

Calls `pseudo(Master, ServerList)`. If we want to start a node from the command line and set up a number of pseudo servers, an Erlang runtime system can be started as follows:

```
% erl -name abc -s slave pseudo klacke@super x --
```

`pseudo(Master, ServerList) -> ok`

Types:

- Master = node()
- ServerList = [atom()]

Starts a number of pseudo servers. A pseudo server is a server with a registered name which does absolutely nothing but pass on all message to the real server which executes at a master node. A pseudo server is an intermediary which only has the same registered name as the real server.

For example, if we have started a slave node `N` and want to execute `pxw` graphics code on this node, we can start the server `pxw_server` as a pseudo server at the slave node. The following code illustrates:

```
rpc:call(N, slave, pseudo, [node(), [pxw_server]]).
```

`relay(Pid)`

Types:

- Pid = pid()

Runs a pseudo server. This function never returns any value and the process which executes the function will receive messages. All messages received will simply be passed on to `Pid`.

sofs

Erlang Module

The `sofs` module implements operations on finite sets and relations represented as sets. Intuitively, a set is a collection of elements; every element belongs to the set, and the set contains every element.

Given a set A and a sentence S(x), where x is a free variable, a new set B whose elements are exactly those elements of A for which S(x) holds can be formed, this is denoted B= {xinA: S(x)}. Sentences are expressed using the logical operators “for some” (or “there exists”), “for all”, “and”, “or”, “not”. If the existence of a set containing all the specified elements is known (as will always be the case in this module), we write B= {x: S(x)}.

The *unordered set* containing the elements a, b and c is denoted {a,b,c}. This notation is not to be confused with tuples. The *ordered pair* of a and b, with first *coordinate* a and second coordinate b, is denoted (a,b). An ordered pair is an *ordered set* of two elements. In this module ordered sets can contain one, two or more elements, and parentheses are used to enclose the elements. Unordered sets and ordered sets are orthogonal, again in this module; there is no unordered set equal to any ordered set.

The set that contains no elements is called the *empty set*. If two sets A and B contain the same elements, then A is *equal* to B, denoted A=B. Two ordered sets are equal if they contain the same number of elements and have equal elements at each coordinate. If a set A contains all elements that B contains, then B is a *subset* of A. The *union* of two sets A and B is the smallest set that contains all elements of A and all elements of B. The *intersection* of two sets A and B is the set that contains all elements of A that belong to B. Two sets are *disjoint* if their intersection is the empty set. The *difference* of two sets A and B is the set that contains all elements of A that do not belong to B. The *symmetric difference* of two sets is the set that contains those element that belong to either of the two sets, but not both. The *union* of a collection of sets is the smallest set that contains all the elements that belong to at least one set of the collection. The *intersection* of a non-empty collection of sets is the set that contains all elements that belong to every set of the collection.

The *Cartesian product* of two sets X and Y, denoted XY, is the set {a: a= (x,y) for some xinX and for some yinY}. A *relation* is a subset of XY. Let R be a relation. The fact that (x,y) belongs to R is written as xRy. Since relations are sets, the definitions of the last paragraph (subset, union, and so on) apply to relations as well. The *domain* of R is the set {x: xRy for some yinY}. The *range* of R is the set {y: xRy for some xinX}. The *converse* of R is the set {a: a= (y,x) for some (x,y)inR}. If A is a subset of X, then the *image* of A under R is the set {y: xRy for some xinA}, and if B is a subset of Y, then the *inverse image* of B is the set {x: xRy for some yinB}. If R is a relation from X to Y and S is a relation from Y to Z, then the *relative product* of R and S is the relation T from X to Z defined so that xTz if and only if there exists an element y in Y such that xRy and ySz. The *restriction* of R to A is the set S defined so that xSy if and only if there exists an element x in A such that xRy. If S is a restriction of R to A, then R is an *extension* of S to X. If X=Y then we call R a relation *in* X. The *field* of a relation R in X is the union of

the domain of R and the range of R. If R is a relation in X, and if S is defined so that xSy if xRy and not x=y, then S is the *strict* relation corresponding to R, and vice versa, if S is a relation in X, and if R is defined so that xRy if xSy or x=y, then R is the *weak* relation corresponding to S. A relation R in X is *reflexive* if xRx for every element x of X; it is *symmetric* if xRy implies that yRx; and it is *transitive* if xRy and yRz imply that xRz.

A *function* F is a relation, a subset of XY, such that the domain of F is equal to X and such that for every x in X there is a unique element y in Y with (x,y) in F. The latter condition can be formulated as follows: if xFy and xFz then y=z. In this module, it will not be required that the domain of F be equal to X for a relation to be considered a function. Instead of writing (x,y)inF or xFy, we write F(x)=y when F is a function, and say that F maps x onto y, or that the value of F at x is y. Since functions are relations, the definitions of the last paragraph (domain, range, and so on) apply to functions as well. If the converse of a function F is a function F', then F' is called the *inverse* of F. The relative product of two functions F1 and F2 is called the *composite* of F1 and F2 if the range of F1 is a subset of the domain of F2.

Sometimes, when the range of a function is more important than the function itself, the function is called a *family*. The domain of a family is called the *index set*, and the range is called the *indexed set*. If x is a family from I to X, then x[i] denotes the value of the function at index i. The notation "a family in X" is used for such a family. When the indexed set is a set of subsets of a set X, then we call x a *family of subsets* of X. If x is a family of subsets of X, then the union of the range of x is called the *union of the family* x. If x is non-empty (the index set is non-empty), the *intersection of the family* x is the intersection of the range of x. In this module, the only families that will be considered are families of subsets of some set X; in the following the word "family" will be used for such families of subsets.

A *partition* of a set X is a collection S of non-empty subsets of X whose union is X and whose elements are pairwise disjoint. A relation in a set is an *equivalence relation* if it is reflexive, symmetric and transitive. If R is an equivalence relation in X, and x is an element of X, the *equivalence class* of x with respect to R is the set of all those elements y of X for which xRy holds. The equivalence classes constitute a partitioning of X. Conversely, if C is a partition of X, then the relation that holds for any two elements of X if they belong to the same equivalence class, is an equivalence relation induced by the partition C. If R is an equivalence relation in X, then the *canonical map* is the function that maps every element of X onto its equivalence class.

Relations as defined above (as sets of ordered pairs) will from now on be referred to as *binary relations*. We call a set of ordered sets (x[1],...,x[n]) an *(n-ary) relation*, and say that the relation is a subset of the Cartesian product X[1]...X[n] where x[i] is an element of X[i], 1<=i<=n. The *projection* of an n-ary relation R onto coordinate i is the set {x[i]: (x[1],...,x[i],...,x[n]) in R for some x[j]inX[j], 1<=j<=n and not i=j}. The projections of a binary relation R onto the first and second coordinates are the domain and the range of R respectively. The relative product of binary relations can be generalized to n-ary relations as follows. Let TR be an ordered set (R[1],...,R[n]) of binary relations from X to Y[i] and S a binary relation from (Y[1]...Y[n]) to Z. The *relative product* of TR and S is the binary relation T from X to Z defined so that xTz if and only if there exists an element y[i] in Y[i] for each 1<=i<=n such that xR[i]y[i] and (y[1],...,y[n])Sz. Now let TR be a an ordered set (R[1],...,R[n]) of binary relations from X[i] to Y[i] and S a subset of X[1]...X[n]. The *multiple relative product* of TR and and S is defined to be the set {z: z= ((x[1],...,x[n]), (y[1],...,y[n])) for some (x[1],...,x[n])inS and for some (x[i],y[i]) in R[i], 1<=i<=n}. The *natural join* of an n-ary relation R and an m-ary relation S on coordinate i and j is defined to be the set {z:

z= (x[1],...,x[n], y[1],...,y[j-1],y[j+1],...,y[m]) for some (x[1],...,x[n])inR and for some (y[1],...,y[m])inS such that x[i]=y[j]}.

The sets recognized by this module will be represented by elements of the relation Sets, defined as the smallest set such that:

- for every atom T except '_' and for every term X, (T,X) belongs to Sets (*atomic sets*);
- (['_'],[]) belongs to Sets (the *untyped empty set*);
- for every tuple T= {T[1],...,T[n]} and for every tuple X= {X[1],...,X[n]}, if (T[i],X[i]) belongs to Sets for every 1<=i<=n then (T,X) belongs to Sets (*ordered sets*);
- for every term T, if X is the empty list or a non-empty sorted list [X[1],...,X[n]] without duplicates such that (T,X[i]) belongs to Sets for every 1<=i<=n, then ([T],X) belongs to Sets (*typed unordered sets*).

An *external set* is an element of the range of Sets. A *type* is an element of the domain of Sets. If S is an element (T,X) of Sets, then T is a *valid type* of X, T is the type of S, and X is the external set of S. from_term/2 [page 383] creates a set from a type and an Erlang term turned into an external set.

The actual sets represented by Sets are the elements of the range of the function Set from Sets to Erlang terms and sets of Erlang terms:

- Set(T,Term)= Term, where T is an atom;
- Set({T[1],...,T[n]},{X[1],...,X[n]}) = (Set(T[1],X[1]),...,Set(T[n],X[n]));
- Set([T],[X[1],...,X[n]]) = {Set(T,X[1]),...,Set(T,X[n])};
- Set([T],[])= {}.

When there is no risk of confusion, elements of Sets will be identified with the sets they represent. For instance, if U is the result of calling `union/2` with S1 and S2 as arguments, then U is said to be the union of S1 and S2. A more precise formulation would be that Set(U) is the union of Set(S1) and Set(S2).

The types are used to implement the various conditions that sets need to fulfill. As an example, consider the relative product of two sets R and S, and recall that the relative product of R and S is defined if R is a binary relation to Y and S is a binary relation from Y. The function that implements the relative product, relative_product/2 [page 390], checks that the arguments represent binary relations by matching [{A,B}] against the type of the first argument (Arg1 say), and [{C,D}] against the type of the second argument (Arg2 say). The fact that [{A,B}] matches the type of Arg1 is to be interpreted as Arg1 representing a binary relation from X to Y, where X is defined as all sets Set(x) for some element x in Sets the type of which is A, and similarly for Y. In the same way Arg2 is interpreted as representing a binary relation from W to Z. Finally it is checked that B matches C, which is sufficient to ensure that W is equal to Y. The untyped empty set is handled separately: its type, ['_'], matches the type of any unordered set.

A few functions of this module (`drestriction/3`, `family_projection/2`, `partition/2`, `partition_family/2`, `projection/2`, `restriction/3`, `substitution/2`) accept an Erlang function as a means to modify each element of a given unordered set. Such a function, called SetFun in the following, can be specified as a functional object (fun), a tuple `{external,Fun}`, or an integer. If SetFun is specified as a fun, the fun is applied to each element of the given set and the return value is

assumed to be a set. If SetFun is specified as a tuple {external, Fun}, Fun is applied to the external set of each element of the given set and the return value is assumed to be an external set. Selecting the elements of an unordered set as external sets and assembling a new unordered set from a list of external sets is in the present implementation more efficient than modifying each element as a set. However, this optimization can only be utilized when the elements of the unordered set are atomic or ordered sets. It must also be the case that the type of the elements matches some clause of Fun (the type of the created set is the result of applying Fun to the type of the given set), and that Fun does nothing but selecting, duplicating or rearranging parts of the elements. Specifying a SetFun as an integer I is equivalent to specifying {external, fun(X)-> element(I,X)}, but is to be preferred since it makes it possible to handle this case even more efficiently. Examples of SetFuns:

```
{sofs, union}
fun(S) -> sofs:partition(1, S) end
{external, fun(A) -> A end}
{external, fun({A,_,C}) -> {C,A} end}
{external, fun({_,{_,C}}) -> C end}
{external, fun({_,{_,{_,E}=C}}) -> {E,{E,C}} end}
2
```

The order in which a SetFun is applied to the elements of an unordered set is not specified, and may change in future versions of sofs.

The execution time of the functions of this module is dominated by the time it takes to sort lists. When no sorting is needed, the execution time is in the worst case proportional to the sum of the sizes of the input arguments and the returned value. A few functions execute in constant time: from_external, is_empty_set, is_set, is_sofs_set, to_external, type.

The functions of this module exit the process with a badarg, bad_function, or type_mismatch message when given badly formed arguments or sets the types of which are not compatible.

Types

```
anyset() = -an unordered, ordered or atomic set-
binary_relation() = -a binary relation-
bool() = true | false
external_set() = -an external set-
family() = -a family (of subsets)-
function() = -a function-
ordset() = -an ordered set-
relation() = -an n-ary relation-
set() = -an unordered set-
set_of_sets() = -an unordered set of set()-
set_fun() = integer() >= 1
          | {external, fun(external_set()) -> external_set()}
          | fun(anyset()) -> anyset()
spec_fun() = {external, fun(external_set()) -> bool()}
           | fun(anyset()) -> bool()
type() = -a type-
```

Exports

`a_function(Tuples [, Type]) -> Function`

Types:

- Function = function()
- Tuples = [tuple()]
- Type = type()

Creates a function [page 373]. `a_function(F,T)` is equivalent to `from_term(F,T)`, if the result is a function. If no type [page 374] is explicitly given, `[{atom,atom}]` is used as type of the function.

`canonical_relation(SetOfSets) -> BinRel`

Types:

- BinRel = binary_relation()
- SetOfSets = set_of_sets()

Returns the binary relation containing the elements (E,Set) such that Set belongs to SetOfSets and E belongs to Set. If SetOfSets is a partition [page 373] of a set X and R is the equivalence relation in X induced by SetOfSets, then the returned relation is the canonical map [page 373] from X onto the equivalence classes with respect to R.

```
1> Ss = sofs:from_term([[a,b],[b,c]]),
CR = sofs:canonical_relation(Ss),
sofs:to_external(CR).
[{a,[a,b]},{b,[a,b]},{b,[b,c]},{c,[b,c]}]
```

`composite(Function1, Function2) -> Function3`

Types:

- Function1 = Function2 = Function3 = function()

Returns the composite [page 373] of the functions Function1 and Function2.

```
1> F1 = sofs:a_function([{a,1},{b,2},{c,2}]),
F2 = sofs:a_function([{1,x},{2,y},{3,z}]),
F = sofs:composite(F1, F2),
sofs:to_external(F).
[{a,x},{b,y},{c,y}]
```

`constant_function(Set, AnySet) -> Function`

Types:

- AnySet = anyset()
- Function = function()
- Set = set()

Creates the function [page 373] that maps each element of the set Set onto AnySet.

```
1> S = sofs:set([a,b]),
E = sofs:from_term(1),
R = sofs:constant_function(S, E),
sofs:to_external(R).
[{a,1},{b,1}]
```

`converse(BinRel1) -> BinRel2`

Types:

- BinRel1 = BinRel2 = binary_relation()

Returns the converse [page 372] of the binary relation BinRel1.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,a}]),
R2 = sofs:converse(R1),
sofs:to_external(R2).
[{a,1},{a,3},{b,2}]
```

`difference(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = set()

Returns the difference [page 372] of the sets Set1 and Set2.

`digraph_to_family(Graph [, Type]) -> Family`

Types:

- Graph = digraph() -see digraph(3)-
- Family = family()
- Type = type()

Creates a family [page 373] from the directed graph Graph. Each vertex a of Graph is represented by a pair (a,{b[1],...,b[n]}) where the b[i]'s are the out-neighbours of a. If no type is explicitly given, [{atom,[atom]}] is used as type of the family. It is assumed that Type is a valid type [page 374] of the external set of the family.

If G is a directed graph, it holds that the vertices and edges of G are the same as the vertices and edges of `family_to_digraph(digraph_to_family(G))`.

`domain(BinRel) -> Set`

Types:

- BinRel = binary_relation()
- Set = set()

Returns the domain [page 372] of the binary relation BinRel.

```
1> R = sofs:relation([{1,a},{1,b},{2,b},{2,c}]),
S = sofs:domain(R),
sofs:to_external(S).
[1,2]
```

`drestriction(BinRel1, Set) -> BinRel2`

Types:

- BinRel1 = BinRel2 = binary_relation()
- Set = set()

Returns the difference between the binary relation BinRel1 and the restriction [page 372] of BinRel1 to Set.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,c}]),
S = sofs:set([2,4,6]),
R2 = sofs:drestriction(R1, S),
sofs:to_external(R2).
[{1,a},{3,c}]
```

`drestriction(R,S)` is equivalent to `difference(R,restriction(R,S))`.

`drestriction(SetFun, Set1, Set2) -> Set3`

Types:

- SetFun = set_fun()
- Set1 = Set2 = Set3 = set()

Returns a subset of Set1 containing those elements that do not yield an element in Set2 as the result of applying SetFun.

```
1> SetFun = {external, fun({_A,B,C}) -> {B,C} end},
R1 = sofs:relation([{a,aa,1},{b,bb,2},{c,cc,3}]),
R2 = sofs:relation([{bb,2},{cc,3},{dd,4}]),
R3 = sofs:drestriction(SetFun, R1, R2),
sofs:to_external(R3).
[{a,aa,1}]
```

`drestriction(F,S1,S2)` is equivalent to `difference(S1,restriction(F,S1,S2))`.

`empty_set() -> Set`

Types:

- Set = set()

Returns the untyped empty set [page 374]. `empty_set()` is equivalent to `from_term([],['_'])`.

`extension(BinRel1, Set, AnySet) -> BinRel2`

Types:

- AnySet = anyset()
- BinRel1 = BinRel2 = binary_relation()
- Set = set()

Returns the extension [page 372] of BinRel1 such that for each element E in Set that does not belong to the domain [page 372] of BinRel1, BinRel2 contains the pair (E,AnySet).

```
1> S = sofs:set([b,c]),
A = sofs:empty_set(),
R = sofs:family([{a,[1,2]},{b,[3]}]),
X = sofs:extension(R, S, A),
sofs:to_external(X).
[{a,[1,2]},{b,[3]},{c,[]}]
```

`family(Tuples [, Type]) -> Family`

Types:

- Family = family()
- Tuples = [tuple()]
- Type = type()

Creates a family of subsets [page 373]. `family(F,T)` is equivalent to `from_term(F,T)`, if the result is a family. If no type [page 374] is explicitly given, `[{atom,[atom]}]` is used as type of the family.

`family_difference(Family1, Family2) -> Family3`

Types:

- Family1 = Family2 = Family3 = family()

If Family1 and Family2 are families [page 373], then Family3 is the family such that the index set is equal to the index set of Family1, and Family3[i] is the difference between Family1[i] and Family2[i] if Family2 maps i, Family1[i] otherwise.

```
1> F1 = sofs:family([{a,[1,2]},{b,[3,4]}]),
F2 = sofs:family([{b,[4,5]},{c,[6,7]}]),
F3 = sofs:family_difference(F1, F2),
sofs:to_external(F3).
[{a,[1,2]},{b,[3]}]
```

`family_domain(Family1) -> Family2`

Types:

- Family1 = Family2 = family()

If Family1 is a family [page 373] and Family1[i] is a binary relation for every i in the index set of Family1, then Family2 is the family with the same index set as Family1 such that Family2[i] is the domain [page 372] of Family1[i].

```
1> FR = sofs:from_term([{a,[{1,a},{2,b},{3,c}]},{b,[]},{c,[{4,d},{5,e}]}]),
F = sofs:family_domain(FR),
sofs:to_external(F).
[{a,[1,2,3]},{b,[]},{c,[4,5]}]
```

`family_field(Family1) -> Family2`

Types:

- Family1 = Family2 = family()

If Family1 is a family [page 373] and Family1[i] is a binary relation for every i in the index set of Family1, then Family2 is the family with the same index set as Family1 such that Family2[i] is the field [page 372] of Family1[i].

```
1> FR = sofs:from_term([{a,[{1,a},{2,b},{3,c}]},{b,[]},{c,[{4,d},{5,e}]}]),
F = sofs:family_field(FR),
sofs:to_external(F).
[{a,[1,2,3,a,b,c]},{b,[]},{c,[4,5,d,e]}]
```

`family_field(Family1)` is equivalent to `family_union(family_domain(Family1), family_range(Family1))`.

`family_intersection(Family1) -> Family2`

Types:

- Family1 = Family2 = family()

If Family1 is a family [page 373] and Family1[i] is a set of sets for every i in the index set of Family1, then Family2 is the family with the same index set as Family1 such that Family2[i] is the intersection [page 372] of Family1[i].

If Family1[i] is an empty set for some i, then the process exits with a `badarg` message.

```
1> F1 = sofs:from_term([{a,[[1,2,3],[2,3,4]]},{b,[[x,y,z],[x,y]]}]),
F2 = sofs:family_intersection(F1),
sofs:to_external(F2).
[{a,[2,3]},{b,[x,y]}]
```

`family_intersection(Family1, Family2) -> Family3`

Types:

- Family1 = Family2 = Family3 = family()

If Family1 and Family2 are families [page 373], then Family3 is the family such that the index set is the intersection of Family1's and Family2's index sets, and Family3[i] is the intersection of Family1[i] and Family2[i].

```
1> F1 = sofs:family([{a,[1,2]},{b,[3,4]},{c,[5,6]}]),
F2 = sofs:family([{b,[4,5]},{c,[7,8]},{d,[9,10]}]),
F3 = sofs:family_intersection(F1, F2),
sofs:to_external(F3).
[{b,[4]},{c,[]}]
```

`family_projection(SetFun, Family1) -> Family2`

Types:

- SetFun = set_fun()
- Family1 = Family2 = family()
- Set = set()

If Family1 is a family [page 373] then Family2 is the family with the same index set as Family1 such that Family2[i] is the result of calling SetFun with Family1[i] as argument.

```
1> F1 = sofs:from_term([{a,[[1,2],[2,3]]},{b,[[]]}]),
F2 = sofs:family_projection({sofs, union}, F1),
sofs:to_external(F2).
[{a,[1,2,3]},{b,[]}]
```

`family_range(Family1) -> Family2`

Types:

- Family1 = Family2 = family()

If Family1 is a family [page 373] and Family1[i] is a binary relation for every i in the index set of Family1, then Family2 is the family with the same index set as Family1 such that Family2[i] is the range [page 372] of Family1[i].

```
1> FR = sofs:from_term([{a,[{1,a},{2,b},{3,c}]},{b,[]},{c,[{4,d},{5,e}]}]),
F = sofs:family_range(FR),
sofs:to_external(F).
[{a,[a,b,c]},{b,[]},{c,[d,e]}]
```

`family_specification(Fun, Family1) -> Family2`

Types:

- Fun = spec_fun()
- Family1 = Family2 = family()

If Family1 is a family [page 373], then Family2 is the restriction [page 372] of Family1 to those elements i of the index set for which Fun applied to Family1[i] returns `true`. If Fun is a tuple {`external`,`Fun2`}, Fun2 is applied to the external set [page 374] of Family1[i], otherwise Fun is applied to Family1[i].

```
1> F1 = sofs:family([{a,[1,2,3]},{b,[1,2]},{c,[1]}]),
SpecFun = fun(S) -> sofs:no_elements(S) =:= 2 end,
F2 = sofs:family_specification(SpecFun, F1),
sofs:to_external(F2).
[{b,[1,2]}]
```

`family_to_digraph(Family [, GraphType]) -> Graph`

Types:

- Graph = digraph()
- Family = family()
- GraphType = -see digraph(3)-

Creates a directed graph from the family [page 373] Family. For each pair (a,{b[1],...,b[n]}) of Family, the vertex a as well the edges (a,b[i]) for 1<=i<=n are added to a newly created directed graph.

If no graph type is given, `digraph:new/1` is used for creating the directed graph, otherwise the GraphType argument is passed on as second argument to `digraph:new/2`.

It F is a family, it holds that F is a subset of `digraph_to_family(family_to_digraph(F),type(F))`. Equality holds if `union_of_family(F)` is a subset of `domain(F)`.

Creating a cycle in an acyclic graph exits the process with a `cyclic` message.

`family_to_relation(Family) -> BinRel`

Types:

- Family = family()
- BinRel = binary_relation()

If Family is a family [page 373], then BinRel is the binary relation containing all pairs (i,x) such that i belongs to the index set of Family and x belongs to Family[i].

```
1> F = sofs:family([{a,[]}, {b,[1]}, {c,[2,3]}]),
R = sofs:family_to_relation(F),
sofs:to_external(R).
[{b,1},{c,2},{c,3}]
```

```
family_union(Family1) -> Family2
```

Types:

- Family1 = Family2 = family()

If Family1 is a family [page 373] and Family1[i] is a set of sets for each i in the index set of Family1, then Family2 is the family with the same index set as Family1 such that Family2[i] is the union [page 372] of Family1[i].

```
1> F1 = sofs:from_term([{a,[[1,2],[2,3]]},{b,[[]]}]),
F2 = sofs:family_union(F1),
sofs:to_external(F2).
[{a,[1,2,3]},{b,[]}]
```

`family_union(F)` is equivalent to `family_projection({sofs,union},F)`.

```
family_union(Family1, Family2) -> Family3
```

Types:

- Family1 = Family2 = Family3 = family()

If Family1 and Family2 are families [page 373], then Family3 is the family such that the index set is the union of Family1's and Family2's index sets, and Family3[i] is the union of Family1[i] and Family2[i] if both maps i, Family1[i] or Family2[i] otherwise.

```
1> F1 = sofs:family([{a,[1,2]},{b,[3,4]},{c,[5,6]}]),
F2 = sofs:family([{b,[4,5]},{c,[7,8]},{d,[9,10]}]),
F3 = sofs:family_union(F1, F2),
sofs:to_external(F3).
[{a,[1,2]},{b,[3,4,5]},{c,[5,6,7,8]},{d,[9,10]}]
```

```
field(BinRel) -> Set
```

Types:

- BinRel = binary_relation()
- Set = set()

Returns the field [page 372] of the binary relation BinRel.

```
1> R = sofs:relation([{1,a},{1,b},{2,b},{2,c}]),
S = sofs:field(R),
sofs:to_external(S).
[1,2,a,b,c]
```

`field(R)` is equivalent to `union(domain(R), range(R))`.

```
from_external(ExternalSet, Type) -> AnySet
```

Types:

- ExternalSet = external_set()
- AnySet = anyset()
- Type = type()

Creates a set from the external set [page 374] ExternalSet and the type [page 374] Type. It is assumed that Type is a valid type [page 374] of ExternalSet.

```
from_sets(ListOfSets) -> Set
```

Types:

- Set = set()
- ListOfSets = [anyset()]

Returns the unordered set [page 374] containing the sets of the list ListOfSets.

```
1> S1 = sofs:relation([{a,1},{b,2}]),
S2 = sofs:relation([{x,3},{y,4}]),
S = sofs:from_sets([S1,S2]),
sofs:to_external(S).
[[{a,1},{b,2}],[{x,3},{y,4}]]
```

`from_sets(TupleOfSets) -> Ordset`

Types:

- Ordset = ordset()
- TupleOfSets = tuple-of(anyset())

Returns the ordered set [page 374] containing the sets of the non-empty tuple TupleOfSets.

`from_term(Term [, Type]) -> AnySet`

Types:

- AnySet = anyset()
- Term = term()
- Type = type()

Creates an element of Sets [page 374] by traversing the term Term, sorting lists, removing duplicates and deriving or verifying a valid type [page 374] for the so obtained external set. An explicitly given type [page 374] Type can be used to limit the depth of the traversal; an atomic type stops the traversal, as demonstrated by this example where “foo” and {“foo”} are left unmodified:

```
1> S = sofs:from_term([{{"foo"},[1,1]},{"foo",[2,2]}], [{atom,[atom]}]),
sofs:to_external(S).
[{{"foo"},[1]},{"foo",[2]}]
```

`from_term` can be used for creating atomic or ordered sets. The only purpose of such a set is that of later building unordered sets since all functions in this module that *do* anything operate on unordered sets. Creating unordered sets from a collection of ordered sets may be the way to go if the ordered sets are big and one does not want to waste heap by rebuilding the elements of the unordered set. An example showing that a set can be built “layer by layer”:

```
1> A = sofs:from_term(a),
S = sofs:set([1,2,3]),
P1 = sofs:from_sets({A,S}),
P2 = sofs:from_term({b,[6,5,4]}),
Ss = sofs:from_sets([P1,P2]),
sofs:to_external(Ss).
[{a,[1,2,3]},{b,[4,5,6]}]
```

Other functions that create sets are `from_external/2` and `from_sets/1`. Special cases of `from_term/2` are `a_function/1,2`, `empty_set/0`, `family/1,2`, `relation/1,2`, and `set/1,2`.

`image(BinRel, Set1) -> Set2`

Types:

- BinRel = binary_relation()
- Set1 = Set2 = set()

Returns the image [page 372] of the set Set1 under the binary relation BinRel.

```
1> R = sofs:relation([{1,a},{2,b},{2,c},{3,d}]),
S1 = sofs:set([1,2]),
S2 = sofs:image(R, S1),
sofs:to_external(S2).
[a,b,c]
```

`intersection(SetOfSets) -> Set`

Types:

- Set = set()
- SetOfSets = set_of_sets()

Returns the intersection [page 372] of the set of sets SetOfSets.

Intersecting an empty set of sets exits the process with a `badarg` message.

`intersection(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = set()

Returns the intersection [page 372] of Set1 and Set2.

`intersection_of_family(Family) -> Set`

Types:

- Family = family()
- Set = set()

Returns the intersection of the family [page 373] Family.

Intersecting an empty family exits the process with a `badarg` message.

```
1> F = sofs:family([{a,[0,2,4]},{b,[0,1,2]},{c,[2,3]}]),
S = sofs:intersection_of_family(F),
sofs:to_external(S).
[2]
```

`inverse(Function1) -> Function2`

Types:

- Function1 = Function2 = function()

Returns the inverse [page 373] of the function Function1.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,c}]),
R2 = sofs:inverse(R1),
sofs:to_external(R2).
[{a,1},{b,2},{c,3}]
```

`inverse_image(BinRel, Set1) -> Set2`

Types:

- BinRel = binary_relation()
- Set1 = Set2 = set()

Returns the inverse image [page 372] of Set1 under the binary relation BinRel.

```
1> R = sofs:relation([{1,a},{2,b},{2,c},{3,d}]),
S1 = sofs:set([c,d,e]),
S2 = sofs:inverse_image(R, S1),
sofs:to_external(S2).
[2,3]
```

`is_a_function(BinRel) -> Bool`

Types:

- Bool = bool()
- BinRel = binary_relation()

Returns `true` if the binary relation BinRel is a function [page 373] or the untyped empty set, `false` otherwise.

`is_disjoint(Set1, Set2) -> Bool`

Types:

- Bool = bool()
- Set1 = Set2 = set()

Returns `true` if Set1 and Set2 are disjoint [page 372], `false` otherwise.

`is_empty_set(AnySet) -> Bool`

Types:

- AnySet = anyset()
- Bool = bool()

Returns `true` if Set is an empty unordered set, `false` otherwise.

`is_equal(AnySet1, AnySet2) -> Bool`

Types:

- AnySet1 = AnySet2 = anyset()
- Bool = bool()

Returns `true` if the AnySet1 and AnySet2 are equal [page 372], `false` otherwise.

`is_set(AnySet) -> Bool`

Types:

- AnySet = anyset()
- Bool = bool()

Returns `true` if AnySet is an unordered set [page 374], and `false` if AnySet is an ordered set or an atomic set.

`is_sofs_set(Term) -> Bool`

Types:

- Bool = bool()
- Term = term()

Returns `true` if Term is an unordered set [page 374], an ordered set or an atomic set, `false` otherwise.

`is_subset(Set1, Set2) -> Bool`

Types:

- Bool = bool()
- Set1 = Set2 = set()

Returns `true` if Set1 is a subset [page 372] of Set2, `false` otherwise.

`is_type(Term) -> Bool`

Types:

- Bool = bool()
- Term = term()

Returns `true` if the term Term is a type [page 374].

`join(Relation1, I, Relation2, J) -> Relation3`

Types:

- Relation1 = Relation2 = Relation3 = relation()
- I = J = integer() > 0

Returns the natural join [page 373] of the relations Relation1 and Relation2 on coordinates I and J.

```
1> R1 = sofs:relation([{a,x,1},{b,y,2}]),
R2 = sofs:relation([{1,f,g},{1,h,i},{2,3,4}]),
J = sofs:join(R1, 3, R2, 1),
sofs:to_external(J).
[{a,x,1,f,g},{a,x,1,h,i},{b,y,2,3,4}]
```

`multiple_relative_product(TupleOfBinRels, BinRel1) -> BinRel2`

Types:

- TupleOfBinRels = tuple-of(BinRel)
- BinRel = BinRel1 = BinRel2 = binary_relation()

If TupleOfBinRels is a non-empty tuple {R[1],...,R[n]} of binary relations and BinRel1 is a binary relation, then BinRel2 is the multiple relative product [page 373] of the ordered set (R[i],...,R[n]) and BinRel1.

```
1> Ri = sofs:relation([{a,1},{b,2},{c,3}]),
R = sofs:relation([{a,b},{b,c},{c,a}]),
MP = sofs:multiple_relative_product({Ri, Ri}, R),
sofs:to_external(sofs:range(MP)).
[{1,2},{2,3},{3,1}]
```

`no_elements(ASet) -> NoElements`

Types:

- ASet = set() | ordset()
- NoElements = integer() >= 0

Returns the number of elements of the ordered or unordered set ASet.

`partition(SetOfSets) -> Partition`

Types:

- SetOfSets = set_of_sets()
- Partition = set()

Returns the partition [page 373] of the union of the set of sets SetOfSets such that two elements are considered equal if they belong to the same elements of SetOfSets.

```
1> Sets1 = sofs:from_term([[a,b,c],[d,e,f],[g,h,i]]),
Sets2 = sofs:from_term([[b,c,d],[e,f,g],[h,i,j]]),
P = sofs:partition(sofs:union(Sets1, Sets2)),
sofs:to_external(P).
[[a],[b,c],[d],[e,f],[g],[h,i],[j]]
```

`partition(SetFun, Set) -> Partition`

Types:

- SetFun = set_fun()
- Partition = set()
- Set = set()

Returns the partition [page 373] of Set such that two elements are considered equal if the results of applying SetFun are equal.

```
1> Ss = sofs:from_term([[a],[b],[c,d],[e,f]]),
SetFun = fun(S) -> sofs:from_term(sofs:no_elements(S)) end,
P = sofs:partition(SetFun, Ss),
sofs:to_external(P).
[[[a],[b]],[[c,d],[e,f]]]
```

`partition(SetFun, Set1, Set2) -> {Set3, Set4}`

Types:

- SetFun = set_fun()
- Set1 = Set2 = Set3 = Set4 = set()

Returns a pair of sets that, regarded as constituting a set, forms a partition [page 373] of Set1. If the result of applying SetFun to an element of Set1 yields an element in Set2, the element belongs to Set3, otherwise the element belongs to Set4.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,c}]),
S = sofs:set([2,4,6]),
{R2,R3} = sofs:partition(1, R1, S),
{sofs:to_external(R2),sofs:to_external(R3)}.
{[{2,b}],[{1,a},{3,c}]}
```

partition(F,S1,S2) is equivalent to {restriction(F,S1,S2), drestriction(F,S1,S2)}.

`partition_family(SetFun, Set) -> Family`

Types:

- Family = family()
- SetFun = set_fun()
- Set = set()

Returns the family [page 373] Family where the indexed set is a partition [page 373] of Set such that two elements are considered equal if the results of applying SetFun are the same value i. This i is the index that Family maps onto the equivalence class [page 373].

```
1> S = sofs:relation([{a,a,a,a},{a,a,b,b},{a,b,b,b}]),
SetFun = {external, fun({A,_,C,_}) -> {A,C} end},
F = sofs:partition_family(SetFun, S),
sofs:to_external(F).
[{{a,a},[{a,a,a,a}]},{{a,b},[{a,a,b,b},{a,b,b,b}]}]
```

`product(TupleOfSets) -> Relation`

Types:

- Relation = relation()
- TupleOfSets = tuple-of(set())

Returns the Cartesian product [page 373] of the non-empty tuple of sets TupleOfSets. If (x[1],...,x[n]) is an element of the n-ary relation Relation, then x[i] is drawn from element i of TupleOfSets.

```
1> S1 = sofs:set([a,b]),
S2 = sofs:set([1,2]),
S3 = sofs:set([x,y]),
P3 = sofs:product({S1,S2,S3}),
sofs:to_external(P3).
[{a,1,x},{a,1,y},{a,2,x},{a,2,y},{b,1,x},{b,1,y},{b,2,x},{b,2,y}]
```

`product(Set1, Set2) -> BinRel`

Types:

- BinRel = binary_relation()
- Set1 = Set2 = set()

Returns the Cartesian product [page 372] of Set1 and Set2.

```
1> S1 = sofs:set([1,2]),
S2 = sofs:set([a,b]),
R = sofs:product(S1, S2),
sofs:to_external(R).
[{1,a},{1,b},{2,a},{2,b}]
```

`product(S1,S2)` is equivalent to `product({S1,S2})`.

`projection(SetFun, Set1) -> Set2`

Types:

- SetFun = set_fun()
- Set1 = Set2 = set()

Returns the set created by substituting each element of Set1 by the result of applying SetFun to the element.

If SetFun is a number i>=1 and Set1 is a relation, then the returned set is the projection [page 373] of Set1 onto coordinate i.

```
1> S1 = sofs:from_term([{1,a},{2,b},{3,a}]),
S2 = sofs:projection(2, S1),
sofs:to_external(S2).
[a,b]
```

`range(BinRel) -> Set`

Types:

- BinRel = binary_relation()
- Set = set()

Returns the range [page 372] of the binary relation BinRel.

```
1> R = sofs:relation([{1,a},{1,b},{2,b},{2,c}]),
S = sofs:range(R),
sofs:to_external(S).
[a,b,c]
```

`relation(Tuples [, Type]) -> Relation`

Types:

- N = integer()
- Type = N | type()
- Relation = relation()
- Tuples = [tuple()]

Creates a relation [page 372]. `relation(R,T)` is equivalent to `from_term(R,T)`, if T is a type [page 374] and the result is a relation. If Type is an integer N, then `[{atom,...,atom}])`, where the size of the tuple is N, is used as type of the relation. If no type is explicitly given, the size of the first tuple of Tuples is used if there is such a tuple. `relation([])` is equivalent to `relation([],2)`.

`relation_to_family(BinRel) -> Family`

Types:

- Family = family()
- BinRel = binary_relation()

Returns the family [page 373] Family such that the index set is equal to the domain [page 372] of the binary relation BinRel, and Family[i] is the image [page 372] of the set of i under BinRel.

```
1> R = sofs:relation([{b,1},{c,2},{c,3}]),
F = sofs:relation_to_family(R),
sofs:to_external(F).
[{b,[1]},{c,[2,3]}]
```

`relative_product(TupleOfBinRels [, BinRel1]) -> BinRel2`

Types:

- TupleOfBinRels = tuple-of(BinRel)
- BinRel = BinRel1 = BinRel2 = binary_relation()

If TupleOfBinRels is a non-empty tuple {R[1],...,R[n]} of binary relations and BinRel1 is a binary relation, then BinRel2 is the relative product [page 373] of the ordered set (R[i],...,R[n]) and BinRel1.

If BinRel1 is omitted, the relation of equality between the elements of the Cartesian product [page 373] of the ranges of R[i], rangeR[1]...rangeR[n], is used instead (intuitively, nothing is "lost").

```
1> TR = sofs:relation([{1,a},{1,aa},{2,b}]),
R1 = sofs:relation([{1,u},{2,v},{3,c}]),
R2 = sofs:relative_product({TR, R1}),
sofs:to_external(R2).
[{1,{a,u}},{1,{aa,u}},{2,{b,v}}]
```

Note that `relative_product({R1},R2)` is different from `relative_product(R1,R2)`; the tuple of one element is not identified with the element itself.

`relative_product(BinRel1, BinRel2) -> BinRel3`

Types:

- BinRel1 = BinRel2 = BinRel3 = binary_relation()

Returns the relative product [page 372] of the binary relations BinRel1 and BinRel2.

`relative_product1(BinRel1, BinRel2) -> BinRel3`

Types:

- BinRel1 = BinRel2 = BinRel3 = binary_relation()

Returns the relative product [page 372] of the converse [page 372] of the binary relation BinRel1 and the binary relation BinRel2.

```
1> R1 = sofs:relation([{1,a},{1,aa},{2,b}]),
R2 = sofs:relation([{1,u},{2,v},{3,c}]),
R3 = sofs:relative_product1(R1, R2),
sofs:to_external(R3).
[{a,u},{aa,u},{b,v}]
```

`relative_product1(R1,R2)` is equivalent to `relative_product(converse(R1),R2)`.

`restriction(BinRel1, Set) -> BinRel2`

Types:

- BinRel1 = BinRel2 = binary_relation()
- Set = set()

Returns the restriction [page 372] of the binary relation BinRel1 to Set.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,c}]),
S = sofs:set([1,2,4]),
R2 = sofs:restriction(R1, S),
sofs:to_external(R2).
[{1,a},{2,b}]
```

`restriction(SetFun, Set1, Set2) -> Set3`

Types:

- SetFun = set_fun()
- Set1 = Set2 = Set3 = set()

Returns a subset of Set1 containing those elements that yield an element in Set2 as the result of applying SetFun.

```
1> S1 = sofs:relation([{1,a},{2,b},{3,c}]),
S2 = sofs:set([b,c,d]),
S3 = sofs:restriction(2, S1, S2),
sofs:to_external(S3).
[{2,b},{3,c}]
```

`set(Terms [, Type]) -> Set`

Types:

- Set = set()
- Terms = [term()]
- Type = type()

Creates an unordered set [page 374]. `set(L,T)` is equivalent to `from_term(L,T)`, if the result is an unordered set. If no type [page 374] is explicitly given, `[atom]` is used as type of the set.

`specification(Fun, Set1) -> Set2`

Types:

- Fun = spec_fun()
- Set1 = Set2 = set()

Returns the set containing every element of Set1 for which Fun returns `true`. If Fun is a tuple `{external,Fun2}`, Fun2 is applied to the external set [page 374] of each element, otherwise Fun is applied to each element.

```
1> R1 = sofs:relation([{a,1},{b,2}]),
R2 = sofs:relation([{x,1},{x,2},{y,3}]),
S1 = sofs:from_sets([R1,R2]),
S2 = sofs:specification({sofs,is_a_function}, S1),
sofs:to_external(S2).
[[{a,1},{b,2}]]
```

`strict_relation(BinRel1) -> BinRel2`

Types:

- BinRel1 = BinRel2 = binary_relation()

Returns the strict relation [page 373] corresponding to the binary relation BinRel1.

```
1> R1 = sofs:relation([{1,1},{1,2},{2,1},{2,2}]),
R2 = sofs:strict_relation(R1),
sofs:to_external(R2).
[{1,2},{2,1}]
```

`substitution(SetFun, Set1) -> Set2`

Types:

- SetFun = set_fun()
- Set1 = Set2 = set()

Returns a function, the domain of which is Set1. The value of an element of the domain is the result of applying SetFun to the element.

```
1> L = [{a,1},{b,2}].
[{a,1},{b,2}]
2> sofs:to_external(sofs:projection(1,sofs:relation(L))).
[a,b]
3> sofs:to_external(sofs:substitution(1,sofs:relation(L))).
[{{a,1},a},{{b,2},b}]
4> SetFun = {external, fun({A,_}=E) -> {E,A} end},
sofs:to_external(sofs:projection(SetFun,sofs:relation(L))).
[{{a,1},a},{{b,2},b}]
```

The relation of equality between the elements of {a,b,c}:

```
1> I = sofs:substitution(fun(A) -> A end, sofs:set([a,b,c])),
sofs:to_external(I).
[{a,a},{b,b},{c,c}]
```

Let SetOfSets be a set of sets and BinRel a binary relation. The function that maps each element Set of SetOfSets onto the image [page 372] of Set under BinRel is returned by this function:

```
images(SetOfSets, BinRel) ->
   Fun = fun(Set) -> sofs:image(BinRel, Set) end,
   sofs:substitution(Fun, SetOfSets).
```

Here might be the place to reveal something that was more or less stated before, namely that external unordered sets are represented as sorted lists. As a consequence, creating the image of a set under a relation R may traverse all elements of R (to that comes the sorting of results, the image). In `images/2`, BinRel will be traversed once for each element of SetOfSets, which may take too long. The following efficient function could be used instead under the assumption that the image of each element of SetOfSets under BinRel is non-empty:

```
images2(SetOfSets, BinRel) ->
   CR = sofs:canonical_relation(SetOfSets),
   R = sofs:relative_product1(CR, BinRel),
   sofs:relation_to_family(R).
```

`symdiff(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = set()

Returns the symmetric difference [page 372] (or the Boolean sum) of Set1 and Set2.

```
1> S1 = sofs:set([1,2,3]),
S2 = sofs:set([2,3,4]),
P = sofs:symdiff(S1, S2),
sofs:to_external(P).
[1,4]
```

`symmetric_partition(Set1, Set2) -> {Set3, Set4, Set5}`

Types:

- Set1 = Set2 = Set3 = Set4 = Set5 = set()

Returns a triple of sets: Set3 contains the elements of Set1 that do not belong to Set2; Set4 contains the elements of Set1 that belong to Set2; Set5 contains the elements of Set2 that do not belong to Set1.

`to_external(AnySet) -> ExternalSet`

Types:

- ExternalSet = external_set()
- AnySet = anyset()

Returns the external set [page 374] of an atomic, ordered or unordered set.

`to_sets(ASet) -> Sets`

Types:

- ASet = set() | ordset()
- Sets = tuple_of(AnySet) | [AnySet]

Returns the elements of the ordered set ASet as a tuple of sets, and the elements of the unordered set ASet as a sorted list of sets without duplicates.

`type(AnySet) -> Type`

Types:

- AnySet = anyset()
- Type = type()

Returns the type [page 374] of an atomic, ordered or unordered set.

`union(SetOfSets) -> Set`

Types:

- Set = set()
- SetOfSets = set_of_sets()

Returns the union [page 372] of the set of sets SetOfSets.

`union(Set1, Set2) -> Set3`

Types:

- Set1 = Set2 = Set3 = set()

Returns the union [page 372] of Set1 and Set2.

`union_of_family(Family) -> Set`

Types:

- Family = family()
- Set = set()

Returns the union of the family [page 373] Family.

```
1> F = sofs:family([{a,[0,2,4]},{b,[0,1,2]},{c,[2,3]}]),
S = sofs:union_of_family(F),
sofs:to_external(S).
[0,1,2,3,4]
```

`weak_relation(BinRel1) -> BinRel2`

Types:

- BinRel1 = BinRel2 = binary_relation()

Returns a subset S of the weak relation [page 373] W corresponding to the binary relation BinRel1. Let F be the field [page 372] of BinRel1. The subset S is defined so that x S y if x W y for some x in F and for some y in F.

```
1> R1 = sofs:relation([{1,1},{1,2},{3,1}]),
R2 = sofs:weak_relation(R1),
sofs:to_external(R2).
[{1,1},{1,2},{2,2},{3,1},{3,3}]
```

See Also

dict(3) [page 97], digraph(3) [page 102], orddict(3) [page 265], ordsets(3) [page 270], sets(3) [page 354]

string

Erlang Module

This module contains functions for string processing.

Exports

`len(String) -> Length`

Types:

- String = string()
- Length = integer()

Returns the number of characters in the string.

`equal(String1, String2) -> bool()`

Types:

- String1 = String2 = string()

Tests whether two strings are equal. Returns `true` if they are, otherwise `false`.

`concat(String1, String2) -> String3`

Types:

- String1 = String2 = String3 = string()

Concatenates two strings to form a new string. Returns the new string.

`chr(String, Character) -> Index`
`rchr(String, Character) -> Index`

Types:

- String = string()
- Character = char()
- Index = integer()

Returns the index of the first/last occurrence of `Character` in `String`. `0` is returned if `Character` does not occur.

`str(String, SubString) -> Index`
`rstr(String, SubString) -> Index`

Types:

- String = SubString = string()

- Index = integer()

Returns the position where the first/last occurrence of `SubString` begins in `String`. `0` is returned if `SubString` does not exist in `String`. For example:

```
> string:str(" Hello Hello World World ", "Hello World").
8
```

`span(String, Chars) -> Length`
`cspan(String, Chars) -> Length`

Types:

- String = Chars = string()
- Length = integer()

Returns the length of the maximum initial segment of String, which consists entirely of characters from (not from) Chars.

For example:

```
> string:span("\t    abcdef", " \t").
5
> string:cspan("\t    abcdef", " \t").
0
```

`substr(String, Start) -> SubString`
`substr(String, Start, Length) -> Substring`

Types:

- String = SubString = string()
- Start = Length = integer()

Returns a substring of `String`, starting at the position `Start`, and ending at the end of the string or at length `Length`.

For example:

```
> substr("Hello World", 4, 5).
"lo Wo"
```

`tokens(String, SeparatorList) -> Tokens`

Types:

- String = SeparatorList = string()
- Tokens = [string()]

Returns a list of tokens in `String`, separated by the characters in `SeparatorList`.

For example:

```
> tokens("abc defxxghix jkl", "x ").
["abc", "def", "ghi", "jkl"]
```

`join(StringList, Separator) -> String`

Types:

- StringList = [string()]
- Separator = string()

Returns a string with the elements of StringList seperated by the string in Seperator.

For example:

```
> join(["one", "two", "three"], ", ").
"one, two, three"
```

```
chars(Character, Number) -> String
chars(Character, Number, Tail) -> String
```

Types:

- Character = char()
- Number = integer()
- String = string()

Returns a string consisting of Number of characters Character. Optionally, the string can end with the string Tail.

```
copies(String, Number) -> Copies
```

Types:

- String = Copies = string()
- Number = integer()

Returns a string containing String repeated Number times.

```
words(String) -> Count
words(String, Character) -> Count
```

Types:

- String = string()
- Character = char()
- Count = integer()

Returns the number of words in String, separated by blanks or Character.

For example:

```
> words(" Hello old boy!", $o).
4
```

```
sub_word(String, Number) -> Word
sub_word(String, Number, Character) -> Word
```

Types:

- String = Word = string()
- Character = char()
- Number = integer()

Returns the word in position Number of String. Words are separated by blanks or Characters.

For example:

```
> string:sub_word(" Hello old boy !",3,$o).
"ld b"
```

```
strip(String) -> Stripped
strip(String, Direction) -> Stripped
strip(String, Direction, Character) -> Stripped
```

Types:

- String = Stripped = string()
- Direction = left | right | both
- Character = char()

Returns a string, where leading and/or trailing blanks or a number of `Character` have been removed. `Direction` can be `left`, `right`, or `both` and indicates from which direction blanks are to be removed. The function `strip/1` is equivalent to `strip(String, both)`.

For example:

```
> string:strip("...Hello.....", both, $.).
"Hello"
```

```
left(String, Number) -> Left
left(String, Number, Character) -> Left
```

Types:

- String = Left = string()
- Character = char
- Number = integer()

Returns the `String` with the length adjusted in accordance with `Number`. The left margin is fixed. If the `length(String)` < `Number`, `String` is padded with blanks or `Characters`.

For example:

```
> string:left("Hello",10,$.).
"Hello....."
```

```
right(String, Number) -> Right
right(String, Number, Character) -> Right
```

Types:

- String = Right = string()
- Character = char
- Number = integer()

Returns the `String` with the length adjusted in accordance with `Number`. The right margin is fixed. If the length of `(String)` < `Number`, `String` is padded with blanks or `Characters`.

For example:

```
> string:right("Hello", 10, $.).
".....Hello"
```

```
centre(String, Number) -> Centered
centre(String, Number, Character) -> Centered
```

Types:

- String = Centered = string()
- Character = char
- Number = integer()

Returns a string, where `String` is centred in the string and surrounded by blanks or characters. The resulting string will have the length `Number`.

```
sub_string(String, Start) -> SubString
sub_string(String, Start, Stop) -> SubString
```

Types:

- String = SubString = string()
- Start = Stop = integer()

Returns a substring of `String`, starting at the position `Start` to the end of the string, or to and including the `Stop` position.

For example:

```
sub_string("Hello World", 4, 8).
"lo Wo"
```

```
to_float(String) -> {Float,Rest} | {error,Reason}
```

Types:

- String = string()
- Float = float()
- Rest = string()
- Reason = no_float | not_a_list

Argument `String` is expected to start with a valid text represented float (the digits being ASCII values). Remaining characters in the string after the float are returned in `Rest`.

Example:

```
> {F1,Fs} = string:to_float("1.0-1.0e-1"),
> {F2,[]} = string:to_float(Fs),
> F1+F2.
0.9
> string:to_float("3/2=1.5").
{error,no_float}
> string:to_float("-1.5eX").
{-1.5,"eX"}
```

```
to_integer(String) -> {Int,Rest} | {error,Reason}
```

Types:

- String = string()
- Int = integer()
- Rest = string()
- Reason = no_integer | not_a_list

Argument String is expected to start with a valid text represented integer (the digits being ASCII values). Remaining characters in the string after the integer are returned in Rest.

Example:

```
> {I1,Is} = string:to_integer("33+22"),
> {I2,[]} = string:to_integer(Is),
> I1-I2.
11
> string:to_integer("0.5").
{0,".5"}
> string:to_integer("x=2").
{error,no_integer}
```

```
to_lower(String) -> Result
to_lower(Char) -> CharResult
to_upper(String) -> Result
to_upper(Char) -> CharResult
```

Types:

- String = Result = string()
- Char = CharResult = integer()

The given string or character is case-converted. Note that the supported character set is ISO/IEC 8859-1 (a.k.a. Latin 1), all values outside this set is unchanged

Notes

Some of the general string functions may seem to overlap each other. The reason for this is that this string package is the combination of two earlier packages and all the functions of both packages have been retained.

> **Note:**
> Any undocumented functions in string should not be used.

supervisor

Erlang Module

A behaviour module for implementing a supervisor, a process which supervises other processes called child processes. A child process can either be another supervisor or a worker process. Worker processes are normally implemented using one of the `gen_event`, `gen_fsm`, or `gen_server` behaviours. A supervisor implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. Supervisors are used to build an hierarchical process structure called a supervision tree, a nice way to structure a fault tolerant application. Refer to *OTP Design Principles* for more information.

A supervisor assumes the definition of which child processes to supervise to be located in a callback module exporting a pre-defined set of functions.

Unless otherwise stated, all functions in this module will fail if the specified supervisor does not exist or if bad arguments are given.

Supervision Principles

The supervisor is responsible for starting, stopping and monitoring its child processes. The basic idea of a supervisor is that it should keep its child processes alive by restarting them when necessary.

The children of a supervisor is defined as a list of *child specifications*. When the supervisor is started, the child processes are started in order from left to right according to this list. When the supervisor terminates, it first terminates its child processes in reversed start order, from right to left.

A supervisor can have one of the following *restart strategies*:

- `one_for_one` - if one child process terminates and should be restarted, only that child process is affected.
- `one_for_all` - if one child process terminates and should be restarted, all other child processes are terminated and then all child processes are restarted.
- `rest_for_one` - if one child process terminates and should be restarted, the 'rest' of the child processes – i.e. the child processes after the terminated child process in the start order – are terminated. Then the terminated child process and all child processes after it are restarted.
- `simple_one_for_one` - a simplified `one_for_one` supervisor, where all child processes are dynamically added instances of the same process type, i.e. running the same code.

  The functions `terminate_child/2`, `delete_child/2` and `restart_child/2` are invalid for `simple_one_for_one` supervisors and will return `{error,simple_one_for_one}` if the specified supervisor uses this restart strategy.

To prevent a supervisor from getting into an infinite loop of child process terminations and restarts, a *maximum restart frequency* is defined using two integer values `MaxR` and `MaxT`. If more than `MaxR` restarts occur within `MaxT` seconds, the supervisor terminates all child processes and then itself.

This is the type definition of a child specification:

```
child_spec() = {Id,StartFunc,Restart,Shutdown,Type,Modules}
 Id = term()
 StartFunc = {M,F,A}
  M = F = atom()
  A = [term()]
 Restart = permanent | transient | temporary
 Shutdown = brutal_kill | int()>=0 | infinity
 Type = worker | supervisor
 Modules = [Module] | dynamic
  Module = atom()
```

- `Id` is a name that is used to identify the child specification internally by the supervisor.
- `StartFunc` defines the function call used to start the child process. It should be a module-function-arguments tuple `{M,F,A}` used as `apply(M,F,A)`.

  The start function *must create and link to* the child process, and should return `{ok,Child}` or `{ok,Child,Info}` where `Child` is the pid of the child process and `Info` an arbitrary term which is ignored by the supervisor.

  The start function can also return `ignore` if the child process for some reason cannot be started, in which case the child specification will be kept by the supervisor but the non-existing child process will be ignored.

  If something goes wrong, the function may also return an error tuple `{error,Error}`.

  Note that the `start_link` functions of the different behaviour modules fulfill the above requirements.
- `Restart` defines when a terminated child process should be restarted. A `permanent` child process should always be restarted, a `temporary` child process should never be restarted and a `transient` child process should be restarted only if it terminates abnormally, i.e. with another exit reason than `normal`.
- `Shutdown` defines how a child process should be terminated. `brutal_kill` means the child process will be unconditionally terminated using `exit(Child,kill)`. An integer timeout value means that the supervisor will tell the child process to terminate by calling `exit(Child,shutdown)` and then wait for an exit signal with reason `shutdown` back from the child process. If no exit signal is received within the specified time, the child process is unconditionally terminated using `exit(Child,kill)`.
  If the child process is another supervisor, `Shutdown` should be set to `infinity` to give the subtree ample time to shutdown.
  Important note on simple-one-for-one supervisors: The dynamically created child processes of a simple-one-for-one supervisor are not explicitly killed, regardless of shutdown strategy, but are expected to terminate when the supervisor does (that is, when an exit signal from the parent process is received).

Note that all child processes implemented using the standard OTP behavior modules automatically adhere to the shutdown protocol.

- `Type` specifies if the child process is a supervisor or a worker.
- `Modules` is used by the release handler during code replacement to determine which processes are using a certain module. As a rule of thumb `Modules` should be a list with one element `[Module]`, where `Module` is the callback module, if the child process is a supervisor, gen_server or gen_fsm. If the child process is an event manager (gen_event) with a dynamic set of callback modules, `Modules` should be `dynamic`. See *OTP Design Principles* for more information about release handling.
- Internally, the supervisor also keeps track of the pid `Child` of the child process, or `undefined` if no pid exists.

Exports

```
start_link(Module, Args) -> Result
start_link(SupName, Module, Args) -> Result
```

Types:

- SupName = {local,Name} | {global,Name}
- Name = atom()
- Module = atom()
- Args = term()
- Result = {ok,Pid} | ignore | {error,Error}
- Pid = pid()
- Error = {already_started,Pid}} | shutdown | term()

Creates a supervisor process as part of a supervision tree. The function will, among other things, ensure that the supervisor is linked to the calling process (its supervisor).

The created supervisor process calls `Module:init/1` to find out about restart strategy, maximum restart frequency and child processes. To ensure a synchronized start-up procedure, `start_link/2,3` does not return until `Module:init/1` has returned and all child processes have been started.

If `SupName={local,Name}` the supervisor is registered locally as `Name` using `register/2`. If `SupName={global,Name}` the supervisor is registered globally as `Name` using `global:register_name/2`. If no name is provided, the supervisor is not registered.

`Module` is the name of the callback module.

`Args` is an arbitrary term which is passed as the argument to `Module:init/1`.

If the supervisor and its child processes are successfully created (i.e. if all child process start functions return `{ok,Child}`, `{ok,Child,Info}`, or `ignore`) the function returns `{ok,Pid}`, where `Pid` is the pid of the supervisor. If there already exists a process with the specified `SupName` the function returns `{error,{already_started,Pid}}`, where `Pid` is the pid of that process.

If `Module:init/1` returns `ignore`, this function returns `ignore` as well and the supervisor terminates with reason `normal`. If `Module:init/1` fails or returns an incorrect value, this function returns `{error,Term}` where `Term` is a term with information about the error, and the supervisor terminates with reason `Term`.

If any child process start function fails or returns an error tuple or an erroneous value, the function returns {error,shutdown} and the supervisor terminates all started child processes and then itself with reason shutdown.

start_child(SupRef, ChildSpec) -> Result

Types:

- SupRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- ChildSpec = child_spec() | [term()]
- Result = {ok,Child} | {ok,Child,Info} | {error,Error}
- Child = pid() | undefined
- Info = term()
- Error = already_present | {already_started,Child} | term()

Dynamically adds a child specification to the supervisor SupRef which starts the corresponding child process.

SupRef can be:

- the pid,
- Name, if the supervisor is locally registered,
- {Name,Node}, if the supervisor is locally registered at another node, or
- {global,Name}, if the supervisor is globally registered.

ChildSpec should be a valid child specification (unless the supervisor is a simple_one_for_one supervisor, see below). The child process will be started by using the start function as defined in the child specification.

If the case of a simple_one_for_one supervisor, the child specification defined in Module:init/1 will be used and ChildSpec should instead be an arbitrary list of terms List. The child process will then be started by appending List to the existing start function arguments, i.e. by calling apply(M, F, A++List) where {M,F,A} is the start function defined in the child specification.

If there already exists a child specification with the specified Id, ChildSpec is discarded and the function returns {error,already_present} or {error,{already_started,Child}}, depending on if the corresponding child process is running or not.

If the child process start function returns {ok,Child} or {ok,Child,Info}, the child specification and pid is added to the supervisor and the function returns the same value.

If the child process start function returns ignore, the child specification is added to the supervisor, the pid is set to undefined and the function returns {ok,undefined}.

If the child process start function returns an error tuple or an erroneous value, or if it fails, the child specification is discarded and the function returns {error,Error} where Error is a term containing information about the error and child specification.

terminate_child(SupRef, Id) -> Result

Types:

- SupRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Id = term()

- Result = ok | {error,Error}
- Error = not_found | simple_one_for_one

Tells the supervisor `SupRef` to terminate the child process corresponding to the child specification identified by `Id`. The process, if there is one, is terminated but the child specification is kept by the supervisor. This means that the child process may be later be restarted by the supervisor. The child process can also be restarted explicitly by calling `restart_child/2`. Use `delete_child/2` to remove the child specification.

See `start_child/2` for a description of `SupRef`.

If successful, the function returns `ok`. If there is no child specification with the specified `Id`, the function returns `{error,not_found}`.

`delete_child(SupRef, Id) -> Result`

Types:

- SupRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Id = term()
- Result = ok | {error,Error}
- Error = running | not_found | simple_one_for_one

Tells the supervisor `SupRef` to delete the child specification identified by `Id`. The corresponding child process must not be running, use `terminate_child/2` to terminate it.

See `start_child/2` for a description of `SupRef`.

If successful, the function returns `ok`. If the child specification identified by `Id` exists but the corresponding child process is running, the function returns `{error,running}`. If the child specification identified by `Id` does not exist, the function returns `{error,not_found}`.

`restart_child(SupRef, Id) -> Result`

Types:

- SupRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Id = term()
- Result = {ok,Child} | {ok,Child,Info} | {error,Error}
- Child = pid() | undefined
- Error = running | not_found | simple_one_for_one | term()

Tells the supervisor `SupRef` to restart a child process corresponding to the child specification identified by `Id`. The child specification must exist and the corresponding child process must not be running.

See `start_child/2` for a description of `SupRef`.

If the child specification identified by `Id` does not exist, the function returns `{error,not_found}`. If the child specification exists but the corresponding process is already running, the function returns `{error,running}`.

If the child process start function returns `{ok,Child}` or `{ok,Child,Info}`, the pid is added to the supervisor and the function returns the same value.

If the child process start function returns `ignore`, the pid remains set to `undefined` and the function returns {`ok`,`undefined`}.

If the child process start function returns an error tuple or an erroneous value, or if it fails, the function returns {`error`,`Error`} where `Error` is a term containing information about the error.

`which_children(SupRef) -> [{Id,Child,Type,Modules}]`

Types:

- SupRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Id = term() | undefined
- Child = pid() | undefined
- Type = worker | supervisor
- Modules = [Module] | dynamic
- Module = atom()

Returns a list with information about all child specifications and child processes belonging to the supervisor `SupRef`.

See `start_child/2` for a description of `SupRef`.

The information given for each child specification/process is:

- `Id` - as defined in the child specification or `undefined` in the case of a `simple_one_for_one` supervisor.
- `Child` - the pid of the corresponding child process, or `undefined` if there is no such process.
- `Type` - as defined in the child specification.
- `Modules` - as defined in the child specification.

`check_childspecs([ChildSpec]) -> Result`

Types:

- ChildSpec = child_spec()
- Result = ok | {error,Error}
- Error = term()

This function takes a list of child specification as argument and returns `ok` if all of them are syntactically correct, or {`error`,`Error`} otherwise.

CALLBACK FUNCTIONS

The following functions should be exported from a `supervisor` callback module.

Exports

`Module:init(Args) -> Result`

Types:

- Args = term()
- Result = {ok,{{RestartStrategy,MaxR,MaxT},[ChildSpec]}} | ignore
- RestartStrategy = one_for_all | one_for_one | rest_for_one | simple_one_for_one
- MaxR = MaxT = int()>=0
- ChildSpec = child_spec()

Whenever a supervisor is started using `supervisor:start_link/2,3`, this function is called by the new process to find out about restart strategy, maximum restart frequency and child specifications.

`Args` is the `Args` argument provided to the start function.

`RestartStrategy` is the restart strategy and `MaxR` and `MaxT` defines the maximum restart frequency of the supervisor. `[ChildSpec]` is a list of valid child specifications defining which child processes the supervisor should start and monitor. See the discussion about Supervision Principles above.

Note that when the restart strategy is `simple_one_for_one`, the list of child specifications must be a list with one child specification only. (The `Id` is ignored). No child process is then started during the initialization phase, but all children are assumed to be started dynamically using `supervisor:start_child/2`.

The function may also return `ignore`.

SEE ALSO

gen_event(3) [page 188], gen_fsm(3) [page 198], gen_server(3) [page 209], sys(3) [page 411]

supervisor_bridge

Erlang Module

A behaviour module for implementing a supervisor_bridge, a process which connects a subsystem not designed according to the OTP design principles to a supervision tree. The supervisor_bridge sits between a supervisor and the subsystem. It behaves like a real supervisor to its own supervisor, but has a different interface than a real supervisor to the subsystem. Refer to *OTP Design Principles* for more information.

A supervisor_bridge assumes the functions for starting and stopping the subsystem to be located in a callback module exporting a pre-defined set of functions.

The `sys` module can be used for debugging a supervisor_bridge.

Unless otherwise stated, all functions in this module will fail if the specified supervisor_bridge does not exist or if bad arguments are given.

Exports

```
start_link(Module, Args) -> Result
start_link(SupBridgeName, Module, Args) -> Result
```

Types:

- SupBridgeName = {local,Name} | {global,Name}
- Name = atom()
- Module = atom()
- Args = term()
- Result = {ok,Pid} | ignore | {error,Error}
- Pid = pid()
- Error = {already_started,Pid} | term()

Creates a supervisor_bridge process, linked to the calling process, which calls `Module:init/1` to start the subsystem. To ensure a synchronized start-up procedure, this function does not return until `Module:init/1` has returned.

If `SupBridgeName={local,Name}` the supervisor_bridge is registered locally as `Name` using `register/2`. If `SupBridgeName={global,Name}` the supervisor_bridge is registered globally as `Name` using `global:register_name/2`. If no name is provided, the supervisor_bridge is not registered. If there already exists a process with the specified `SupBridgeName` the function returns `{error,{already_started,Pid}}`, where `Pid` is the pid of that process.

`Module` is the name of the callback module.

`Args` is an arbitrary term which is passed as the argument to `Module:init/1`.

If the supervisor_bridge and the subsystem are successfully started the function returns `{ok,Pid}`, where `Pid` is is the pid of the supervisor_bridge.

If `Module:init/1` returns `ignore`, this function returns `ignore` as well and the supervisor_bridge terminates with reason `normal`. If `Module:init/1` fails or returns an error tuple or an incorrect value, this function returns `{error,Term}` where `Term` is a term with information about the error, and the supervisor_bridge terminates with reason `Term`.

CALLBACK FUNCTIONS

The following functions should be exported from a `supervisor_bridge` callback module.

Exports

`Module:init(Args) -> Result`

Types:

- Args = term()
- Result = {ok,Pid,State} | ignore | {error,Error}
- Pid = pid()
- State = term()
- Error = term()

Whenever a supervisor_bridge is started using `supervisor_bridge:start_link/2,3`, this function is called by the new process to start the subsystem and initialize.

`Args` is the `Args` argument provided to the start function.

The function should return `{ok,Pid,State}` where `Pid` is the pid of the main process in the subsystem and `State` is any term.

If later `Pid` terminates with a reason `Reason`, the supervisor bridge will terminate with reason `Reason` as well. If later the supervisor_bridge is stopped by its supervisor with reason `Reason`, it will call `Module:terminate(Reason,State)` to terminate.

If something goes wrong during the initialization the function should return `{error,Error}` where `Error` is any term, or `ignore`.

`Module:terminate(Reason, State)`

Types:

- Reason = shutdown | term()
- State = term()

This function is called by the supervisor_bridge when it is about to terminate. It should be the opposite of `Module:init/1` and stop the subsystem and do any necessary cleaning up. The return value is ignored.

`Reason` is `shutdown` if the supervisor_bridge is terminated by its supervisor. If the supervisor_bridge terminates because a a linked process (apart from the main process of the subsystem) has terminated with reason `Term`, `Reason` will be `Term`.

`State` is taken from the return value of `Module:init/1`.

SEE ALSO

supervisor(3) [page 401], sys(3) [page 411]

sys

Erlang Module

This module contains functions for sending system messages used by programs, and messaged used for debugging purposes.

Functions used for implementation of processes should also understand system messages such as debugging messages and code change. These functions must be used to implement the use of system messages for a process; either directly, or through standard behaviours, such as `gen_server`.

The following types are used in the functions defined below:

- `Name = pid() | atom() | {global, atom()}`
- `Timeout = int() >= 0 | infinity`
- `system_event() = {in, Msg} | {in, Msg, From} | {out, Msg, To} | term()`

The default timeout is 5000 ms, unless otherwise specified. The `timeout` defines the time period to wait for the process to respond to a request. If the process does not respond, the function evaluates `exit({timeout, {M, F, A}})`.

The functions make reference to a debug structure. The debug structure is a list of `dbg_opt()`. `dbg_opt()` is an internal data type used by the `handle_system_msg/6` function. No debugging is performed if it is an empty list.

System Messages

Processes which are not implemented as one of the standard behaviours must still understand system messages. There are three different messages which must be understood:

- Plain system messages. These are received as `{system, From, Msg}`. The content and meaning of this message are not interpreted by the receiving process module. When a system message has been received, the function `sys:handle_system_msg/6` is called in order to handle the request.
- Shutdown messages. If the process traps exits, it must be able to handle an shut-down request from its parent, the supervisor. The message `{'EXIT', Parent, Reason}` from the parent is an order to terminate. The process must terminate when this message is received, normally with the same `Reason` as `Parent`.

- There is one more message which the process must understand if the modules used to implement the process change dynamically during runtime. An example of such a process is the `gen_event` processes. This message is {`get_modules, From`}. The reply to this message is `From ! {modules, Modules}`, where `Modules` is a list of the currently active modules in the process.

  This message is used by the release handler to find which processes execute a certain module. The process may at a later time be suspended and ordered to perform a code change for one of its modules.

System Events

When debugging a process with the functions of this module, the process generates *system_events* which are then treated in the debug function. For example, `trace` formats the system events to the tty.

There are three predefined system events which are used when a process receives or sends a message. The process can also define its own system events. It is always up to the process itself to format these events.

Exports

`log(Name,Flag)`
`log(Name,Flag,Timeout) -> ok | {ok, [system_event()]}`

Types:

- Flag = true | {true, N} | false | get | print
- N = integer() > 0

Turns the logging of system events On or Off. If On, a maximum of `N` events are kept in the debug structure (the default is 10). If `Flag` is `get`, a list of all logged events is returned. If `Flag` is `print`, the logged events are printed to `standard_io`. The events are formatted with a function that is defined by the process that generated the event (with a call to `sys:handle_debug/4`).

`log_to_file(Name,Flag)`
`log_to_file(Name,Flag,Timeout) -> ok | {error, open_file}`

Types:

- Flag = FileName | false
- FileName = string()

Enables or disables the logging of all system events in textual format to the file. The events are formatted with a function that is defined by the process that generated the event (with a call to `sys:handle_debug/4`).

`statistics(Name,Flag)`
`statistics(Name,Flag,Timeout) -> ok | {ok, Statistics}`

Types:

- Flag = true | false | get

- Statistics = [{start_time, {Date1, Time1}}, {current_time, {Date, Time2}}, {reductions, integer()}, {messages_in, integer()}, {messages_out, integer()}]
- Date1 = Date2 = {Year, Month, Day}
- Time1 = Time2 = {Hour, Min, Sec}

Enables or disables the collection of statistics. If `Flag` is `get`, the statistical collection is returned.

```
trace(Name,Flag)
trace(Name,Flag,Timeout) -> void()
```

Types:

- Flag = boolean()

Prints all system events on `standard_io`. The events are formatted with a function that is defined by the process that generated the event (with a call to `sys:handle_debug/4`).

```
no_debug(Name)
no_debug(Name,Timeout) -> void()
```

Turns off all debugging for the process. This includes functions that have been installed explicitly with the `install` function, for example triggers.

```
suspend(Name)
suspend(Name,Timeout) -> void()
```

Suspends the process. When the process is suspended, it will only respond to other system messages, but not other messages.

```
resume(Name)
resume(Name,Timeout) -> void()
```

Resumes a suspended process.

```
change_code(Name, Module, OldVsn, Extra)
change_code(Name, Module, OldVsn, Extra, Timeout) -> ok | {error, Reason}
```

Types:

- OldVsn = undefined | term()
- Module = atom()
- Extra = term()

Tells the process to change code. The process must be suspended to handle this message. The `Extra` argument is reserved for each process to use as its own. The function `Mod:system_code_change/4` is called. `OldVsn` is the old version of the `Module`.

```
get_status(Name)
get_status(Name,Timeout) -> {status, Pid, {module, Mod}, [PDict, SysState, Parent,
Dbg, Misc]}
```

Types:

- PDict = [{Key, Value}]
- SysState = running | suspended
- Parent = pid()

- Dbg = [dbg_opt()]
- Misc = term()

Gets the status of the process.

```
install(Name,{Func,FuncState})
install(Name,{Func,FuncState},Timeout)
```

Types:

- Func = dbg_fun()
- dbg_fun() = fun(FuncState, Event, ProcState) -> done | NewFuncState
- FuncState = term()
- Event = system_event()
- ProcState = term()
- NewFuncState = term()

This function makes it possible to install other debug functions than the ones defined above. An example of such a function is a trigger, a function that waits for some special event and performs some action when the event is generated. This could, for example, be turning on low level tracing.

`Func` is called whenever a system event is generated. This function should return `done`, or a new func state. In the first case, the function is removed. It is removed if the function fails.

```
remove(Name,Func)
remove(Name,Func,Timeout) -> void()
```

Types:

- Func = dbg_fun()

Removes a previously installed debug function from the process. `Func` must be the same as previously installed.

Process Implementation Functions

The following functions are used when implementing a special process. This is an ordinary process which does not use a standard behaviour, but a process which understands the standard system messages.

Exports

`debug_options(Options) -> [dbg_opt()]`

Types:

- Options = [Opt]
- Opt = trace | log | statistics | {log_to_file, FileName} | {install, {Func, FuncState}}
- Func = dbg_fun()
- FuncState = term()

This function can be used by a process that initiates a debug structure from a list of options. The values of the `Opt` argument are the same as the corresponding functions.

`get_debug(Item,Debug,Default) -> term()`

Types:

- Item = log | statistics
- Debug = [dbg_opt()]
- Default = term()

This function gets the data associated with a debug option. `Default` is returned if the `Item` is not found. Can be used by the process to retrieve debug data for printing before it terminates.

`handle_debug([dbg_opt()],FormFunc,Extra,Event) -> [dbg_opt()]`

Types:

- FormFunc = dbg_fun()
- Extra = term()
- Event = system_event()

This function is called by a process when it generates a system event. `FormFunc` is a formatting function which is called as `FormFunc(Device, Event, Extra)` in order to print the events, which is necessary if tracing is activated. `Extra` is any extra information which the process needs in the format function, for example the name of the process.

`handle_system_msg(Msg,From,Parent,Module,Debug,Misc)`

Types:

- Msg = term()
- From = pid()
- Parent = pid()
- Module = atom()
- Debug = [dbg_opt()]
- Misc = term()

This function is used by a process module that wishes to take care of system messages. The process receives a {system, From, Msg} message and passes the Msg and From to this function.

This function *never* returns. It calls the function Module:system_continue(Parent, NDebug, Misc) where the process continues the execution, or Module:system_terminate(Reason, Parent, Debug, Misc) if the process should terminate. The Module must export system_continue/3, system_terminate/4, and system_code_change/4 (see below).

The Misc argument can be used to save internal data in a process, for example its state. It is sent to Module:system_continue/3 or Module:system_terminate/4

print_log(Debug) -> void()

Types:

- Debug = [dbg_opt()]

Prints the logged system events in the debug structure using FormFunc as defined when the event was generated by a call to handle_debug/4.

Mod:system_continue(Parent, Debug, Misc)

Types:

- Parent = pid()
- Debug = [dbg_opt()]
- Misc = term()

This function is called from sys:handle_system_msg/6 when the process should continue its execution (for example after it has been suspended). This function never returns.

Mod:system_terminate(Reason, Parent, Debug, Misc)

Types:

- Reason = term()
- Parent = pid()
- Debug = [dbg_opt()]
- Misc = term()

This function is called from sys:handle_system_msg/6 when the process should terminate. For example, this function is called when the process is suspended and its parent orders shut-down. It gives the process a chance to do a clean-up. This function never returns.

Mod:system_code_change(Misc, Module, OldVsn, Extra) -> {ok, NMisc}

Types:

- Misc = term()
- OldVsn = undefined | term()
- Module = atom()
- Extra = term()
- NMisc = term()

Called from `sys:handle_system_msg/6` when the process should perform a code change. The code change is used when the internal data structure has changed. This function converts the `Misc` argument to the new data structure. `OldVsn` is the *vsn* attribute of the old version of the `Module`. If no such attribute was defined, the atom `undefined` is sent.

timer

Erlang Module

This module provides useful functions related to time. Unless otherwise stated, time is always measured in `milliseconds`. All timer functions return immediately, regardless of work carried out by another process.

Successful evaluations of the timer functions yield return values containing a timer reference, denoted `TRef` below. By using `cancel/1`, the returned reference can be used to cancel any requested action. A `TRef` is an Erlang term, the contents of which must not be altered.

The timeouts are not exact, but should be `at least` as long as requested.

Exports

`start() -> ok`

Starts the timer server. Normally, the server does not need to be started explicitly. It is started dynamically if it is needed. This is useful during development, but in a target system the server should be started explicitly. Use configuration parameters for `kernel` for this.

`apply_after(Time, Module, Function, Arguments) -> {ok, Tref} | {error, Reason}`

Types:

- Time = integer() in Milliseconds
- Module = Function = atom()
- Arguments = [term()]

Evaluates `apply(M, F, A)` after `Time` amount of time has elapsed. Returns {`ok`, `TRef`}, or {`error, Reason`}.

`send_after(Time, Pid, Message) -> {ok, TRef} | {error,Reason}`
`send_after(Time, Message) -> {ok, TRef} | {error,Reason}`

Types:

- Time = integer() in Milliseconds
- Pid = pid() | atom()
- Message = term()
- Result = {ok, TRef} | {error, Reason}

`send_after/3` Evaluates `Pid ! Message` after `Time` amount of time has elapsed. (`Pid` can also be an atom of a registered name.) Returns {`ok, TRef`}, or {`error, Reason`}.

`send_after/2` Same as `send_after(Time, self(), Message)`.

```
exit_after(Time, Pid, Reason1) -> {ok, TRef} | {error,Reason2}
exit_after(Time, Reason1) -> {ok, TRef} | {error,Reason2}
kill_after(Time, Pid)-> {ok, TRef} | {error,Reason2}
kill_after(Time) -> {ok, TRef} | {error,Reason2}
```

Types:

- Time = integer() in milliseconds
- Pid = pid() | atom()
- Reason1 = Reason2 = term()

`exit_after/3` Send an exit signal with reason `Reason1` to Pid `Pid`. Returns {`ok`, `TRef`}, or {`error`, `Reason2`}.

`exit_after/2` Same as `exit_after(Time, self(), Reason1)`.

`kill_after/2` Same as `exit_after(Time, Pid, kill)`.

`kill_after/1` Same as `exit_after(Time, self(), kill)`.

```
apply_interval(Time, Module, Function, Arguments) -> {ok, TRef} | {error, Reason}
```

Types:

- Time = integer() in milliseconds
- Module = Function = atom()
- Arguments = [term()]

Evaluates `apply(Module, Function, Arguments)` repeatedly at intervals of `Time`. Returns {`ok`, `TRef`}, or {`error`, `Reason`}.

```
send_interval(Time, Pid, Message) -> {ok, TRef} | {error, Reason}
send_interval(Time, Message) -> {ok, TRef} | {error, Reason}
```

Types:

- Time = integer() in milliseconds
- Pid = pid() | atom()
- Message = term()
- Reason = term()

`send_interval/3` Evaluates `Pid ! Message` repeatedly after `Time` amount of time has elapsed. (`Pid` can also be an atom of a registered name.) Returns {`ok`, `TRef`} or {`error`, `Reason`}.

`send_interval/2` Same as `send_interval(Time, self(), Message)`.

```
cancel(TRef) -> {ok, cancel} | {error, Reason}
```

Cancels a previously requested timeout. `TRef` is a unique timer reference returned by the timer function in question. Returns {`ok`, `cancel`}, or {`error`, `Reason`} when `TRef` is not a timer reference.

```
sleep(Time) -> ok
```

Types:

- Time = integer() in milliseconds

Suspends the process calling this function for `Time` amount of milliseconds and then returns `ok`. Naturally, this function does *not* return immediately.

`tc(Module, Function, Arguments) -> {Time, Value}`

Types:

- Module = Function = atom()
- Arguments = [term()]
- Time = integer() in microseconds
- Value = term()

Evaluates `apply(Module, Function, Arguments)` and measures the elapsed real time. Returns `{Time, Value}`, where `Time` is the elapsed real time in *microseconds*, and `Value` is what is returned from the apply.

`now_diff(T2, T1) -> Tdiff`

Types:

- T1 = T2 = {MegaSecs, Secs, MicroSecs}
- Tdiff = MegaSecs = Secs = MicroSecs = integer()

Calculates the time difference `Tdiff = T2 - T1` in *microseconds*, where `T1` and `T2` probably are timestamp tuples returned from `erlang:now/0`.

`seconds(Seconds) -> Milliseconds`

Returns the number of milliseconds in `Seconds`.

`minutes(Minutes) -> Milliseconds`

Return the number of milliseconds in `Minutes`.

`hours(Hours) -> Milliseconds`

Returns the number of milliseconds in `Hours`.

`hms(Hours, Minutes, Seconds) -> Milliseconds`

Returns the number of milliseconds in `Hours + Minutes + Seconds`.

Examples

This example illustrates how to print out “Hello World!” in 5 seconds:

```
1> timer:apply_after(5000, io, format, ["~nHello World!~n", []]).
{ok,TRef}
Hello World!
```

The following coding example illustrates a process which performs a certain action and if this action is not completed within a certain limit, then the process is killed.

```
Pid = spawn(mod, fun, [foo, bar]),
%% If pid is not finished in 10 seconds, kill him
{ok, R} = timer:kill_after(timer:seconds(10), Pid),
...
%% We change our mind...
timer:cancel(R),
...
```

WARNING

A timer can always be removed by calling `cancel/1`.

An interval timer, i.e. a timer created by evaluating any of the functions `apply_interval/4`, `send_interval/3`, and `send_interval/2`, is linked to the process towards which the timer performs its task.

A one-shot timer, i.e. a timer created by evaluating any of the functions `apply_after/4`, `send_after/3`, `send_after/2`, `exit_after/3`, `exit_after/2`, `kill_after/2`, and `kill_after/1` is not linked to any process. Hence, such a timer is removed only when it reaches its timeout, or if it is explicitly removed by a call to `cancel/1`.

win32reg

Erlang Module

win32reg provides read and write access to the registry on Windows. It is essentially a port driver wrapped around the Win32 API calls for accessing the registry.

The registry is a hierarchical database, used to store various system and software information in Windows. It is available in Windows 95 and Windows NT. It contains installation data, and is updated by installers and system programs. The Erlang installer updates the registry by adding data that Erlang needs.

The registry contains keys and values. Keys are like the directories in a file system, they form a hierarchy. Values are like files, they have a name and a value, and also a type.

Paths to keys are left to right, with sub-keys to the right and backslash between keys. (Remember that backslashes must be doubled in Erlang strings.) Case is preserved but not significant. Example:
"\\hkey_local_machine\\software\\Ericsson\\Erlang\\5.0" is the key for the installation data for the latest Erlang release.

There are six entry points in the Windows registry, top level keys. They can be abbreviated in the win32reg module as:

Abbrev.	Registry key
hkcr	HKEY_CLASSES_ROOT
current_user	HKEY_CURRENT_USER
hkcu	HKEY_CURRENT_USER
local_machine	HKEY_LOCAL_MACHINE
hklm	HKEY_LOCAL_MACHINE
users	HKEY_USERS
hku	HKEY_USERS
current_config	HKEY_CURRENT_CONFIG
hkcc	HKEY_CURRENT_CONFIG
dyn_data	HKEY_DYN_DATA
hkdd	HKEY_DYN_DATA

The key above could be written as "\\hklm\\software\\ericsson\\erlang\\5.0".

The win32reg module uses a current key. It works much like the current directory. From the current key, values can be fetched, sub-keys can be listed, and so on.

Under a key, any number of named values can be stored. They have name, and types, and data.

Currently, the win32reg module supports storing only the following types: REG_DWORD, which is an integer, REG_SZ which is a string and REG_BINARY which is a binary. Other types can be read, and will be returned as binaries.

There is also a “default” value, which has the empty string as name. It is read and written with the atom default instead of the name.

Some registry values are stored as strings with references to environment variables, e.g. `"%SystemRoot%Windows"`. `SystemRoot` is an environment variable, and should be replaced with its value. A function `expand/1` is provided, so that environment variables surrounded in % can be expanded to their values.

For additional information on the Windows registry consult the Win32 Programmer's Reference.

Exports

`change_key(RegHandle, Key) -> ReturnValue`

Types:

- RegHandle = term()
- Key = string()

Changes the current key to another key. Works like cd. The key can be specified as a relative path or as an absolute path, starting with \.

`change_key_create(RegHandle, Key) -> ReturnValue`

Types:

- RegHandle = term()
- Key = string()

Creates a key, or just changes to it, if it is already there. Works like a combination of `mkdir` and `cd`. Calls the Win32 API function `RegCreateKeyEx()`.

The registry must have been opened in write-mode.

`close(RegHandle)-> ReturnValue`

Types:

- RegHandle = term()

Closes the registry. After that, the `RegHandle` cannot be used.

`current_key(RegHandle) -> ReturnValue`

Types:

- RegHandle = term()
- ReturnValue = {ok, string()}

Returns the path to the current key. This is the equivalent of `pwd`.

Note that the current key is stored in the driver, and might be invalid (e.g. if the key has been removed).

`delete_key(RegHandle) -> ReturnValue`

Types:

- RegHandle = term()
- ReturnValue = ok | {error, ErrorId}

Deletes the current key, if it is valid. Calls the Win32 API function `RegDeleteKey()`. Note that this call does not change the current key, (unlike `change_key_create/2`.) This means that after the call, the current key is invalid.

`delete_value(RegHandle, Name) -> ReturnValue`

Types:

- RegHandle = term()
- ReturnValue = ok | {error, ErrorId}

Deletes a named value on the current key. The atom `default` is used for the the default value.

The registry must have been opened in write-mode.

`expand(String) -> ExpandedString`

Types:

- String = string()
- ExpandedString = string()

Expands a string containing environment variables between percent characters. Anything between two % is taken for a environment variable, and is replaced by the value. Two consecutive % is replaced by one %.

A variable name that is not in the environment, will result in an error.

`format_error(ErrorId) -> ErrorString`

Types:

- ErrorId = atom()
- ErrorString = string()

Convert an POSIX errorcode to a string (by calling `erl_posix_msg:message`).

`open(OpenModeList)-> ReturnValue`

Types:

- OpenModeList = [OpenMode]
- OpenMode = read | write

Opens the registry for reading or writing. The current key will be the root (`HKEY_CLASSES_ROOT`). The `read` flag in the mode list can be omitted.

Use `change_key/2` with an absolute path after `open`.

`set_value(RegHandle, Name, Value) -> ReturnValue`

Types:

- Name = string() | default
- Value = string() | integer() | binary()

Sets the named (or default) value to value. Calls the Win32 API function `RegSetValueEx()`. The value can be of three types, and the corresponding registry type will be used. Currently the types supported are: `REG_DWORD` for integers, `REG_SZ` for strings and `REG_BINARY` for binaries. Other types cannot currently be added or changed.

The registry must have been opened in write-mode.

`sub_keys(RegHandle) -> ReturnValue`

Types:

- ReturnValue = {ok, SubKeys} | {error, ErrorId}
- SubKeys = [SubKey]
- SubKey = string()

Returns a list of subkeys to the current key. Calls the Win32 API function `EnumRegKeysEx()`.

Avoid calling this on the root keys, it can be slow.

`value(RegHandle, Name) -> ReturnValue`

Types:

- Name = string() | default
- ReturnValue = {ok, Value}
- Value = string() | integer() | binary()

Retrieves the named value (or default) on the current key. Registry values of type `REG_SZ`, are returned as strings. Type `REG_DWORD` values are returned as integers. All other types are returned as binaries.

`values(RegHandle) -> ReturnValue`

Types:

- ReturnValue = {ok, ValuePairs}
- ValuePairs = [ValuePair]
- ValuePair = {Name, Value}
- Name = string | default
- Value = string() | integer() | binary()

Retrieves a list of all values on the current key. The values have types corresponding to the registry types, see `value`. Calls the Win32 API function `EnumRegValuesEx()`.

SEE ALSO

Win32 Programmer's Reference (from Microsoft)

`erl_posix_msg`

The Windows 95 Registry (book from O'Reilly)

zip

Erlang Module

The `zip` module archives and extract files to and from a zip archive. The zip format is specified by the "ZIP Appnote.txt" file available on PKWare's website www.pkware.com.

The zip module supports zip archive versions up to 6.1. However, password-protection and Zip64 is not supported.

By convention, the name of a zip file should end in "`.zip`". To abide to the convention, you'll need to add "`.zip`" yourself to the name.

Zip archives are created with the zip/2 [page 427] or the zip/3 [page 427] function. (They are also available as `create`, to resemble the `erl_tar` module.)

To extract files from a zip archive, use the unzip/1 [page 428] or the unzip/2 [page 428] function. (They are also available as `extract`.)

To return a list of the files in a zip archive, use the list_dir/1 [page 429] or the list_dir/2 [page 429] function. (They are also available as `table`.)

To print a list of files to the Erlang shell, use either the t/1 [page 429] or tt/1 [page 430] function.

In some cases, it is desirable to open a zip archive, and to unzip files from it file by file, without having to reopen the archive. The functions zip_open [page 430], zip_get [page 430], zip_list_dir [page 430] and zip_close [page 430] do this.

LIMITATIONS

Zip64 archives are not currently supported.

Password-protected and encrypted archives are not currently supported

Only the DEFLATE (zlib-compression) and the STORE (uncompressed data) zip methods are supported.

The size of the archive is limited to 2 G-byte (32 bits).

Comments for indivudal files is not supported when creating zip archives. The zip archive comment for the whole zip archive is supported.

There is currently no support for altering an existing zip archive. To add or remove a file from an archive, the whole archive must be recreated.

DATA TYPES

`zip_file()`

The record `zip_file` contains the following fields.

`name = string()` the name of the file

`info = file_info()` file info as in [file:read_file_info/1]

`comment = string()` the comment for the file in the zip archive

`offset = integer()` the offset of the file in the zip archive (used internally)

`comp_size = integer()` the compressed size of the file (the uncompressed size is found in `info`)

`zip_comment`

The record `zip_comment` just contains the archive comment for a zip archive

`comment = string()` the comment for the zip archive

Exports

```
zip(Name, FileList) -> RetValue
zip(Name, FileList, Options) -> RetValue
create(Name, FileList) -> RetValue
create(Name, FileList, Options) -> RetValue
```

Types:

- Name = filename()
- FileList = [FileSpec]
- FileSpec = filename() | {filename(), binary()}
- Options = [Option]
- Option = memory | cooked | verbose | {comment, Comment} | {cwd, CWD}
- Comment = CWD = string()
- RetValue = {ok, Name} | {ok, {Name, binary()}} | {error, Reason}
- Reason = term()

The `zip` function creates a zip archive containing the files specified in `FileList`.

As synonyms, the functions `create/2` and `create/3` are provided, to make it resemble the `erl_tar` module.

The file-list is a list of files, with paths relative to the current directory, they will be stored with this path in the archive. Files may also be specified with data in binaries, to create an archive directly from data.

Files will be compressed using the DEFLATE compression, as described in the Appnote.txt file. However, files will be stored without compression if they already are compressed. The `zip/2` and `zip/3` checks the file extension to see whether the file should be stored without compression. Files with the following extensions are not compressed: `.Z`, `.zip`, `.zoo`, `.arc`, `.lzh`, `.arj`.

The following options are available:

`cooked` By default, the `open/2` function will open the zip file in `raw` mode, which is faster but does not allow a remote (erlang) file server to be used. Adding `cooked` to the mode list will override the default and open the zip file without the `raw` option. The same goes for the files added.

`verbose` Print an informational message about each file being added.

`memory` The output will not be to a file, but instead as a tuple `{FileName, binary()}`. The binary will be a full zip archive with header, and can be extracted with for instance `unzip/2`.

`{comment, Comment}` Add a comment to the zip-archive.

`{cwd, CWD}` Use the given directory as current directory, it will be prepended to file names when adding them, although it will not be in the zip-archive. (Acting like a file:set_cwd/1, but without changing the global cwd property.)

```
unzip(Archive) -> RetValue
unzip(Archive, Options) -> RetValue
extract(Archive) -> RetValue
extract(Archive, Options) -> RetValue
```

Types:

- Archive = filename() | binary()
- Options = [Option]
- Option = {file_list, FileList} | keep_old_files | verbose | memory | {file_filter, FileFilter} | {cwd, CWD}
- FileList = [filename()]
- FileBinList = [{filename(),binary()}]
- FileFilter = fun(ZipFile) -> true | false
- CWD = string()
- ZipFile = zip_file()
- RetValue = {ok,FileList} | {ok,FileBinList} | {error, Reason} | {error, {Name, Reason}}
- Reason = term()

The `unzip/1` function extracts all files from a zip archive. The `unzip/2` function provides options to extract some files, and more.

If the `Archive` argument is given as a binary, the contents of the binary is assumed to be a zip archive, otherwise it should be a filename.

The following options are available:

`{file_list, FileList}` By default, all files will be extracted from the zip archive. With the `{file_list,FileList}` option, the `unzip/2` function will only extract the files whose names are included in `FileList`. The full paths, including the names of all sub directories within the zip archive, must be specified.

`cooked` By default, the `open/2` function will open the zip file in `raw` mode, which is faster but does not allow a remote (erlang) file server to be used. Adding `cooked` to the mode list will override the default and open zip file without the `raw` option. The same goes for the files extracted.

`keep_old_files` By default, all existing files with the same name as file in the zip archive will be overwritten. With the `keep_old_files` option, the `unzip/2` function will not overwrite any existing files. Not that even with the `memory` option given, which means that no files will be overwritten, files existing will be excluded from the result.

`verbose` Print an informational message as each file is being extracted.

`memory` Instead of extracting to the current directory, the `memory` option will give the result as a list of tuples `{Filename, Binary}`, where `Binary` is a binary containing the extracted data of the file named `Filename` in the zip archive.

`{cwd, CWD}` Use the given directory as current directory, it will be prepended to file names when extracting them from the zip-archive. (Acting like a file:set_cwd/1, but without changing the global cwd property.)

```
list_dir(Archive) -> RetValue
list_dir(Archive, Options)
table(Archive) -> RetValue
table(Archive, Options)
```

Types:

- Archive = filename() | binary()
- RetValue = {ok, [Comment, Files]} | {error, Reason}
- Comment = zip_comment()
- Files = [zip_file()]
- Options = [Option]
- Option = cooked
- Reason = term()

The `list_dir/1` function retrieves the names of all files in the zip archive `Archive`. The `list_dir/2` function provides options.

As synonyms, the functions `table/2` and `table/3` are provided, to make it resemble the `erl_tar` module.

The result value is the tuple `{ok, List}`, where `List` contains the zip archive comment as the first element.

The following options are available:

`cooked` By default, the `open/2` function will open the zip file in `raw` mode, which is faster but does not allow a remote (erlang) file server to be used. Adding `cooked` to the mode list will override the default and open zip file without the `raw` option.

```
t(Archive)
```

Types:

- Archive = filename() | binary() | ZipHandle
- ZipHandle = pid()

The `t/1` function prints the names of all files in the zip archive `Archive` to the Erlang shell. (Similar to "`tart`".)

```
tt(Archive)
```

Types:

- Archive = filename() | binary()

The `tt/1` function prints names and information about all files in the zip archive `Archive` to the Erlang shell. (Similar to "`tar tv`".)

```
zip_open(Archive) -> {ok, ZipHandle} | {error, Reason}
zip_open(Archive, Options) -> {ok, ZipHandle} | {error, Reason}
```

Types:

- Archive = filename() | binary()
- Options = [Option]
- Options = cooked | memory | {cwd, CWD}
- CWD = string()
- ZipHandle = pid()

The `zip_open` function opens a zip archive, and reads and saves its directory. This means that subsequently reading files from the archive will be faster than unzipping files one at a time with `unzip`.

The archive must be closed with `zip_close/1`.

```
zip_list_dir(ZipHandle) -> Result | {error, Reason}
```

Types:

- Result = [ZipComment, ZipFile...]
- ZipComment = #zip_comment{}
- ZipFile = #zip_file{}
- ZipHandle = pid()

The `zip_list_dir/1` function returns the file list of an open zip archive.

```
zip_get(ZipHandle) -> {ok, [Result]} | {error, Reason}
zip_get(FileName, ZipHandle) -> {ok, Result} | {error, Reason}
```

Types:

- FileName = filename()
- ZipHandle = pid()
- Result = filename() | {filename(), binary()}

The `zip_get` function extracts one or all files from an open archive.

The files will be unzipped to memory or to file, depending on the options given to the `zip_open` function when the archive was opened.

```
zip_close(ZipHandle) -> ok | {error, einval}
```

Types:

- ZipHandle = pid()

The `zip_close/1` function closes a zip archive, previously opened with `zip_open`. All resources are closed, and the handle should not be used after closing.

Index of Modules and Functions

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