STDLIB

version 1.13
## Contents

1 **STD LIB Reference Manual** .............................................................. 1

1.1 beam_lib .......................................................................................... 50
1.2 c ........................................................................................................ 57
1.3 calendar ............................................................................................ 61
1.4 dets .................................................................................................... 66
1.5 dict ..................................................................................................... 82
1.6 digraph ............................................................................................... 87
1.7 digraph_utils ..................................................................................... 94
1.8 epp ...................................................................................................... 98
1.9 erl_eval .............................................................................................. 100
1.10 erl_id_trans ....................................................................................... 103
1.11 erl_internal ...................................................................................... 104
1.12 erl_lint ............................................................................................ 106
1.13 erl_parse .......................................................................................... 108
1.14 erl_pp .............................................................................................. 111
1.15 erl_scan ............................................................................................ 114
1.16 erl_tar .............................................................................................. 116
1.17 ets ..................................................................................................... 122
1.18 file_sorter .......................................................................................... 142
1.19 filelib ............................................................................................... 147
1.20 filename ............................................................................................ 150
1.21 gb_sets ............................................................................................. 156
1.22 gb_trees ............................................................................................ 162
1.23 gen_event .......................................................................................... 167
1.24 gen_fsm ............................................................................................ 177
1.25 gen_server ......................................................................................... 188
1.26 io ....................................................................................................... 197
1.27 io_lib ................................................................................................ 208
1.28 lib ..................................................................................................... 212
1.29 lists ................................................................................................. 214
<table>
<thead>
<tr>
<th>Section</th>
<th>Module/Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.30</td>
<td>log_mfh</td>
<td>228</td>
</tr>
<tr>
<td>1.31</td>
<td>math</td>
<td>229</td>
</tr>
<tr>
<td>1.32</td>
<td>ms_transform</td>
<td>231</td>
</tr>
<tr>
<td>1.33</td>
<td>orddict</td>
<td>242</td>
</tr>
<tr>
<td>1.34</td>
<td>ordsets</td>
<td>243</td>
</tr>
<tr>
<td>1.35</td>
<td>pg</td>
<td>244</td>
</tr>
<tr>
<td>1.36</td>
<td>pool</td>
<td>246</td>
</tr>
<tr>
<td>1.37</td>
<td>proc_lib</td>
<td>248</td>
</tr>
<tr>
<td>1.38</td>
<td>proplists</td>
<td>253</td>
</tr>
<tr>
<td>1.39</td>
<td>qlc</td>
<td>258</td>
</tr>
<tr>
<td>1.40</td>
<td>queue</td>
<td>272</td>
</tr>
<tr>
<td>1.41</td>
<td>random</td>
<td>276</td>
</tr>
<tr>
<td>1.42</td>
<td>regexp</td>
<td>278</td>
</tr>
<tr>
<td>1.43</td>
<td>sets</td>
<td>283</td>
</tr>
<tr>
<td>1.44</td>
<td>shell</td>
<td>286</td>
</tr>
<tr>
<td>1.45</td>
<td>shell_default</td>
<td>295</td>
</tr>
<tr>
<td>1.46</td>
<td>slave</td>
<td>296</td>
</tr>
<tr>
<td>1.47</td>
<td>sofs</td>
<td>299</td>
</tr>
<tr>
<td>1.48</td>
<td>string</td>
<td>322</td>
</tr>
<tr>
<td>1.49</td>
<td>supervisor</td>
<td>328</td>
</tr>
<tr>
<td>1.50</td>
<td>supervisor_bridge</td>
<td>335</td>
</tr>
<tr>
<td>1.51</td>
<td>sys</td>
<td>338</td>
</tr>
<tr>
<td>1.52</td>
<td>timer</td>
<td>345</td>
</tr>
<tr>
<td>1.53</td>
<td>win32reg</td>
<td>349</td>
</tr>
</tbody>
</table>
STDLIB Reference Manual

Short Summaries

- Erlang Module **beam_lib** [page 50] – An Interface To the BEAM File Format
- Erlang Module **c** [page 57] – Command Interface Module
- Erlang Module **calendar** [page 61] – Local and universal time, day-of-the-week, date and time conversions
- Erlang Module **dets** [page 66] – A Disk Based Term Storage
- Erlang Module **dict** [page 82] – Key-Value Dictionary
- Erlang Module **digraph** [page 87] – Directed Graphs
- Erlang Module **digraph_utils** [page 94] – Algorithms for Directed Graphs
- Erlang Module **epp** [page 98] – An Erlang Code Preprocessor
- Erlang Module **erl_eval** [page 100] – The Erlang Meta Interpreter
- Erlang Module **erl_id_trans** [page 103] – An Identity Parse Transform
- Erlang Module **erl_internal** [page 104] – Internal Erlang Definitions
- Erlang Module **erl_lint** [page 106] – The Erlang Code Linter
- Erlang Module **erl_parse** [page 108] – The Erlang Parser
- Erlang Module **erl_pp** [page 111] – The Erlang Pretty Printer
- Erlang Module **erl_scan** [page 114] – The Erlang Token Scanner
- Erlang Module **erl_tar** [page 116] – Unix ‘tar’ utility for reading and writing tar archives
- Erlang Module **ets** [page 122] – Built-In Term Storage
- Erlang Module **file_sorter** [page 142] – File Sorter
- Erlang Module **filelib** [page 147] – File utilities, such as wildcard matching of filenames
- Erlang Module **filename** [page 150] – Filename Manipulation Functions
- Erlang Module **gb_sets** [page 156] – General Balanced Trees
- Erlang Module **gb_trees** [page 162] – General Balanced Trees
- Erlang Module **gen_event** [page 167] – Generic Event Handling Behaviour
- Erlang Module **gen_fsm** [page 177] – Generic Finite State Machine Behaviour
- Erlang Module **gen_server** [page 188] – Generic Server Behaviour
- Erlang Module **io** [page 197] – Standard IO Server Interface Functions
- Erlang Module **io_lib** [page 208] – IO Library Functions
- Erlang Module `lib` [page 212] - A number of useful library functions
- Erlang Module `lists` [page 214] - List Processing Functions
- Erlang Module `log.mf.h` [page 228] - An Event Handler which Logs Events to Disk
- Erlang Module `math` [page 229] - Mathematical Functions
- Erlang Module `ms.transform` [page 231] - Parse_transform that translates fun syntax into match specifications.
- Erlang Module `orddict` [page 242] - Key-Value Dictionary as Ordered List
- Erlang Module `ordsets` [page 243] - Functions for Manipulating Sets as Ordered Lists
- Erlang Module `pg` [page 244] - Distributed, Named Process Groups
- Erlang Module `pool` [page 246] - Load Distribution Facility
- Erlang Module `proc_lib` [page 248] - Plug-in Replacements for spawn/1,2,3,4, spawn_link/1,2,3,4, and spawn_opt/2,3,4,5.
- Erlang Module `proplists` [page 253] - Support functions for property lists
- Erlang Module `qlc` [page 258] - Query Interface to Mnesia, ETS, Dets, etc
- Erlang Module `queue` [page 272] - Abstract Data Type for FIFO Queues
- Erlang Module `random` [page 276] - Pseudo random number generation
- Erlang Module `regexp` [page 278] - Regular Expression Functions for Strings
- Erlang Module `sets` [page 283] - Functions for Set Manipulation
- Erlang Module `shell` [page 286] - The Erlang Shell
- Erlang Module `shell_default` [page 295] - Customizing the Erlang Environment
- Erlang Module `slave` [page 296] - Functions to Starting and Controlling Slave Nodes
- Erlang Module `sofs` [page 299] - Functions for Manipulating Sets of Sets
- Erlang Module `string` [page 322] - String Processing Functions
- Erlang Module `supervisor_bridge` [page 335] - Generic Supervisor Bridge Behaviour.
- Erlang Module `sys` [page 338] - A Functional Interface to System Messages
- Erlang Module `timer` [page 345] - Timer Functions
- Erlang Module `win32reg` [page 349] - win32reg provides access to the registry on Windows

**beam_lib**

The following functions are exported:

- `chunks(Beam, [ChunkRef]) -> {ok, [Module, [ChunkData]], Reason} | {error, beam_lib, Reason}` [page 52] Read selected chunks from a BEAM file or binary
- `version(Beam) -> {ok, [Module, [Version]], Reason} | {error, beam_lib, Reason}` [page 52] Read the BEAM file's module version
- `info(Beam)` -> [(Item, Info)] | {error, beam_lib, Reason1}  
  [page 53] Information about a BEAM file
- `cmp(Beam1, Beam2)` -> ok | {error, beam_lib, Reason}  
  [page 53] Compare two BEAM files
- `cmp_dirs(Dir1, Dir2)` -> [Only1, Only2, Different] | {error, beam_lib, Reason1}  
  [page 53] Compare the BEAM files in two directories
- `diff_dirs(Dir1, Dir2)` -> ok | {error, beam_lib, Reason1}  
  [page 54] Compare the BEAM files in two directories
- `strip(Beam1)` -> [ok, Module, Beam2] | {error, beam_lib, Reason1}  
  [page 54] Removes chunks not needed by the loader from a BEAM file
- `strip_files(Files)` -> [ok, [Module, Beam2]] | {error, beam_lib, Reason1}  
  [page 54] Removes chunks not needed by the loader from BEAM files
- `strip_release(Dir)` -> [ok, [Module, Filename]] | {error, beam_lib, Reason1}  
  [page 54] Removes chunks not needed by the loader from all BEAM files of a release
- `format_error(Reason)` -> Chars  
  [page 55] Return an English description of a BEAM read error reply
- `crypto_key_fun(CryptoKeyFun)` -> ok | {error, Reason}  
  [page 55] Register a fun that provides a crypto key
- `clear_crypto_key_fun()` -> ok, Result  
  [page 55] Unregister the current crypto key fun

C

The following functions are exported:
- `bt(Pid)` -> void()  
  [page 57] Stack backtrace for a process
- `c(File)` -> [ok, Module] | error  
  [page 57] Compile and load code in a file
- `c(File, Options)` -> [ok, Module] | error  
  [page 57] Compile and load code in a file
- `cd(Dir)` -> void()  
  [page 57] Change working directory
- `flush()` -> void()  
  [page 58] Flush any messages sent to the shell
- `help()` -> void()  
  [page 58] Help information
- `i()` -> void()  
  [page 58] Information about the system
- `ni()` -> void()  
  [page 58] Information about the system
- `i(X, Y, Z)` -> void()  
  [page 58] Information about pid <X.Y.Z>
calendar

The following functions are exported:

- `date_to_gregorian_days(Date) -> Days`
  [page 62] Compute the number of days from year 0 up to the given date
- `date_to_gregorian_days(Year, Month, Day) -> Days`
  [page 62] Compute the number of days from year 0 up to the given date
- `datetime_to_gregorian_seconds([Date, Time]) -> Seconds`
  [page 62] Compute the number of seconds from year 0 up to the given date and time
- `day_of_the_week(Date) -> DayNumber`
  [page 62] Compute the day of the week
- `day_of_the_week(Year, Month, Day) -> DayNumber`
  [page 62] Compute the day of the week
- `gregorian_days_to_date(Days) -> Date`
  [page 62] Compute the date given the number of gregorian days
- `gregorian_seconds_to_datetime(Seconds) -> [Date, Time]`
  [page 62] Compute the date given the number of gregorian days
- `is_leap_year(Year) -> bool()`
  [page 62] Check if a year is a leap year
- `last_day_of_the_month(Year, Month) -> int()`  
  [page 62] Compute the number of days in a month
- `local_time() -> {Date, Time}`
  [page 63] Compute local time
- `local_time_to_universal_time([Date1, Time1]) -> {Date2, Time2}`
  [page 63] Convert from local time to universal time (deprecated)
- `local_time_to_universal_time_dst([Date1, Time1]) -> [{Date, Time}]`
  [page 63] Convert from local time to universal time(s)
- `now_to_local_time(Now) -> {Date, Time}`
  [page 63] Convert now to local date and time
- `now_to_universal_time(Now) -> {Date, Time}`
  [page 64] Convert now to date and time
- `now_to_datetime(Now) -> {Date, Time}`
  [page 64] Convert now to date and time
- `seconds_to_daytime(Seconds) -> {Days, Time}`
  [page 64] Compute days and time from seconds
- `seconds_to_time(Seconds) -> Time`
  [page 64] Compute time from seconds
- `time_difference(T1, T2) -> {Days, Time}`
  [page 64] Compute the difference between two times (deprecated)
- `time_to_seconds(Time) -> Seconds`
  [page 64] Compute the number of seconds since midnight up to the given time
- `universal_time() -> {Date, Time}`
  [page 64] Compute universal time
- `universal_time_to_local_time([Date1, Time1]) -> {Date2, Time2}`
  [page 65] Convert from universal time to local time
- `valid_date(Date) -> bool()`
  [page 65] Check if a date is valid
- `valid_date(Year, Month, Day) -> bool()`
  [page 65] Check if a date is valid
dets

The following functions are exported:

- `all() -> [Name]`
  - [page 67] Return a list of the names of all open Dets tables on this node.
- `bchunk(Name, Continuation) -> {Continuation2, Data} | '$end_of_table' | {error, Reason}`
  - [page 67] Return a chunk of objects stored in a Dets table.
- `close(Name) -> ok | {error, Reason}`
  - [page 68] Close a Dets table.
- `delete(Name, Key) -> ok | {error, Reason}`
  - [page 68] Delete all objects with a given key from a Dets table.
- `delete_all_objects(Name) -> ok | {error, Reason}`
  - [page 68] Delete all objects from a Dets table.
- `delete_object(Name, Object) -> ok | {error, Reason}`
  - [page 68] Delete a given object from a Dets table.
- `first(Name) -> Key | '$end_of_table'`
  - [page 68] Return the first key stored in a Dets table.
- `foldl(Function, Acc0, Name) -> Acc1 | {error, Reason}`
  - [page 69] Fold a function over a Dets table.
- `foldr(Function, Acc0, Name) -> Acc1 | {error, Reason}`
  - [page 69] Fold a function over a Dets table.
- `from_ets(Name, EtsTab) -> ok | {error, Reason}`
  - [page 69] Replace the objects of a Dets table with the objects of an Ets table.
- `info(Name) -> InfoList | undefined`
  - [page 69] Return information about a Dets table.
- `info(Name, Item) -> Value | undefined`
  - [page 70] Return the information associated with a given item for a Dets table.
- `init_table(Name, InitFun [, Options]) -> ok | {error, Reason}`
  - [page 70] Replace all objects of a Dets table.
- `insert(Name, Objects) -> ok | {error, Reason}`
  - [page 71] Insert one or more objects into a Dets table.
- `insert_new(Name, Objects) -> Bool`
  - [page 71] Insert one or more objects into a Dets table.
- `is_compatible_bchunk_format(Name, BchunkFormat) -> Bool`
  - [page 72] Test compatibility of a table's chunk data.
- `is_dets_file(FileName) -> Bool | {error, Reason}`
  - [page 72] Test for a Dets table.
- `lookup(Name, Key) -> [Object] | {error, Reason}`
  - [page 72] Return all objects with a given key stored in a Dets table.
- `match(Continuation) -> {[Match], Continuation2} | '$end_of_table' | {error, Reason}`
  - [page 72] Match a chunk of objects stored in a Dets table and return a list of variable bindings.
- `match(Name, Pattern) -> [Match] | {error, Reason}`
  - [page 73] Match the objects stored in a Dets table and return a list of variable bindings.
- `match(Name, Pattern, N) -> {[Match], Continuation} | '$end_of_table' | {error, Reason}
  [page 73] 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 73] Delete all objects that match a given pattern from a Dets table.

- `match_object(Continuation) -> {[Object], Continuation2} | '$end_of_table' | {error, Reason}
  [page 74] Match a chunk of objects stored in a Dets table and return a list of objects.

- `match_object(Name, Pattern) -> {[Object], Continuation} | {error, Reason}
  [page 74] 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 74] Match the first chunk of objects stored in a Dets table and return a list of objects.

- `member(Name, Key) -> Bool | {error, Reason}
  [page 75] Test for occurrence of a key in a Dets table.

- `next(Name, Key1) -> Key2 | '$end_of_table'
  [page 75] Return the next key in a Dets table.

- `open_file(Filename) -> {ok, Reference} | {error, Reason}
  [page 75] Open an existing Dets table.

- `open_file(Name, Args) -> {ok, Name} | {error, Reason}
  [page 75] Open a Dets table.

- `pid2name(Pid) -> {ok, Name} | undefined
  [page 77] Return the name of the Dets table handled by a pid.

- `repair_continuation(Continuation, MatchSpec) -> Continuation2
  [page 77] Repair a continuation from select/1 or select/3.

- `safe_fixtable(Name, Fix)
  [page 77] Fix a Dets table for safe traversal.

- `select(Continuation) -> {Selection, Continuation2} | '$end_of_table' | {error, Reason}
  [page 78] Apply a match specification to some objects stored in a Dets table.

- `select(Name, MatchSpec) -> Selection | {error, Reason}
  [page 78] Apply a match specification to all objects stored in a Dets table.

- `select(Name, MatchSpec, N) -> {Selection, Continuation} | '$end_of_table' | {error, Reason}
  [page 78] Apply a match specification to the first chunk of objects stored in a Dets table.

- `select_delete(Name, MatchSpec) -> N | {error, Reason}
  [page 79] Delete all objects that match a given pattern from a Dets table.

- `slot(Name, I) -> '$end_of_table' | {Object} | {error, Reason}
  [page 79] Return the list of objects associated with a slot of a Dets table.

- `sync(Name) -> ok | {error, Reason}
  [page 79] Ensure that all updates made to a Dets table are written to disk.

- `table(Name [, Options]) -> QueryHandle
  [page 79] Return a QLC query handle.
to_ets(Name, EtsTab) -> EtsTab | {error, Reason}  
[page 80] Insert all objects of a Dets table into an Ets table.

traverse(Name, Fun) -> Return | {error, Reason}  
[page 81] Apply a function to all or some objects stored in a Dets table.

update_counter(Name, Key, Increment) -> Result  
[page 81] Update a counter object stored in a Dets table.

dict

The following functions are exported:

- append(Key, Value, Dict1) -> Dict2  
[page 82] Append a value to keys in a dictionary

- append_list(Key, ValList, Dict1) -> Dict2  
[page 82] Append new values to keys in a dictionary

- erase(Key, Dict1) -> Dict2  
[page 82] Erase a key from a dictionary

- fetch(Key, Dict) -> Value  
[page 82] Look-up values in a dictionary

- fetch_keys(Dict) -> Keys  
[page 83] Return all keys in a dictionary

- filter(Pred, Dict1) -> Dict2  
[page 83] Choose elements which satisfy a predicate

- find(Key, Dict) -> {ok, Value} | error  
[page 83] Search for a key in a dictionary

- fold(Fun, Acc0, Dict) -> Acc1  
[page 83] Fold a function over a dictionary

- from_list(List) -> Dict  
[page 83] Convert a list of pairs to a dictionary

- is_key(Key, Dict) -> bool()  
[page 83] Test if a key is in a dictionary

- map(Fun, Dict1) -> Dict2  
[page 84] Map a function over a dictionary

- merge(Fun, Dict1, Dict2) -> Dict3  
[page 84] Merge two dictionaries

- new() -> dictionary()  
[page 84] Create a dictionary

- store(Key, Value, Dict1) -> Dict2  
[page 84] Store a value in a dictionary

- to_list(Dict) -> List  
[page 84] Convert a dictionary to a list of pairs

- update(Key, Fun, Dict1) -> Dict2  
[page 85] Update a value in a dictionary

- update(Key, Fun, Initial, Dict1) -> Dict2  
[page 85] Update a value in a dictionary

- update_counter(Key, Increment, Dict1) -> Dict2  
[page 85] Increment a value in a dictionary
digraph

The following functions are exported:

- `add_edge(G, E, V1, V2, Label) -> edge() | {error, Reason}`  
  [page 87] Add an edge to a digraph.

- `add_edge(G, V1, V2, Label) -> edge() | {error, Reason}`  
  [page 87] Add an edge to a digraph.

- `add_edge(G, V1, V2) -> edge() | {error, Reason}`  
  [page 87] Add an edge to a digraph.

- `add_vertex(G, V, Label) -> vertex()`  
  [page 88] Add or modify a vertex of a digraph.

- `add_vertex(G, V) -> vertex()`  
  [page 88] Add or modify a vertex of a digraph.

- `add_vertex(G) -> vertex()`  
  [page 88] Add or modify a vertex of a digraph.

- `del_edge(G, E) -> true`  
  [page 88] Delete an edge from a digraph.

- `del_edges(G, Edges) -> true`  
  [page 88] Delete edges from a digraph.

- `del_path(G, V1, V2) -> true`  
  [page 88] Delete paths from a digraph.

- `del_vertex(G, V) -> true`  
  [page 89] Delete a vertex from a digraph.

- `del_vertices(G, Vertices) -> true`  
  [page 89] Delete vertices from a digraph.

- `delete(G) -> true`  
  [page 89] Delete a digraph.

- `edge(G, E) -> {E, V1, V2, Label} | false`  
  [page 89] Return the vertices and the label of an edge of a digraph.

- `edges(G) -> Edges`  
  [page 89] Return all edges of a digraph.

- `edges(G, V) -> Edges`  
  [page 89] Return the edges emanating from or incident on a vertex of a digraph.

- `get_cycle(G, V) -> Vertices | false`  
  [page 90] Find one cycle in a digraph.

- `get_path(G, V1, V2) -> Vertices | false`  
  [page 90] Find one path in a digraph.

- `get_short_cycle(G, V) -> Vertices | false`  
  [page 90] Find one short cycle in a digraph.

- `get_short_path(G, V1, V2) -> Vertices | false`  
  [page 90] Find one short path in a digraph.

- `in_degree(G, V) -> integer()`  
  [page 91] Return the in-degree of a vertex of a digraph.

- `in_edges(G, V) -> Edges`  
  [page 91] Return all edges incident on a vertex of a digraph.
• \texttt{in}\_neighbours(G, V) -> Vertices
  [page 91] Return all in-neighbours of a vertex of a digraph.
• \texttt{info}(G) -> InfoList
  [page 91] Return information about a digraph.
• \texttt{new()} -> digraph()
  [page 92] Return a protected empty digraph, where cycles are allowed.
• \texttt{new}(Type) -> digraph() | \{error, Reason\}
  [page 92] Create a new empty digraph.
• \texttt{no}\_edges(G) -> integer() >= 0
  [page 92] Return the number of edges of the a digraph.
• \texttt{no}\_vertices(G) -> integer() >= 0
  [page 92] Return the number of vertices of a digraph.
• \texttt{out}\_degree(G, V) -> integer()
  [page 92] Return the out-degree of a vertex of a digraph.
• \texttt{out}\_edges(G, V) -> Edges
  [page 92] Return all edges emanating from a vertex of a digraph.
• \texttt{out}\_neighbours(G, V) -> Vertices
  [page 92] Return all out-neighbours of a vertex of a digraph.
• \texttt{vertex}(G, V) -> \{V, Label\} | false
  [page 93] Return the label of a vertex of a digraph.
• \texttt{vertices}(G) -> Vertices
  [page 93] Return all vertices of a digraph.

\textbf{digraph\_utils}

The following functions are exported:

• \texttt{components}(Digraph) -> [Component]
  [page 95] Return the components of a digraph.
• \texttt{condensation}(Digraph) -> CondensedDigraph
  [page 95] Return a condensed graph of a digraph.
• \texttt{cyclic}\_strong\_components(Digraph) -> [StrongComponent]
  [page 95] Return the cyclic strong components of a digraph.
• \texttt{is}\_acyclic(Digraph) -> bool()
  [page 95] Check if a digraph is acyclic.
• \texttt{loop}\_vertices(Digraph) -> Vertices
  [page 95] Return the vertices of a digraph included in some loop.
• \texttt{postorder}(Digraph) -> Vertices
  [page 96] Return the vertices of a digraph in post-order.
• \texttt{preorder}(Digraph) -> Vertices
  [page 96] Return the vertices of a digraph in pre-order.
• \texttt{reachable}(Vertices, Digraph) -> Vertices
  [page 96] Return the vertices reachable from some vertices of a digraph.
• \texttt{reachable}\_neighbours(Vertices, Digraph) -> Vertices
  [page 96] Return the neighbours reachable from some vertices of a digraph.
• \texttt{reaching}(Vertices, Digraph) -> Vertices
  [page 96] Return the vertices that reach some vertices of a digraph.
- reaching_neighbours(Vertices, Digraph) -> Vertices
  [page 96] Return the neighbours that reach some vertices of a digraph.
- strong_components(Digraph) -> [StrongComponent]
  [page 97] Return the strong components of a digraph.
- subgraph(Digraph, Vertices [, Options]) -> Subgraph | {error, Reason}
  [page 97] Return a subgraph of a digraph.
- topsort(Digraph) -> Vertices | false
  [page 97] Return a topological sorting of the vertices of a digraph.

**epp**

The following functions are exported:

- open(FileName, IncludePath) -> {ok, Epp} | {error, ErrorDescriptor}
  [page 98] Open a file for preprocessing
- open(FileName, IncludePath, PredefMacros) -> {ok, Epp} | {error, ErrorDescriptor}
  [page 98] Open a file for preprocessing
- close(Epp) -> ok
  [page 98] Close the preprocessing of the file associated with Epp
- parse_erl_form(Epp) -> {ok, AbsForm} | {eof, Line} | {error, ErrorInfo}
  [page 98] Return the next Erlang form from the opened Erlang source file
- parse_file(FileName, IncludePath, PredefMacro) -> {ok, [Form]} | {error, OpenError}
  [page 98] Preprocess and parse an Erlang source file

**erl_eval**

The following functions are exported:

- exprs(Expressions, Bindings) -> {value, Value, NewBindings}
  [page 100] Evaluate expressions
- exprs(Expressions, Bindings, LocalFunctionHandler) -> {value, Value, NewBindings}
  [page 100] Evaluate expressions
- exprs(Expressions, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {value, Value, NewBindings}
  [page 100] Evaluate expressions
- expr(Expression, Bindings) -> {value, Value, NewBindings}
  [page 100] Evaluate expression
- expr(Expression, Bindings, LocalFunctionHandler) -> {value, Value, NewBindings}
  [page 100] Evaluate expression
- expr(Expression, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {value, Value, NewBindings}
  [page 100] Evaluate expression
- `expr_list(ExpressionList, Bindings) -> {ValueList, NewBindings}`
  [page 101] Evaluate a list of expressions
- `expr_list(ExpressionList, Bindings, LocalFunctionHandler) -> {ValueList, NewBindings}`
  [page 101] Evaluate a list of expressions
- `expr_list(ExpressionList, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {ValueList, NewBindings}`
  [page 101] Evaluate a list of expressions
- `new_bindings() -> BindingStruct`
  [page 101] Return a bindings structure
- `bindings(BindingStruct) -> Bindings`
  [page 101] Return bindings
- `binding(Name, BindingStruct) -> Binding`
  [page 101] Return bindings
- `add_binding(Name, Value, Bindings) -> BindingStruct`
  [page 101] Add a binding
- `del_binding(Name, Bindings) -> BindingStruct`
  [page 101] Delete a binding

**erl_id_trans**

The following functions are exported:

- `parse_transform(Forms, Options) -> Forms`
  [page 103] Transform Erlang forms

**erl_internal**

The following functions are exported:

- `bif(Name, Arity) -> bool()`
  [page 104] Test for an Erlang BIF
- `guard_bif(Name, Arity) -> bool()`
  [page 104] Test for an Erlang BIF allowed in guards
- `type_test(Name, Arity) -> bool()`
  [page 104] Test for a valid type test
- `arith_op(OpName, Arity) -> bool()`
  [page 104] Test for an arithmetic operator
- `bool_op(OpName, Arity) -> bool()`
  [page 104] Test for a Boolean operator
- `comp_op(OpName, Arity) -> bool()`
  [page 105] Test for a comparison operator
- `list_op(OpName, Arity) -> bool()`
  [page 105] Test for a list operator
- `send_op(OpName, Arity) -> bool()`
  [page 105] Test for a send operator
- `op_type(OpName, Arity) -> Type`
  [page 105] Return operator type
erl_lint

The following functions are exported:

- `module(AbsForms) -> {ok, Warnings} | {error, Errors, Warnings}`
  
  [page 106] Check a module for errors

- `module(AbsForms, FileName) -> {ok, Warnings} | {error, Errors, Warnings}`
  
  [page 106] Check a module for errors

- `module(AbsForms, FileName, CompileOptions) -> {ok, Warnings} | {error, Errors, Warnings}`
  
  [page 106] Check a module for errors

- `is_guard_test(Expr) -> bool()`
  
  [page 107] Test for a guard test

- `format_error(ErrorDescriptor) -> Chars`
  
  [page 107] Format an error descriptor

erl_parse

The following functions are exported:

- `parse_form(Tokens) -> {ok, AbsForm} | {error, ErrorInfo}`
  
  [page 108] Parse an Erlang form

- `parse_exprs(Tokens) -> {ok, Expr} list | {error, ErrorInfo}`
  
  [page 108] Parse Erlang expressions

- `parse_term(Tokens) -> {ok, Term} | {error, ErrorInfo}`
  
  [page 108] Parse an Erlang term

- `format_error(ErrorDescriptor) -> Chars`
  
  [page 107] Format an error descriptor

- `tokens(AbsTerm) -> Tokens`  
  
  [page 109] Generate a list of tokens for an expression

- `tokens(AbsTerm, MoreTokens) -> Tokens`  
  
  [page 109] Generate a list of tokens for an expression

- `normalise(AbsTerm) -> Data`  
  
  [page 109] Convert abstract form to an Erlang term

- `abstract(Data) -> AbsTerm`  
  
  [page 109] Convert an Erlang term into an abstract form

erl_pp

The following functions are exported:

- `form(Form) -> DeepCharList`
  
  [page 111] Pretty print a form

- `form(Form, HookFunction) -> DeepCharList`
  
  [page 111] Pretty print a form

- `attribute(Attribute) -> DeepCharList`
  
  [page 111] Pretty print an attribute
STDLIB Reference Manual

- attribute(Attribute, HookFunction) -> DeepCharList
  [page 111] Pretty print an attribute
- function(Function) -> DeepCharList
  [page 111] Pretty print a function
- function(Function, HookFunction) -> DeepCharList
  [page 111] Pretty print a function
- guard(Guard) -> DeepCharList
  [page 111] Pretty print a guard
- guard(Guard, HookFunction) -> DeepCharList
  [page 111] Pretty print a guard
- exprs(Expressions) -> DeepCharList
  [page 112] Pretty print Expressions
- exprs(Expressions, HookFunction) -> DeepCharList
  [page 112] Pretty print Expressions
- exprs(Expressions, Indent, HookFunction) -> DeepCharList
  [page 112] Pretty print Expressions
- expr(Expression) -> DeepCharList
  [page 112] Pretty print one Expression
- expr(Expression, HookFunction) -> DeepCharList
  [page 112] Pretty print one Expression
- expr(Expression, Indent, HookFunction) -> DeepCharList
  [page 112] Pretty print one Expression
- expr(Expression, Indent, Precedence, HookFunction) -> DeepCharList
  [page 112] Pretty print one Expression

erl_scan

The following functions are exported:

- string(CharList, StartLine) -> {ok, Tokens, EndLine} | Error
  [page 114] Scan a string and returns the Erlang tokens
- tokens(Continuation, CharList, StartLine) -> Return
  [page 114] Re-entrant scanner
- reserved_word(Atom) -> bool()
  [page 115] Test for a reserved word
- format_error(ErrorCode) -> string()
  [page 115] Format an error descriptor

erl_tar

The following functions are exported:

- add(TarDescriptor, Filename, Options) -> RetValue
  [page 117] Add a file to an open tar file
The following functions are exported:

- `all()` -> [Tab]
  [page 123] Return a list of all ETS tables.
- `delete(Tab)` -> true
  [page 123] Delete an entire ETS table.
- `delete(Tab, Key)` -> true
  [page 123] Delete all objects with a given key from an ETS table.
- `delete_all_objects(Tab)` -> true
  [page 123] Delete all objects in an ETS table.
- `delete_object(Tab, Object)` -> true
  [page 123] Deletes a specific from an ETS table.
- `file2tab(Filename)` -> {ok,Tab} | {error,Reason}
  [page 123] Read an ETS table from a file.
- `first(Tab)` -> Key | '$end_of_table'
  [page 124] Return the first key in an ETS table.
- `fix_table(Tab, true|false)` -> true | false
  [page 124] Fix an ETS table for safe traversal (obsolete).
- `foldl(Function, Acc0, Tab)` -> Acc1
  [page 124] Fold a function over an ETS table.
• foldr(Function, Acc0, Tab) -> Acc1
   [page 124] Fold a function over an ETS table

• from_dets(Tab, DetsTab) -> Tab
   [page 125] Fill an ETS table with objects from a Dets table.

• fun2ms(LiteralFun) -> MatchSpec
   [page 125] Pseudo function that transforms fun syntax to a match_spec.

• i() -> void()
   [page 126] Display information about all ETS tables on tty.

• i(Tab) -> void()
   [page 126] Browse an ETS table on tty.

• info(Tab) -> tuple() | undefined
   [page 126] Return information about an ETS table.

• info(Tab, Item) -> Value | undefined
   [page 127] Return the information associated with given item for an ETS table.

• init_table(Name, InitFun) -> true
   [page 127] Replace all objects of an ETS table.

• insert(Tab, ObjectOrObjects) -> true
   [page 128] Insert an object into an ETS table.

• insert_new(Tab, ObjectOrObjects) -> bool()
   [page 128] Insert an object into an ETS table if the key is not already present.

• is_compiled_ms(Term) -> bool()
   [page 128] Checks if an Erlang term is the result of ets:match_spec:compile

• last(Tab) -> Key | '$end_of_table'
   [page 129] Return the last key in an ETS table of type ordered_set.

• lookup(Tab, Key) -> [Object]
   [page 129] Return all objects with a given key in an ETS table.

• lookup_element(Tab, Key, Pos) -> Elem
   [page 129] Return the Pos:th element of all objects with a given key in an ETS table.

• match(Tab, Pattern) -> [Match]
   [page 130] Match the objects in an ETS table against a pattern.

• match(Tab, Pattern, Limit) -> [{[Match],Continuation} | '$end_of_table']
   [page 130] Match the objects in an ETS table against a pattern and returns part of the answers.

• match(Continuation) -> [{[Match],Continuation} | '$end_of_table']
   [page 131] Continues matching objects in an ETS table.

• match_delete(Tab, Pattern) -> true
   [page 131] Delete all objects which match a given pattern from an ETS table.

• match_object(Tab, Pattern) -> [Object]
   [page 131] Match the objects in an ETS table against a pattern.

• match_object(Tab, Pattern, Limit) -> [{[Match],Continuation} | '$end_of_table']
   [page 131] 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 131] Continues matching objects in an ETS table.
- `match_spec:compile(MatchSpec) -> CompiledMatchSpec`  
  [page 132] Compiles a match specification into its internal representation

- `match_spec:run(List,CompiledMatchSpec) -> list()`  
  [page 132] Performs matching, using a compiled match_spec, on a list of tuples

- `member(Tab, Key) -> true | false`  
  [page 133] Tests for occurrence of a key in an ETS table

- `new(Name, Options) -> tid()`  
  [page 133] Create a new ETS table.

- `next(Tab, Key1) -> Key2 | '$end_of_table'`  
  [page 134] Return the next key in an ETS table.

- `prev(Tab, Key1) -> Key2 | '$end_of_table'`  
  [page 134] Return the previous key in an ETS table of type ordered_set.

- `rename(Tab, Name) -> Name`  
  [page 134] Rename a named ETS table.

- `repair_continuation(Continuation, MatchSpec) -> Continuation`  
  [page 134] Repair a continuation from ets:select/1 or ets:select/3 that has passed through external representation

- `safe_fixtable(Tab, true|false) -> true`  
  [page 135] Fix an ETS table for safe traversal.

- `select(Tab, MatchSpec) -> [Object]`  
  [page 136] Match the objects in an ETS table against a match_spec.

- `select(Tab, MatchSpec, Limit) -> [{[Match],Continuation}] | '$end_of_table'`  
  [page 138] Match the objects in an ETS table against a match_spec and returns part of the answers.

- `select(Continuation) -> [{[Match],Continuation}] | '$end_of_table'`  

- `select_delete(Tab, MatchSpec) -> NumDeleted`  
  [page 138] 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 138] 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 139] Return all objects in a given slot of an ETS table.

- `tab2file(Tab, Filename) -> ok | {error,Reason}`  
  [page 139] Dump an ETS table to a file.

- `tab2list(Tab) -> [Object]`  
  [page 139] Return a list of all objects in an ETS table.

- `table(Tab [, Options]) -> QueryHandle`  
  [page 139] Return a QLC query handle.

- `test_ms(Tuple, MatchSpec) -> {ok, Result} | {error, Errors}`  

- `to_dets(Tab, DetsTab) -> Tab`  
  [page 141] Fill a Dets table with objects from an ETS table.

- `update_counter(Tab, Key, {Pos,Incr,Threshold,SetValue}) -> Result`  
  [page 141] Update a counter object in an ETS table.
• `update_counter(Tab, Key, {Pos, Incr}) -> Result`
  [page 141] Update a counter object in an ETS table.

• `update_counter(Tab, Key, Incr) -> Result`
  [page 141] Update a counter object in an ETS table.

`file_sorter`

The following functions are exported:

• `sort(FileName) -> Reply`
  [page 145] Sort terms on files.

• `sort(Input, Output) -> Reply`
  [page 145] Sort terms on files.

• `sort(Input, Output, Options) -> Reply`
  [page 145] Sort terms on files.

• `keysort(KeyPos, FileName) -> Reply`
  [page 145] Sort terms on files by key.

• `keysort(KeyPos, Input, Output) -> Reply`
  [page 145] Sort terms on files by key.

• `keysort(KeyPos, Input, Output, Options) -> Reply`
  [page 145] Sort terms on files by key.

• `merge(FileNames, Output) -> Reply`
  [page 145] Merge terms on files.

• `merge(FileNames, Output, Options) -> Reply`
  [page 145] Merge terms on files.

• `keymerge(KeyPos, FileNames, Output) -> Reply`
  [page 146] Merge terms on files by key.

• `keymerge(KeyPos, FileNames, Output, Options) -> Reply`
  [page 146] Merge terms on files by key.

• `check(FileName) -> Reply`
  [page 146] Check whether terms on files are sorted.

• `check(FileNames, Options) -> Reply`
  [page 146] Check whether terms on files are sorted.

• `keycheck(KeyPos, FileName) -> CheckReply`
  [page 146] Check whether terms on files are sorted by key.

• `keycheck(KeyPos, FileNames, Options) -> Reply`
  [page 146] Check whether terms on files are sorted by key.

`filelib`

The following functions are exported:

• `ensure_dir(Name) -> true`
  [page 147] Ensure that all parent directories needed to create Name exists.

• `file_size(Filename) -> integer()`  
  [page 147] Return the size in bytes of the file.

• `fold_files(Dir, RegExp, Recursive, Fun, AccIn) -> AccOut`
  [page 147] Fold over all files matching a regular expression.
file name

The following functions are exported:

- `is_dir(Name)` -> `true | false`
  [page 147] Test whether Name refer to a directory or not
- `is_file(Name)` -> `true | false`
  [page 147] Test whether Name refer to a file or directory.
- `is_regular(Name)` -> `true | false`
  [page 148] Test whether Name refer to a (regular) file.
- `last_modified(Name)` -> `{Year, Month, Day}, {Hour, Min, Sec}`
  [page 148] Return the local date and time when a file was last modified.
- `wildcard(Wildcard)` -> `list()`
- `wildcard(Wildcard, Cwd)` -> `list()`
  [page 149] Match filenames using Unix-style wildcards startin at a specified directory.

```plaintext
filename
```

The following functions are exported:

- `absname(Filename)` -> `string()`
  [page 150] Convert a filename to an absolute name, relative the working directory
- `absname(Filename, Dir)` -> `string()`
  [page 151] Convert a filename to an absolute name, relative a specified directory
- `absname_join(Dir, Filename)` -> `string()`
  [page 151] Join an absolute directory with a relative filename
- `basename(Filename)` -> `string()`
  [page 151] Return the last component of a filename
- `basename(Filename, Ext)` -> `string()`
  [page 151] Return the last component of a filename, stripped of the specified extension
- `dirname(Filename)` -> `string()`
  [page 152] Return the directory part of a path name
- `extension(Filename)` -> `string()`
  [page 152] Return the file extension
- `flatten(Filename)` -> `string()`
  [page 152] Convert a filename to a flat string
- `join(Components)` -> `string()`
  [page 153] Join a list of filename components with directory separators
- `join(Name1, Name2)` -> `string()`
  [page 153] Join two filename components with directory separators
- `nativename(Path)` -> `string()`
  [page 153] Return the native form of a file path
- `pathtype(Path)` -> `absolute | relative | volumerelative`
  [page 153] Return the type of a path
- `rootname(Filename)` -> `string()`
  [page 154] Remove a filename extension
- `rootname(Filename, Ext)` -> `string()`
  [page 154] Remove a filename extension
• split(Filename) -> Components
  [page 154] Split a filename into its path components

• find_src(Beam) -> {SourceFile, Options}
  [page 154] Find the filename and compiler options for a module

• find_src(Beam, Rules) -> {SourceFile, Options}
  [page 154] Find the filename and compiler options for a module

gb_sets

The following functions are exported:

• add(Element, Set1) -> Set2
  [page 157] Add a (possibly existing) element to a gb_set

• add_element(Element, Set1) -> Set2
  [page 157] Add a (possibly existing) element to a gb_set

• balance(Set1) -> Set2
  [page 157] Rebalance tree representation of a gb_set

• delete(Element, Set1) -> Set2
  [page 157] Remove an element from a gb_set

• delete_any(Element, Set1) -> Set2
  [page 157] Remove a (possibly non-existing) element from a gb_set

• del_element(Element, Set1) -> Set2
  [page 157] Remove a (possibly non-existing) element from a gb_set

• difference(Set1, Set2) -> Set3
  [page 157] Return the difference of two gb_sets

• subtract(Set1, Set2) -> Set3
  [page 158] Return the difference of two gb_sets

• empty() -> Set
  [page 158] Return an empty gb_set

• new() -> Set
  [page 158] Return an empty gb_set

• filter(Pred, Set1) -> Set2
  [page 158] Filter gb_set elements

• fold(Function, Acc0, Set) -> Acc1
  [page 158] Fold over gb_set elements

• from_list(List) -> Set
  [page 158] Convert a list into a gb_set

• from_ordset(List) -> Set
  [page 158] Make a gb_set from an ordset list

• insert(Element, Set1) -> Set2
  [page 158] Add a new element to a gb_set

• intersection(Set1, Set2) -> Set3
  [page 159] Return the intersection of two gb_sets

• intersection(SetList) -> Set
  [page 159] Return the intersection of a list of gb_sets

• is_empty(Set) -> bool()
  [page 159] Test for empty gb_set
The following functions are exported:

- `is_member(Element, Set) -> bool()` [page 159] Test for membership of a gb_set
- `is_element(Element, Set) -> bool()` [page 159] Test for membership of a gb_set
- `is_set(Set) -> bool()` [page 159] Test for a gb_set
- `is_subset(Set1, Set2) -> bool()` [page 159] Test for subset
- `iterator(Set) -> Iter` [page 159] Return an iterator for a gb_set
- `largest(Set) -> term()` [page 160] Return largest element
- `next(Iter1) -> {Element, Iter2 | none}` [page 160] Traverse a gb_set with an iterator
- `singleton(Element) -> gb_set()` [page 160] Return a gb_set with one element
- `size(Set) -> int()` [page 160] Return the number of elements in a gb_set
- `smallest(Set) -> term()` [page 160] Return smallest element
- `take_largest(Set1) -> {Element, Set2}` [page 160] Extract largest element
- `take_smallest(Set1) -> {Element, Set2}` [page 160] Extract smallest element
- `to_list(Set) -> List` [page 161] Convert a gb_set into a list
- `union(Set1, Set2) -> Set3` [page 161] Return the union of two gb_sets
- `union(SetList) -> Set` [page 161] Return the union of a list of gb_sets

**gb_trees**

The following functions are exported:

- `balance(Tree1) -> Tree2` [page 162] Rebalance a tree
- `delete(Key, Tree1) -> Tree2` [page 162] Remove a node from a tree
- `delete_any(Key, Tree1) -> Tree2` [page 163] Remove a (possibly non-existing) node from a tree
- `empty() -> Tree` [page 163] Return an empty tree
- `enter(Key, Val, Tree1) -> Tree2` [page 163] Insert or update key with value in a tree
- `from_orddict(List) -> Tree` [page 163] Make a tree from an orddict
- get(Key, Tree) -> Val
  [page 163] Look up a key in a tree, if present
- lookup(Key, Tree) -> {value, Val} | none
  [page 163] Look up a key in a tree
- insert(Key, Val, Tree1) -> Tree2
  [page 164] Insert a new key and value in a tree
- is_defined(Key, Tree) -> bool()
  [page 164] Test for membership of a tree
- is_empty(Tree) -> bool()
  [page 164] Test for empty tree
- iterator(Tree) -> Iter
  [page 164] Return an iterator for a tree
- keys(Tree) -> [Key]
  [page 164] Return a list of the keys in a tree
- largest(Tree) -> {Key, Val}
  [page 164] Return largest key and value
- next(Iter1) -> {Key, Val, Iter2}
  [page 164] Traverse a tree with an iterator
- size(Tree) -> int()
  [page 165] Return the number of nodes in a tree
- smallest(Tree) -> {Key, Val}
  [page 165] Return smallest key and value
- take_largest(Tree1) -> {Key, Val, Tree2}
  [page 165] Extract largest key and value
- take_smallest(Tree1) -> {Key, Val, Tree2}
  [page 165] Extract smallest key and value
- to_list(Tree) -> [{Key, Val}]
  [page 165] Convert a tree into a list
- update(Key, Val, Tree1) -> Tree2
  [page 165] Update a key to new value in a tree
- values(Tree) -> [Val]
  [page 166] Return a list of the values in a tree

### gen_event

The following functions are exported:

- start_link() -> Result
  [page 168] Create a generic event manager process in a supervision tree.
- start_link(EventMgrName) -> Result
  [page 168] Create a generic event manager process in a supervision tree.
- start() -> Result
  [page 168] Create a stand-alone event manager process.
- start(EventMgrName) -> Result
  [page 168] Create a stand-alone event manager process.
- add_handler(EventMgrRef, Handler, Args) -> Result
  [page 168] Add an event handler to a generic event manager.
### STDLIB Reference Manual

- `add_sup_handler(EventMgrRef, Handler, Args) -> Result`  
  [page 169] Add a supervised event handler to a generic event manager.

- `notify(EventMgrRef, Event) -> ok`  
  [page 170] Notify an event manager about an event.

- `sync_notify(EventMgrRef, Event) -> ok`  
  [page 170] Notify an event manager about an event.

- `call(EventMgrRef, Handler, Request) -> Result`  
  [page 170] Make a synchronous call to a generic event manager.

- `call(EventMgrRef, Handler, Request, Timeout) -> Result`  
  [page 170] Make a synchronous call to a generic event manager.

- `delete_handler(EventMgrRef, Handler, Args) -> Result`  
  [page 171] Delete an event handler from a generic event manager.

- `swap_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result`  
  [page 171] Replace an event handler in a generic event manager.

- `swap_sup_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result`  
  [page 172] Replace an event handler in a generic event manager.

- `which_handlers(EventMgrRef) -> [Handler]`  
  [page 172] Return all event handlers installed in a generic event manager.

- `stop(EventMgrRef) -> ok`  
  [page 173] Terminate a generic event manager.

- `Module:init(InitArgs) -> {ok,State}`  
  [page 173] Initialize an event handler.

- `Module:handle_event(Event, State) -> Result`  
  [page 173] Handle an event.

- `Module:handle_call(Request, State) -> Result`  
  [page 174] Handle a synchronous request.

- `Module:handle_info(Info, State) -> Result`  
  [page 174] Handle an incoming message.

- `Module:terminate(Arg, State) -> term()`  
  [page 175] Clean up before deletion.

- `Module:code_change(OldVsn, State, Extra) -> {ok,NewState}`  
  [page 175] Update the internal state during upgrade/downgrade.

#### gen_fsm

The following functions are exported:

- `start_link(Module, Args, Options) -> Result`  
  [page 178] Create a `gen_fsm` process in a supervision tree.

- `start_link(FsmName, Module, Args, Options) -> Result`  
  [page 178] Create a `gen_fsm` process in a supervision tree.

- `start(Module, Args, Options) -> Result`  
  [page 179] Create a stand-alone `gen_fsm` process.

- `start(FsmName, Module, Args, Options) -> Result`  
  [page 179] Create a stand-alone `gen_fsm` process.
- `send_event(FsmRef, Event) -> ok`  
  [page 179] Send an event asynchronously to a generic FSM.

- `send_all_state_event(FsmRef, Event) -> ok`  
  [page 179] Send an event asynchronously to a generic FSM.

- `sync_send_event(FsmRef, Event) -> Reply`  
  [page 180] Send an event synchronously to a generic FSM.

- `sync_send_event(FsmRef, Event, Timeout) -> Reply`  
  [page 180] Send an event synchronously to a generic FSM.

- `sync_send_all_state_event(FsmRef, Event) -> Reply`  
  [page 180] Send an event synchronously to a generic FSM.

- `sync_send_all_state_event(FsmRef, Event, Timeout) -> Reply`  
  [page 180] Send an event synchronously to a generic FSM.

- `reply(Caller, Reply) -> true`  
  [page 181] Send a reply to a caller.

- `send_event_after(Time, Event) -> Ref`  
  [page 181] Send a delayed event internally in a generic FSM.

- `start_timer(Time, Msg) -> Ref`  
  [page 181] Send a timeout event internally in a generic FSM.

- `cancel_timer(Ref) -> RemainingTime | false`  
  [page 181] Cancel an internal timer in a generic FSM.

- `enter_loop(Module, Options, StateName, StateData)`  
  [page 182] Enter the gen_fsm receive loop.

- `enter_loop(Module, Options, StateName, StateData, FsmName)`  
  [page 182] Enter the gen_fsm receive loop.

- `enter_loop(Module, Options, StateName, StateData, Timeout)`  
  [page 182] Enter the gen_fsm receive loop.

- `enter_loop(Module, Options, StateName, StateData, FsmName, Timeout)`  
  [page 182] Enter the gen_fsm receive loop.

- `Module: init(Args) -> Result`  
  [page 183] Initialize process and internal state name and state data.

- `Module: StateName(Event, StateData) -> Result`  
  [page 183] Handle an asynchronous event.

- `Module: handle_event(Event, StateName, StateData) -> Result`  
  [page 184] Handle an asynchronous event.

- `Module: StateName(Event, From, StateData) -> Result`  
  [page 184] Handle a synchronous event.

- `Module: handle_sync_event(Event, From, StateName, StateData) -> Result`  
  [page 185] Handle a synchronous event.

- `Module: handle_info(Info, StateName, StateData) -> Result`  
  [page 185] Handle an incoming message.

- `Module: terminate(Reason, StateName, StateData)`  
  [page 186] Clean up before termination.

- `Module: code_change(OldVsn, StateName, StateData, Extra) -> {ok, NextStateName, NewStateData}`  
  [page 186] Update the internal state data during upgrade/downgrade.
The following functions are exported:

- **start_link(Module, Args, Options) -> Result**
  [page 188] Create a gen_server process in a supervision tree.

- **start_link(ServerName, Module, Args, Options) -> Result**
  [page 188] Create a gen_server process in a supervision tree.

- **start(Module, Args, Options) -> Result**
  [page 189] Create a stand-alone gen_server process.

- **start(ServerName, Module, Args, Options) -> Result**
  [page 189] Create a stand-alone gen_server process.

- **call(ServerRef, Request) -> Reply**
  [page 190] Make a synchronous call to a generic server.

- **call(ServerRef, Request, Timeout) -> Reply**
  [page 190] Make a synchronous call to a generic server.

- **multi_call(Name, Request) -> Result**
  [page 191] Make a synchronous call to several generic servers.

- **multi_call(Nodes, Name, Request) -> Result**
  [page 191] Make a synchronous call to several generic servers.

- **multi_call(Nodes, Name, Request, Timeout) -> Result**
  [page 191] Make a synchronous call to several generic servers.

- **cast(ServerRef, Request) -> ok**
  [page 192] Send an asynchronous request to a generic server.

- **abcast(Name, Request) -> abcast**
  [page 192] Send an asynchronous request to several generic servers.

- **abcast(Nodes, Name, Request) -> abcast**
  [page 192] Send an asynchronous request to several generic servers.

- **reply(Client, Reply) -> true**
  [page 192] Send a reply to a client.

- **enter_loop(Module, Options, State)**
  [page 192] Enter the gen_server receive loop.

- **enter_loop(Module, Options, State, ServerName)**
  [page 193] Enter the gen_server receive loop.

- **enter_loop(Module, Options, State, Timeout)**
  [page 193] Enter the gen_server receive loop.

- **enter_loop(Module, Options, State, ServerName, Timeout)**
  [page 193] Enter the gen_server receive loop.

- **Module:init(Args) -> Result**
  [page 193] Initialize process and internal state.

- **Module:handle_call(Request, From, State) -> Result**
  [page 194] Handle a synchronous request.

- **Module:handle_cast(Request, State) -> Result**
  [page 194] Handle an asynchronous request.

- **Module:handle_info(Info, State) -> Result**
  [page 195] Handle an incoming message.
Module: terminate(Reason, State)
   [page 195] Clean up before termination.

Module: code_change(OldVsn, State, Extra) -> {ok, NewState}
   [page 196] Update the internal state during upgrade/downgrade.

io

The following functions are exported:

- put_chars([IoDevice,] IoData) -> ok
  [page 197] Write a list of characters

- nl([IoDevice]) -> ok
  [page 197] Write a newline

- get_chars([IoDevice,] Prompt, Count) -> string() | eof
  [page 197] Read a specified number of characters

- get_line([IoDevice,] Prompt) -> string() | eof
  [page 198] Read a line

- setopts([IoDevice,] Opts) -> ok | {error, Reason}
  [page 198] Set options

- write([IoDevice,] Term) -> ok
  [page 198] Write a term

- read([IoDevice,] Prompt) -> Result
  [page 198] Read a term

- read(IoDevice, Prompt, StartLine) -> Result
  [page 199] Read a term

- fwrite(Format) ->
  [page 199] Write formatted output

- fwrite([IoDevice,] Format, Data) -> ok
  [page 199] Write formatted output

- format(Format) ->
  [page 199] Write formatted output

- format([IoDevice,] Format, Data) -> ok
  [page 199] Write formatted output

- fread([IoDevice,] Prompt, Format) -> Result
  [page 203] Read formatted input

- scan_erl_exprs(Prompt) ->
  [page 204] Read and tokenize Erlang expressions

- scan_erl_exprs([IoDevice,] Prompt, StartLine) -> Result
  [page 204] Read and tokenize Erlang expressions

- scan_erl_form(Prompt) ->
  [page 205] Read and tokenize an Erlang form

- scan_erl_form([IoDevice,] Prompt, StartLine) -> Result
  [page 205] Read and tokenize an Erlang form

- parse_erl_exprs(Prompt) ->
  [page 205] Read, tokenize and parse Erlang expressions

- parse_erl_exprs([IoDevice,] Prompt, StartLine) -> Result
  [page 205] Read, tokenize and parse Erlang expressions
- `parse.erl_form(Prompt)`: [page 206] Read, tokenize and parse an Erlang form
- `parse.erl_form([IoDevice,] Prompt, StartLine)`: [page 206] Read, tokenize and parse an Erlang form

### io_lib

The following functions are exported:

- `nl()`: [page 208] Write a newline
- `write(Term)`: [page 208] Write a term
- `write(Term, Depth)`: [page 208] Write a term
- `print(Term)`: [page 208] Pretty print a term
- `print(Term, Column, LineLength, Depth)`: [page 208] Pretty print a term
- `fwrite(Format, Data)`: [page 209] Write formatted output
- `format(Format, Data)`: [page 209] Write formatted output
- `fread(Format, String)`: [page 209] Read formatted input
- `fread(Continuation, String, Format)`: [page 209] Re-entrant formatted reader
- `write_atom(Atom)`: [page 210] Write an atom
- `write_string(String)`: [page 210] Write a string
- `write_char(Integer)`: [page 210] Write a character
- `indentation(String, StartIndent)`: [page 210] Indentation after printing string
- `char_list(Term)`: [page 211] Test for a list of characters
- `deep_char_list(Term)`: [page 211] Test for a deep list of characters
- `printable_list(Term)`: [page 211] Test for a list of printable characters
lib

The following functions are exported:

- `flush_receive() -> void()`  
  [page 212] Flush messages
- `error_message(Format, Args) -> ok`  
  [page 212] Print error message
- `proname() -> atom()`  
  [page 212] Return name of Erlang start script
- `nonl(String1) -> String2`  
  [page 212] Remove last newline
- `send(To, Msg)`  
  [page 212] Send a message
- `sendv(To, Msg)`  
  [page 213] Send a message and wait for an answer

lists

The following functions are exported:

- `append(ListOfLists) -> List1`  
  [page 214] Append a list of lists
- `append(List1, List2) -> List3`  
  [page 214] Append two lists
- `concat(Things) -> string()`  
  [page 214] Concatenate a list of atoms
- `delete(Elem, List1) -> List2`  
  [page 215] Delete an element from a list
- `duplicate(N, Elem) -> List`  
  [page 215] Make N copies of element
- `flatlength(DeepList) -> int()`  
  [page 215] Length of flattened deep list
- `flatten(DeepList) -> List`  
  [page 215] Flatten a deep list
- `flatten(DeepList, Tail) -> List`  
  [page 215] Flatten a deep list
- `keydelete(Key, N, TupleList1) -> TupleList2`  
  [page 215] Delete an element from a list of tuples
- `keymember(Key, N, TupleList) -> bool()`  
  [page 216] Test for membership of a list of tuples
- `keymerge(N, TupleList1, TupleList2) -> TupleList3`  
  [page 216] Merge two key-sorted lists of tuples
- `keyreplace(Key, N, TupleList1, NewTuple) -> TupleList2`  
  [page 216] Replace an element in a list of tuples
- `keysearch(Key, N, TupleList) -> {value, Tuple} | false`  
  [page 216] Search for an element in a list of tuples
- `keysort(N, TupleList1) -> TupleList2`  
  [page 216] Sort a list of tuples
- `last(List) -> Last`  
  [page 217] Return last element in a list
- `max(List) -> Max`  
  [page 217] Return maximum element of list
- `member(Elem, List) -> bool()`  
  [page 217] Test for membership of a list
- `merge(ListOfLists) -> List1`  
  [page 217] Merge a list of sorted lists
- `merge(List1, List2) -> List3`  
  [page 217] Merge two sorted lists
- `merge(Fun, List1, List2) -> List3`  
  [page 217] Merge two sorted list
- `merge3(List1, List2, List3) -> List4`  
  [page 218] Merge three sorted lists
- `min(List) -> Min`  
  [page 218] Return minimum element of list
- `nth(N, List) -> Elem`  
  [page 218] Return the Nth element of a list
- `nthtail(N, List1) -> Tail`  
  [page 218] Return the Nth tail of a list
- `prefix(List1, List2) -> bool()`  
  [page 218] Test for list prefix
- `reverse(List1) -> List2`  
  [page 218] Reverse a list
- `reverse(List1, Tail) -> List2`  
  [page 219] Reverse a list appending a tail
- `seq(From, To) -> Seq`  
  [page 219] Generate a sequence of integers
- `seq(From, To, Incr) -> Seq`  
  [page 219] Generate a sequence of integers
- `sort(List1) -> List2`  
  [page 219] Sort a list
- `sort(Fun, List1) -> List2`  
  [page 219] Sort a list
- `split(N, List1) -> {List2, List3}`  
  [page 220] Split a list into two lists
- `sublist(List1, N) -> List2`  
  [page 220] Return the first N elements of a list
- `sublist(List1, Start, Length) -> List2`  
  [page 220] Return a sub-list of a list
- `subtract(List1, List2) -> List3`  
  [page 220] Subtract the element in one list from another list
- `suffix(List1, List2) -> bool()`  
  [page 220] Test for list suffix
- `sum(List) -> number()`  
  [page 221] Return sum of elements in a list
- `ukeymerge(N, TupleList1, TupleList2) -> TupleList3`  
  [page 221] Merge two key-sorted lists of tuples, removing duplicates
- `ukeysort(N, TupleList1) -> TupleList2`  
  [page 221] Sort a list of tuples, removing duplicates
- `umerge(ListOfLists) -> List1`  
  [page 221] Merge a list of sorted lists, removing duplicates
- `umerge(List1, List2) -> List3`  
  [page 221] Merge two sorted lists, removing duplicates
- `umerge(Fun, List1, List2) -> List3`  
  [page 221] Merge two sorted lists, removing duplicates
- `umerge3(List1, List2, List3) -> List4`  
  [page 222] Merge three sorted lists, removing duplicates
- `unzip(List1) -> {List2, List3}`  
  [page 222] Unzip a list of two-tuples into two lists
- `unzip3(List1) -> {List2, List3, List4}`  
  [page 222] Unzip a list of three-tuples into three lists
- `usort(List1) -> List2`  
  [page 222] Sort a list, removing duplicates
- `usort(Fun, List1) -> List2`  
  [page 222] Sort a list, removing duplicates
- `zip(List1, List2) -> List3`  
  [page 222] Zip two lists into a list of two-tuples
- `zip3(List1, List2, List3) -> List4`  
  [page 223] Zip three lists into a list of three-tuples
- `zipwith(Combine, List1, List2) -> List3`  
  [page 223] Zip two lists into one list according to a fun
- `zipwith3(Combine, List1, List2, List3) -> List4`  
  [page 223] Zip three lists into one list according to a fun
- `all(Pred, List) -> bool()`  
  [page 224] Return true if all elements in the list satisfy Pred
- `any(Pred, List) -> bool()`  
  [page 224] Return true if any of the elements in the list satisfies Pred
- `dropwhile(Pred, List1) -> List2`  
  [page 224] Drop elements from a list while a predicate is true
- `filter(Pred, List1) -> List2`  
  [page 224] Choose elements which satisfy a predicate
- `flatMap(Fun, List1) -> List2`  
  [page 225] Map and flatten in one pass
- `foldl(Fun, Acc0, List) -> Acc1`  
  [page 225] Fold a function over a list
- `foldr(Fun, Acc0, List) -> Acc1`  
  [page 225] Fold a function over a list
- `foreach(Fun, List) -> void()`  
  [page 226] Apply a function to each element of a list
- `map(Fun, List1) -> List2`
  [page 226] Map a function over a list
- `mapfoldl(Fun, Acc0, List1) -> {List2, Acc1}`
  [page 226] Map and fold in one pass
- `mapfoldr(Fun, Acc0, List1) -> {List2, Acc1}`
  [page 226] Map and fold in one pass
- `partition(Pred, List) -> {Satisfying, NonSatisfying}`
  [page 227] Partition a list into two lists based on a predicate
- `splitwith(Pred, List) -> {List1, List2}`
  [page 227] Split a list into two lists based on a predicate
- `takewhile(Pred, List1) -> List2`
  [page 227] Take elements from a list while a predicate is true

### log-mf.h

The following functions are exported:

- `init(Dir, MaxBytes, MaxFiles)`
  [page 228] Initiate the event handler
- `init(Dir, MaxBytes, MaxFiles, Pred) -> Args`
  [page 228] Initiate the event handler

### math

The following functions are exported:

- `pi() -> float()`
  [page 229] A useful number
- `sin(X)`
  [page 229] Diverse math functions
- `cos(X)`
  [page 229] Diverse math functions
- `tan(X)`
  [page 229] Diverse math functions
- `asin(X)`
  [page 229] Diverse math functions
- `acos(X)`
  [page 229] Diverse math functions
- `atan(X)`
  [page 229] Diverse math functions
- `atan2(Y, X)`
  [page 229] Diverse math functions
- `sinh(X)`
  [page 229] Diverse math functions
- `cosh(X)`
  [page 229] Diverse math functions
- `tanh(X)`
  [page 229] Diverse math functions
• asinh(X)  
  [page 229] Diverse math functions
• acosh(X)  
  [page 229] Diverse math functions
• atanh(X)  
  [page 229] Diverse math functions
• exp(X)  
  [page 229] Diverse math functions
• log(X)  
  [page 229] Diverse math functions
• log10(X)  
  [page 229] Diverse math functions
• pow(X, Y)  
  [page 229] Diverse math functions
• sqrt(X)  
  [page 229] Diverse math functions
• erf(X)  
• erfc(X)  
  [page 230] Another error function

ms_transform

The following functions are exported:

• parse_transform(Forms, Options) -> Forms  
  [page 240] Transforms Erlang abstract format containing calls to ets/dbg:fun2ms into literal match specifications.
• transform_from_shell(Dialect, Clauses, BoundEnvironment) -> term()  
  [page 240] Used when transforming fun's created in the shell into match specifications.
• format_error(Errcode) -> ErrorMessage  
  [page 241] Error formatting function as required by the parse_transform interface.

orddict

No functions are exported.

ordsets

No functions are exported.
pg

The following functions are exported:

- `create(PgName) -> ok | {error, Reason}`  
  [page 244] Create an empty group
- `create(PgName, Node) -> ok | {error, Reason}`  
  [page 244] Create an empty group on another node
- `join(PgName, Pid) -> Members`  
  [page 244] Join a pid to a process group
- `send(PgName, Msg) -> void()`  
  [page 245] Send a message to all members of a process group
- `esend(PgName, Msg) -> void()`  
  [page 245] Send a message to all members of a process group, except ourselves
- `members(PgName) -> Members`  
  [page 245] Return a list of all members of a process group

pool

The following functions are exported:

- `start(Name) ->`  
  [page 246] Start a new pool
- `start(Name, Args) -> Nodes`  
  [page 246] Start a new pool
- `attach(Node) -> allready_attached | attached`  
  [page 246] Ensure that a pool master is running
- `stop() ->`  
  [page 247] Stop the pool and kill all the slave nodes
- `get_nodes() -> Nodes`  
  [page 247] Return a list of the current member nodes of the pool
- `pspawn(Mod, Fun, Args) -> pid()`  
  [page 247] Spawn a process on the pool node with expected lowest future load
- `pspawn_link(Mod, Fun, Args) -> pid()`  
  [page 247] Spawn and link to a process on the pool node with expected lowest future load
- `get_node() -> node()`  
  [page 247] Return the node with the expected lowest future load

proc_lib

The following functions are exported:

- `spawn(Fun) -> Pid`  
  [page 248] Spawn a new process.
- `spawn(Node,Fun) -> Pid`  
  [page 248] Spawn a new process.
- `spawn(Module,Func,Args) -> Pid`  
  [page 248] Spawn a new process.
• spawn(Node, Module, Func, Args) -> Pid
  [page 248] Spawn a new process.

• spawn_link(Fun) -> Pid
  [page 248] Spawn a new process and set a link.

• spawn_link(Node, Fun) -> Pid
  [page 248] Spawn a new process and set a link.

• spawn_link(Module, Func, Args) -> Pid
  [page 248] Spawn a new process and set a link.

• spawn_link(Node, Module, Func, Args) -> Pid
  [page 248] Spawn a new process and set a link.

• spawn_opt(Fun, Opts) -> Pid
  [page 249] Spawn a new process with given options.

• spawn_opt(Node, Fun, Opts) -> Pid
  [page 249] Spawn a new process with given options.

• spawn_opt(Module, Func, Args, Opts) -> Pid
  [page 249] Spawn a new process with given options.

• spawn_opt(Node, Module, Func, Args, Opts) -> Pid
  [page 249] Spawn a new process with given options.

• start(Module, Func, Args) -> Ret
  [page 249] Start a new process synchronously.

• start(Module, Func, Args, Time) -> Ret
  [page 249] Start a new process synchronously.

• start(Module, Func, Args, Time, SpawnOpts) -> Ret
  [page 249] Start a new process synchronously.

• start_link(Module, Func, Args) -> Ret
  [page 249] Start a new process synchronously.

• start_link(Module, Func, Args, Time) -> Ret
  [page 249] Start a new process synchronously.

• start_link(Module, Func, Args, Time, SpawnOpts) -> Ret
  [page 249] Start a new process synchronously.

• init_ack(Parent, Ret) -> void()
  [page 250] Used by a process when it has started.

• init_ack(Ret) -> void()
  [page 250] Used by a process when it has started.

• format(CrashReport) -> string()

• initial_call(PidOrPinfo) -> {Module, Function, Args} | Fun | false
  [page 251] Extract the initial call of a proc_lib spawned process.

• translate_initial_call(PidOrPinfo) -> {Module, Function, Arity} | Fun
  [page 251] Extract and translate the initial call of a proc_lib spawned process.

• hibernate(Module, Function, Arguments)
  [page 251] Hibernate the current process until a message is sent to it.
proplists

The following functions are exported:

- `append_values(Key, List) -> List`
  [page 253]

- `compact(List) -> List`
  [page 253]

- `delete(Key, List) -> List`
  [page 253]

- `expand(Expansions, List) -> List`
  [page 253]

- `get_all_values(Key, List) -> [term()]`
  [page 254]

- `get_bool(Key, List) -> bool()`
  [page 254]

- `get_keys(List) -> [term()]`
  [page 254]

- `get_value(Key, List) -> term()`
  [page 255]

- `get_value(Key, List, Default) -> term()`
  [page 255]

- `is_defined(Key, List) -> bool()`
  [page 255]

- `lookup(Key, List) -> none | tuple()`
  [page 255]

- `lookup_all(Key, List) -> [tuple()]`
  [page 255]

- `normalize(List, Stages) -> List`
  [page 255]

- `property(Property) -> Property`
  [page 256]

- `property(Key, Value) -> Property`
  [page 256]

- `split(List, Keys) -> {Lists, Rest}`
  [page 256]

- `substitute_aliases(Aliases, List) -> List`
  [page 257]

- `substitute_negations(Negations, List) -> List`
  [page 257]

- `unfold(List) -> List`
  [page 257]
The following functions are exported:

- `append(QHL) -> QH`
  [page 264] Return a query handle.
- `append(QH1, QH2) -> QH3`
  [page 264] Return a query handle.
- `cursor(QueryHandleOrList [, Options]) -> QueryCursor`
  [page 264] Create a query cursor.
- `delete_cursor(QueryCursor) -> ok`
  [page 264] Delete a query cursor.
- `eval(QueryHandleOrList [, Options]) -> Answers | Error`
  [page 264] Return all answers to a query.
- `e(QueryHandleOrList [, Options]) -> Answers`
  [page 264] Return all answers to a query.
- `fold(Function, Acc0, QueryHandleOrList [, Options]) -> Acc1 | Error`
  [page 265] Fold a function over the answers to a query.
- `format_error(Error) -> Chars`
  [page 265] Return an English description of a an error tuple.
- `info(QueryHandleOrList [, Options]) -> Info`
  [page 265] Return code describing a query handle.
- `keysort(KeyPos, QH1 [, SortOptions]) -> QH2`
  [page 266] Return a query handle.
- `next_answers(QueryCursor [, NumberOfAnswers]) -> Answers | Error`
  [page 266] Return some or all answers to a query.
- `q(QueryListComprehension [, Options]) -> QueryHandle`
  [page 267] Return a handle for a query list comprehension.
- `sort(QH1 [, SortOptions]) -> QH2`
  [page 268] Return a query handle.
- `string_to_handle(QueryString [, Options [, Bindings]]]) -> QueryHandle | Error`
  [page 268] Return a handle for a query list comprehension.
- `table(TraverseFun, Options) -> QueryHandle`
  [page 269] Return a query handle for a table.

The following functions are exported:

- `cons(Item, Q1) -> Q2`
  [page 272] Insert an item at the head of a queue
- `daeh(Q) -> Item`
  [page 272] Return the last item of a queue
- `from_list(L) -> queue()`
  [page 272] Convert a list to a queue
- `head(Q) -> Item`
  [page 272] Return the item at the head of a queue
• in(Item, Q1) -> Q2
  [page 272] Insert an item at the tail of a queue

• in_r(Item, Q1) -> Q2
  [page 273] Insert an item at the head of a queue

• init(Q1) -> Q2
  [page 273] Remove the last item from a queue

• is_empty(Q) -> true | false
  [page 273] Test if a queue is empty

• join(Q1, Q2) -> Q3
  [page 273] Join two queues

• lait(Q1) -> Q2
  [page 273] Remove the last item from a queue

• last(Q) -> Item
  [page 273] Return the last item of a queue

• len(Q) -> N
  [page 273] Get the length of a queue

• new() -> Q
  [page 274] Create a new empty FIFO queue

• out(Q1) -> Result
  [page 274] Remove the head item from a queue

• out_r(Q1) -> Result
  [page 274] Remove the last item from a queue

• reverse(Q1) -> Q2
  [page 274] Reverse a queue

• snoc(Q1, Item) -> Q2
  [page 274] Insert an item at the end of a queue

• split(N, Q1) -> {Q2, Q3}
  [page 274] Split a queue in two

• tail(Q1) -> Q2
  [page 274] Remove the head item from a queue

• to_list(Q) -> list()
  [page 275] Convert a queue to a list

random

The following functions are exported:

• seed() -> ran()
  [page 276] Seeds random number generation with default values

• seed(A1, A2, A3) -> ran()
  [page 276] Seeds random number generator

• seed0() -> ran()
  [page 276] Return default state for random number generation

• uniform() -> float()
  [page 276] Return a random float

• uniform(N) -> int()
  [page 276] Return a random integer
• uniform_s(State0) -> {float(), State1}
  [page 277] Return a random float
• uniform_s(N, State0) -> {int(), State1}
  [page 277] Return a random integer

regexp

The following functions are exported:

• match(String, RegExp) -> MatchRes
  [page 278] Match a regular expression
• first_match(String, RegExp) -> MatchRes
  [page 278] Match a regular expression
• matches(String, RegExp) -> MatchRes
  [page 278] Match a regular expression
• sub(String, RegExp, New) -> SubRes
  [page 279] Substitute the first occurrence of a regular expression
• gsub(String, RegExp, New) -> SubRes
  [page 279] Substitute all occurrences of a regular expression
• split(String, RegExp) -> SplitRes
  [page 279] Split a string into fields
• sh_to_awk(ShRegExp) -> AwkRegExp
  [page 280] Convert an sh regular expression into an AWK one
• parse(RegExp) -> ParseRes
  [page 280] Parse a regular expression
• format_error(ErrorDescriptor) -> Chars
  [page 280] Format an error descriptor

sets

The following functions are exported:

• new() -> Set
  [page 283] Return an empty set
• is_set(Set) -> bool()
  [page 283] Test for an Set
• size(Set) -> int()
  [page 283] Return the number of elements in a set
• to_list(Set) -> List
  [page 283] Convert an Set into a list
• from_list(List) -> Set
  [page 283] Convert a list into an Set
• is_element(Element, Set) -> bool()
  [page 283] Test for membership of an Set
• add_element(Element, Set1) -> Set2
  [page 284] Add an element to an Set
• del_element(Element, Set1) -> Set2
  [page 284] Remove an element from an Set
- union(Set1, Set2) -> Set3  
  [page 284] Return the union of two Sets
- union(SetList) -> Set  
  [page 284] Return the union of a list of Sets
- intersection(Set1, Set2) -> Set3  
  [page 284] Return the intersection of two Sets
- intersection(SetList) -> Set  
  [page 284] Return the intersection of a list of Sets
- subtract(Set1, Set2) -> Set3  
  [page 284] Return the difference of two Sets
- is_subset(Set1, Set2) -> bool()  
  [page 285] Test for subset
- fold(Function, Acc0, Set) -> Acc1  
  [page 285] Fold over set elements
- filter(Pred, Set1) -> Set2  
  [page 285] Filter set elements

shell

The following functions are exported:
- history(N) -> integer()  
  [page 294] Sets the number of previous commands to keep
- results(N) -> integer()  
  [page 294] Sets the number of previous commands to keep
- start_restricted(Module) -> ok  
  [page 294] Exits a normal shell and starts a restricted shell.
- stop_restricted() -> ok  
  [page 294] Exits a restricted shell and starts a normal shell.

shell_default

No functions are exported.

slave

The following functions are exported:
- start(Host) ->  
  [page 296] Start a slave node on a host
- start(Host, Name) ->  
  [page 296] Start a slave node on a host
- start(Host, Name, Args) -> {ok, Node} | {error, Reason}  
  [page 296] Start a slave node on a host
- start_link(Host) ->  
  [page 297] Start and link to a slave node on a host
start_link(Host, Name) ->
[page 297] Start and link to a slave node on a host

start_link(Host, Name, Args) -> {ok, Node} | {error, Reason}
[page 297] Start and link to a slave node on a host

stop(Node) -> ok
[page 298] Stop (kill) a node

pseudo([Master | ServerList]) -> ok
[page 298] Start a number of pseudo servers

pseudo(Master, ServerList) -> ok
[page 298] Start a number of pseudo servers

relay(Pid)
[page 298] Run a pseudo server

sofs

The following functions are exported:

- a_function(Tuples [, Type]) -> Function
  [page 303] Create a function.

- canonical_relation(SetOfSets) -> BinRel
  [page 303] Return the canonical map.

- composite(Function1, Function2) -> Function3
  [page 303] Return the composite of two functions.

- constant_function(Set, AnySet) -> Function
  [page 303] Create the function that maps each element of a set onto another set.

- converse(BinRel1) -> BinRel2
  [page 304] Return the converse of a binary relation.

- difference(Set1, Set2) -> Set3
  [page 304] Return the difference of two sets.

- digraph_to_family(Graph [, Type]) -> Family
  [page 304] Create a family from a directed graph.

- domain(BinRel) -> Set
  [page 304] Return the domain of a binary relation.

- drestriction(BinRel1, Set) -> BinRel2
  [page 304] Return a restriction of a binary relation.

- drestriction(SetFun, Set1, Set2) -> Set3
  [page 305] Return a restriction of a relation.

- empty_set() -> Set
  [page 305] Return the untyped empty set.

- extension(BinRel1, Set, AnySet) -> BinRel2
  [page 305] Extend the domain of a binary relation.

- family(Tuples [, Type]) -> Family
  [page 306] Create a family of subsets.

- family_difference(Family1, Family2) -> Family3
  [page 306] Return the difference of two families.

- family_domain(Family1) -> Family2
  [page 306] Return a family of domains.
- **family_field**(Family1) → Family2
  [page 306] Return a family of fields.

- **family_intersection**(Family1) → Family2
  [page 307] Return the intersection of a family of sets of sets.

- **family_intersection**(Family1, Family2) → Family3
  [page 307] Return the intersection of two families.

- **family_projection**(SetFun, Family1) → Family2
  [page 307] Return a family of modified subsets.

- **family_range**(Family1) → Family2
  [page 307] Return a family of ranges.

- **family_specification**(Fun, Family1) → Family2
  [page 308] Select a subset of a family using a predicate.

- **family_to_diagram**(Family [, GraphType]) → Graph
  [page 308] Create a directed graph from a family.

- **family_to_relation**(Family) → BinRel
  [page 308] Create a binary relation from a family.

- **family_union**(Family1) → Family2
  [page 309] Return the union of a family of sets of sets.

- **family_union**(Family1, Family2) → Family3
  [page 309] Return the union of two families.

- **field**(BinRel) → Set
  [page 309] Return the field of a binary relation.

- **from_external**(ExternalSet, Type) → AnySet
  [page 309] Create a set.

- **from_sets**(ListOfSets) → Set
  [page 309] Create a set out of a list of sets.

- **from_sets**(TupleOfSets) → Ordset
  [page 309] Create an ordered set out of a tuple of sets.

- **from_term**(Term [, Type]) → AnySet
  [page 310] Create a set.

- **image**(BinRel, Set1) → Set2
  [page 311] Return the image of a set under a binary relation.

- **intersection**(SetOfSets) → Set
  [page 311] Return the intersection of a set of sets.

- **intersection**(Set1, Set2) → Set3
  [page 311] Return the intersection of two sets.

- **intersection_of_family**(Family) → Set
  [page 311] Return the intersection of a family.

- **inverse**(Function1) → Function2
  [page 311] Return the inverse of a function.

- **inverse_image**(BinRel, Set1) → Set2
  [page 312] Return the inverse image of a set under a binary relation.

- **is_a_function**(BinRel) → Bool
  [page 312] Test for a function.

- **is_disjoint**(Set1, Set2) → Bool
  [page 312] Test for disjoint sets.
- `is_empty_set(AnySet) -> Bool` [page 312] Test for an empty set.
- `is_equal(AnySet1, AnySet2) -> Bool` [page 312] Test two sets for equality.
- `is_set(AnySet) -> Bool` [page 312] Test for an unordered set.
- `is_sofs_set(Term) -> Bool` [page 313] Test for an unordered set.
- `is_subset(Set1, Set2) -> Bool` [page 313] Test two sets for subset.
- `is_type(Term) -> Bool` [page 313] Test for a type.
- `join(Relation1, I, Relation2, J) -> Relation3` [page 313] Return the join of two relations.
- `multiple_relative_product(TupleOfBinRels, BinRel1) -> BinRel2` [page 313] Return the multiple relative product of a tuple of binary relations and a relation.
- `no_elements(ASet) -> NoElements` [page 314] Return the number of elements of a set.
- `partition(SetOfSets) -> Partition` [page 314] Return the coarsest partition given a set of sets.
- `partition(SetFun, Set) -> Partition` [page 314] Return a partition of a set.
- `partition(SetFun, Set1, Set2) -> {Set3, Set4}` [page 314] Return a partition of a set.
- `partition_family(SetFun, Set) -> Family` [page 315] Return a family indexing a partition.
- `product(Set1, Set2) -> BinRel` [page 315] Return the Cartesian product of two sets.
- `projection(SetFun, Set1) -> Set2` [page 316] Return a set of substituted elements.
- `range(BinRel) -> Set` [page 316] Return the range of a binary relation.
- `relation(Tuples [, Type]) -> Relation` [page 316] Create a relation.
- `relation_to_family(BinRel) -> Family` [page 316] Create a family from a binary relation.
- `relative_product(TupleOfBinRels [, BinRel1]) -> BinRel2` [page 317] Return the relative product of a tuple of binary relations and a binary relation.
- `relative_product(BinRel1, BinRel2) -> BinRel3` [page 317] Return the relative product of two binary relations.
- `relative_product1(BinRel1, BinRel2) -> BinRel3` [page 317] Return the relative product of two binary relations.
• restriction(BinRel1, Set) -> BinRel2
  [page 317] Return a restriction of a binary relation.

• restriction(SetFun, Set1, Set2) -> Set3
  [page 318] Return a restriction of a set.

• set(Terms [, Type]) -> Set
  [page 318] Create a set of atoms or any type of sets.

• specification(Fun, Set1) -> Set2
  [page 318] Select a subset using a predicate.

• strict_relation(BinRel1) -> BinRel2
  [page 318] Return the strict relation corresponding to a given relation.

• substitution(SetFun, Set1) -> Set2
  [page 319] Return a function with a given set as domain.

• symdiff(Set1, Set2) -> Set3
  [page 319] Return the symmetric difference of two sets.

• symmetric_partition(Set1, Set2) -> {Set3, Set4, Set5}
  [page 320] Return a partition of two sets.

• to_external(AnySet) -> ExternalSet
  [page 320] Return the elements of a set.

• to_sets(ASet) -> Sets
  [page 320] Return a list or a tuple of the elements of set.

• type(AnySet) -> Type
  [page 320] Return the type of a set.

• union(SetOfSets) -> Set
  [page 320] Return the union of a set of sets.

• union(Set1, Set2) -> Set3
  [page 320] Return the union of two sets.

• union_of_family(Family) -> Set
  [page 321] Return the union of a family.

• weak_relation(BinRel1) -> BinRel2
  [page 321] Return the weak relation corresponding to a given relation.

string

The following functions are exported:

• len(String) -> Length
  [page 322] Return the length of a string.

• equal(String1, String2) -> bool()
  [page 322] Test string equality.

• concat(String1, String2) -> String3
  [page 322] Concatenate two strings.

• chr(String, Character) -> Index
  [page 322] Return the index of the first/last occurrence of Character in String.

• rchr(String, Character) -> Index
  [page 322] Return the index of the first/last occurrence of Character in String.

• str(String, SubString) -> Index
  [page 322] Find the index of a substring.
STDLIB Reference Manual

- `rstr(String, SubString) -> Index`
  [page 322] Find the index of a substring
- `span(String, Chars) -> Length`
  [page 323] Span characters at start of string
- `cspan(String, Chars) -> Length`
  [page 323] Span characters at start of string
- `substr(String, Start) -> SubString`
  [page 323] Return a substring of String
- `substr(String, Start, Length) -> Substring`
  [page 323] Return a substring of string
- `tokens(String, SeparatorList) -> Tokens`
  [page 323] Split string into tokens
- `chars(Character, Number) -> String`
  [page 323] Returns a string consisting of numbers of characters
- `chars(Character, Number, Tail) -> String`
  [page 323] Returns a string consisting of numbers of characters
- `copies(String, Number) -> Copies`
  [page 324] Copy a string
- `words(String) -> Count`
  [page 324] Count blank separated words
- `words(String, Character) -> Count`
  [page 324] Count blank separated words
- `subword(String, Number) -> Word`
  [page 324] Extract subword
- `subword(String, Number, Character) -> Word`
  [page 324] Extract subword
- `strip(String) -> Stripped`
  [page 324] Strip leading or trailing characters
- `strip(String, Direction) -> Stripped`
  [page 324] Strip leading or trailing characters
- `strip(String, Direction, Character) -> Stripped`
  [page 324] Strip leading or trailing characters
- `left(String, Number) -> Left`
  [page 325] Adjust left end of string
- `left(String, Number, Character) -> Left`
  [page 325] Adjust left end of string
- `right(String, Number) -> Right`
  [page 325] Adjust right end of string
- `right(String, Number, Character) -> Right`
  [page 325] Adjust right end of string
- `centre(String, Number) -> Centered`
  [page 325] Center a string
- `centre(String, Number, Character) -> Centered`
  [page 325] Center a string
- `sub_string(String, Start) -> SubString`
  [page 325] Extract a substring
- **substring(String, Start, Stop) -> SubString**  
  [page 326] Extract a substring

- **to_float(String) -> {Float,Rest} | {error,Reason}**  
  [page 326] Returns a float whose text representation is the integers (ASCII values) in String.

- **to_integer(String) -> {Int,Rest} | {error,Reason}**  
  [page 326] Returns an integer whose text representation is the integers (ASCII values) in String.

### supervisor

The following functions are exported:

- **start_link(Module, Args) -> Result**  
  [page 330] Create a supervisor process

- **start_link(SupName, Module, Args) -> Result**  
  [page 330] Create a supervisor process

- **start_child(SupRef, ChildSpec) -> Result**  
  [page 331] Dynamically add a child process to a supervisor.

- **terminate_child(SupRef, Id) -> Result**  
  [page 331] Terminate a child process belonging to a supervisor.

- **delete_child(SupRef, Id) -> Result**  
  [page 332] Delete a child specification from a supervisor.

- **restart_child(SupRef, Id) -> Result**  
  [page 332] Restart a terminated child process belonging to a supervisor.

- **which_children(SupRef) -> [{Id,Child,Type,Modules}]**  
  [page 333] Return information about all children specifications and child processes belonging to a supervisor.

- **check_childspecs([ChildSpec]) -> Result**  
  [page 333] Check if child specifications are syntactically correct.

- **Module:init(Args) -> Result**  
  [page 334] Return a supervisor specification.

### supervisor_bridge

The following functions are exported:

- **start_link(Module, Args) -> Result**  
  [page 335] Create a supervisor bridge process

- **start_link(SupBridgeName, Module, Args) -> Result**  
  [page 335] Create a supervisor bridge process

- **Module:init(Args) -> Result**  
  [page 336] Initialize process and start subsystem.

- **Module:terminate(Reason, State)**  
  [page 336] Clean up and stop subsystem.
sys

The following functions are exported:

- `log(Name,Flag)`  
  [page 339] Log system events in memory

- `log(Name,Flag,Timeout) -> ok | [system_event()]`  
  [page 339] Log system events in memory

- `log_to_file(Name,Flag)`  
  [page 339] Log system events to the specified file

- `log_to_file(Name,Flag,Timeout) -> ok | {error, open_file}`  
  [page 339] Log system events to the specified file

- `statistics(Name,Flag)`  
  [page 339] Enable or disable the collections of statistics

- `statistics(Name,Flag,Timeout) -> ok | {ok, Statistics}`  
  [page 339] Enable or disable the collections of statistics

- `trace(Name,Flag)`  
  [page 340] Print all system events on standard_io

- `trace{Name,Flag,Timeout) -> void()`  
  [page 340] Print all system events on standard_io

- `no_debug{Name)`  
  [page 340] Turn off debugging

- `no_debug{Name,Timeout) -> void()`  
  [page 340] Turn off debugging

- `suspend{Name)`  
  [page 340] Suspend the process

- `suspend{Name,Timeout) -> void()`  
  [page 340] Suspend the process

- `resume{Name)`  
  [page 340] Resume a suspended process

- `resume{Name,Timeout) -> void()`  
  [page 340] Resume a suspended process

- `change_code{Name, Module, OldVsn, Extra)`  
  [page 340] Send the code change system message to the process

- `change_code{Name, Module, OldVsn, Extra, Timeout) -> ok | {error, Reason}`  
  [page 340] Send the code change system message to the process

- `get_status{Name)`  
  [page 340] Get the status of the process

- `get_status{Name,Timeout) -> {status, Pid, {module, Mod}, [PDict, SysState, Parent, Dbg, Misc]}`  
  [page 340] Get the status of the process

- `install{Name,Func,FuncState)`  
  [page 341] Install a debug function in the process

- `install{Name,Func,FuncState},Timeout)`  
  [page 341] Install a debug function in the process

- `remove{Name,Func)`  
  [page 341] Remove a debug function from the process
- `remove(Name, Func, Timeout) -> void()`
  [page 341] Remove a debug function from the process

- `debug_options(Options) -> [dbg_opt()]`
  [page 342] Convert a list of options to a debug structure

- `get_debug(Item, Debug, Default) -> term()`
  [page 342] Get the data associated with a debug option

- `handle_debug([dbg_opt()], FormFunc, Extra, Event) -> [dbg_opt()]`
  [page 342] Generate a system event

- `handle_system_msg(Msg, From, Parent, Module, Debug, Misc)`
  [page 342] Take care of system messages

- `print_log(Debug) -> void()`
  [page 343] Print the logged events in the debug structure

- `Mod:system_continue(Parent, Debug, Misc)`
  [page 343] Called when the process should continue its execution

- `Mod:system_terminate(Reason, Parent, Debug, Misc)`
  [page 343] Called when the process should terminate

- `Mod:system_code_change(Misc, Module, OldVsn, Extra) -> {ok, NMisc}`
  [page 343] Called when the process should perform a code change

**timer**

The following functions are exported:

- `start() -> ok`
  [page 345] Start a global timer server (named `timer_server`).

- `apply_after(Time, Module, Function, Arguments) -> {ok, Tref} | {error, Reason}`

- `send_after(Time, Pid, Message) -> {ok, TRef} | {error, Reason}`
  [page 345] Send Message to Pid after a specified Time.

- `send_after(Time, Pid, Message) -> {ok, TRef} | {error, Reason}`
  [page 345] Send Message to Pid after a specified Time.

- `exit_after(Time, Pid, Reason1) -> {ok, TRef} | {error, Reason2}`
  [page 346] Send an exit signal with Reason after a specified Time.

- `exit_after(Time, Reason1) -> {ok, TRef} | {error, Reason2}`
  [page 346] Send an exit signal with Reason after a specified Time.

- `kill_after(Time, Pid) -> {ok, TRef} | {error, Reason2}`
  [page 346] Send an exit signal with Reason after a specified Time.

- `kill_after(Time) -> {ok, TRef} | {error, Reason2}`
  [page 346] Send an exit signal with Reason after a specified Time.

- `apply_interval(Time, Module, Function, Arguments) -> {ok, TRef} | {error, Reason}`

- `send_interval(Time, Pid, Message) -> {ok, TRef} | {error, Reason}`
  [page 346] Send Message repeatedly at intervals of Time.
The following functions are exported:

- `change_key(RegHandle, Key) -> ReturnValue`
  Move to a key in the registry.

- `change_key_create(RegHandle, Key) -> ReturnValue`
  Move to a key, create it if it is not there.

- `close(RegHandle) -> ReturnValue`
  Close the registry.

- `current_key(RegHandle) -> ReturnValue`
  Return the path to the current key.

- `delete_key(RegHandle) -> ReturnValue`
  Delete the current key.

- `delete_value(RegHandle, Name) -> ReturnValue`
  Delete the named value on the current key.

- `expand(String) -> ExpandedString`
  Expand a string with environment variables.

- `format_error(ErrorId) -> ErrorString`
  Convert a POSIX errorcode to a string.

- `open(OpenModeList) -> ReturnValue`
  Open the registry for reading or writing.

- `set_value(RegHandle, Name, Value) -> ReturnValue`
  Set value at the current registry key with specified name.

- `sub_keys(RegHandle) -> ReturnValue`
  Get subkeys to the current key.
• `value(RegHandle, Name) -> ReturnValue`
  [page 352] Get the named value on the current key.

• `values(RegHandle) -> ReturnValue`
  [page 352] Get all values on the current key.
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")
- labeledlocals ("LocT")
- atoms ("Atom")

ENCRYPTED ABSTRACT CODE

The abstract code can be encrypted in order to keep the source code secret, but still be able to use tools such as Xref or Debugger. See [compile(3)] for how to encrypt the abstract code.

To enable tools to use the abstract code, the key must be made available for beam_lib. There are two ways to provide the key:

1) Use the function crypto_key_fun/1 [page 55] to register a fun that will be called whenever beam_lib needs to decrypt the abstract code.

2) Store the key in a text file named `.erlang.crypt` located in either the current directory or the home directory for the current user. beam_lib will search for and read the `.erlang.crypt` file if no crypto fun has been registered using crypto_key_fun/1. If the file exists 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. The elements of the list should be tuples looking like this:

{debug_info, Mode, Module, Key}
Mode is the type of key; currently, the only allowed value 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 a non-empty 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$MwL#&_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$MwL#&_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

chucks(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.

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]]}

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()

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 = 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 abstract code 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 abstract code 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 = 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

```erlang
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)`: Called when the fun is registered, in the process that holds the fun. Here the crypto key fun can do any necessary initializations. Allowed return values:
  - `ok`
  - `{ok, NewCryptoKeyFun}` - NewCryptoKeyFun will be registered instead of CryptoKeyFun.
  - `{error, Term}` - the registration will be aborted and `crypto_key_fun/1` will return `{error, Term}`, where Term can be any term.

- `CryptoKeyFun({debug_info, Mode, Module, Filename})`: Called when the key is needed for module `Module` in the file `Filename`. Mode describes the method of encryption; currently the only allowed value is `des3_cbc`. The fun should return the key as list of characters, or fail if there is no key available.

- `CryptoKeyFun(clear)`: Called just before the fun is unregistered. Do any needed cleanup here. The return value is not important, but will be passed back to the caller of `clear_crypto_key_fun/0`.

### 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 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 = Filename | Module`
  - `Filename = string() | atom()`
  - `Options = [Opt] – see compile:file/2`
  - `Module = atom()`
  `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 = string() | atom()`
  Changes working directory to `Dir`, which may be a relative name, and then prints the name of the new working directory.
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()
i() 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 = Filename | Module
   • Filename = string() | atom()
   • Module = atom()
   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 = string() | atom()
  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 = Filename | Module
  • Filename = string() | atom()
  • Options = [Opt] - see compile:file/2
  • Module = atom()
  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.

\texttt{pwd()} \rightarrow \texttt{void()}

Prints the name of the working directory.

\texttt{q()} \rightarrow \texttt{void()}

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

\texttt{regs()} \rightarrow \texttt{void()}
\texttt{nregs()} \rightarrow \texttt{void()}

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

\texttt{xm(ModSpec)} \rightarrow \texttt{void()}

Types:
- \texttt{ModSpec = Module | Filename}
- \texttt{Module = atom()}
- \texttt{Filename = string()}

This function finds undefined functions and unused functions in a module by calling \texttt{xref:m/1}.

\textbf{See Also}

\texttt{erlang(3)}
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/0`. 
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_daytime(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 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 equal 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.

```erlang
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.

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

Types:
- `Name = name()`

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

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

Types:
- `Name = name()`

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

```erlang
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.

```erlang
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**.

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.

\[
\text{insert(Name, Objects)} \rightarrow \text{ok | \{error, Reason\}}
\]

Types:
- **Name** = name()
- **Objects** = object() | [object()]

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

\[
\text{insert\_new(Name, Objects)} \rightarrow \text{Bool}
\]
Types:
- Name = name()
- Objects = object() | [object()]
- Bool = bool()

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

\[\text{is}\_\text{compatible}\_bchunk\_\text{format}(\text{Name}, \text{BchunkFormat}) \rightarrow \text{Bool}\]

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.

\[\text{is}\_\text{dets}\_\text{file}(\text{FileName}) \rightarrow \text{Bool} | \{\text{error}, \text{Reason}\}\]

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

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

\[\text{lookup}(\text{Name}, \text{Key}) \rightarrow \text{[Object]} | \{\text{error}, \text{Reason}\}\]

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

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

\[
\begin{align*}
2> & \text{dets:open_file}(\text{abc}, [\{\text{type, bag}\}]).
\{\text{ok,abc}\} \\
3> & \text{dets:insert}(\text{abc}, \{1,2,3\}). \quad \text{ok} \\
4> & \text{dets:insert}(\text{abc}, \{1,3,4\}). \quad \text{ok} \\
5> & \text{dets:lookup}(\text{abc}, 1). \quad \{\{1,2,3\}, \{1,3,4\}\}
\end{align*}
\]

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.

\[\text{match}(\text{Continuation}) \rightarrow \{[\text{Match}], \text{Continuation2}\} | \{\text{end_of_table}\} | \{\text{error}, \text{Reason}\}\]

Types:
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 122] 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 list of the bindings that match Pattern in some unspecified order. See ets(3) [page 122] 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. 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 122] 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 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 122] 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 \texttt{Name} and returns a list of the objects that match \texttt{Pattern} in some unspecified order. See \texttt{ets(3)} [page 122] for a description of patterns.

A list of objects and a continuation is returned, unless the table is empty, in which case \texttt{'$end\_of\_table'} is returned. The continuation is to be used when matching further objects by calling \texttt{match\_object/1}.

If the keypos'th element of \texttt{Pattern} is bound, all objects of the table are matched. If the keypos'th element is unbound, all objects of the table are matched, \texttt{N} objects at a time. The default, indicated by giving \texttt{N} the value \texttt{default}, is to let the number of objects vary depending on the sizes of the objects. If \texttt{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 \texttt{N} objects may sometimes be returned.

The table should always be protected using \texttt{safe\_fixtable/2} before calling \texttt{match\_object/3}, or errors may occur when calling \texttt{match\_object/1}.

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

Types:
- \texttt{Name = name()}
- \texttt{Key = term()}
- \texttt{Bool = bool()}

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

\texttt{next(Name, Key1) -> Key2 | '$end\_of\_table'}

Types:
- \texttt{Name = name()}
- \texttt{Key1 = Key2 = term()}

Returns the key following \texttt{Key1} in the table \texttt{Name} according to the table's internal order, or \texttt{'$end\_of\_table'} if there is no next key.

Should an error occur, the process is exited with an error tuple \texttt{\{error, Reason\}}. Use \texttt{first/1} to find the first key in the table.

\texttt{open\_file(Filename) -> \{ok, Reference\} | \{error, Reason\}}

Types:
- \texttt{Filename = file()}
- \texttt{Reference = ref()}

Opens an existing table. If the table has not been properly closed, the error \texttt{\{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.

\texttt{open\_file(Name, Args) -> \{ok, Name\} | \{error, Reason\}}

Types:
- \texttt{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 processes 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()}`, A 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 implemented 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()]`

Returns the results of applying a match specification to some objects stored in a table. 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 \texttt{MatchSpec} to some or all objects stored in the table \texttt{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 \texttt{select}/1.

If the \texttt{keypos}'th element of \texttt{MatchSpec} is bound, the match specification is applied to all objects of the table with the right key(s). If the \texttt{keypos}'th element of \texttt{MatchSpec} is unbound, the match specification is applied to all objects of the table, \texttt{N} objects at a time. The default, indicated by giving \texttt{N} the value \texttt{default}, is to let the number of objects vary depending on the sizes of the objects. If \texttt{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 \texttt{N} objects.

The table should always be protected using \texttt{safe \_ fixtable}/2 before calling \texttt{select}/3, or errors may occur when calling \texttt{select}/1.

\begin{verbatim}
select \_ delete (Name, MatchSpec) -> N | \{error, Reason\}
\end{verbatim}

\textbf{Types:}
- \texttt{Name} = \texttt{name()}
- \texttt{MatchSpec} = \texttt{match \_ spec()}
- \texttt{N} = \texttt{int()}

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

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

\begin{verbatim}
slot (Name, I) -> '$end \_ of \_ table$' | [Object] | \{error, Reason\}
\end{verbatim}

\textbf{Types:}
- \texttt{Name} = \texttt{name()}
- \texttt{I} = \texttt{int()}
- \texttt{Object} = \texttt{object()}

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

\begin{verbatim}
sync (Name) -> ok | \{error, Reason\}
\end{verbatim}

\textbf{Types:}
- \texttt{Name} = \texttt{name()}

Ensures that all updates made to the table \texttt{Name} are written to disk. This also applies to tables which have been opened with the \texttt{ram \_ file} flag set to \texttt{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.

\begin{verbatim}
table (Name [, Options]) -> QueryHandle
\end{verbatim}
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:

```erlang
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:

```erlang
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`:

```erlang
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.
• 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 function 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 122], mnesia(3), qlc(3) [page 258]
dict

Erlang Module

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

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()
map(Fun, Dict1) -> Dict2

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

map calls Fun 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.

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 162], orddict(3) [page 242]
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 \(v[1], v[2], \ldots, v[k]\) of vertices in \(V\) such that there is an edge \((v[i], v[i+1])\) in \(E\) for \(1 \leq i < k\). 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

\[
\text{add\_edge}(G, E, V1, V2, \text{Label}) \rightarrow \text{edge()} \mid \{\text{error, Reason}\}
\]
\[
\text{add\_edge}(G, V1, V2, \text{Label}) \rightarrow \text{edge()} \mid \{\text{error, Reason}\}
\]
\[
\text{add\_edge}(G, V1, V2) \rightarrow \text{edge()} \mid \{\text{error, Reason}\}
\]

Types:
- \(G = \text{digraph()}\)
- \(E = \text{edge()}\)
- \(V1 = V2 = \text{vertex()}\)
- \(\text{Label} = \text{label()}\)
- \(\text{Reason} = \{\text{bad\_edge, Path}\} \mid \{\text{bad\_vertex, V}\}\)
- \(\text{Path} = [\text{vertex()}]\)
add_edge/5 creates (or modifies) the edge \( E \) of the digraph \( G \), using Label as the (new) label [page 87] of the edge. The edge is emanating [page 87] from \( V_1 \) and incident [page 87] on \( V_2 \). Returns \( E \).

\( \text{add_edge}(G, V_1, V_2, \text{Label}) \) is equivalent to \( \text{add_edge}(G, E, V_1, V_2, \text{Label}) \), where \( E \) is a created edge. Tuples on the form \(['$e'|N]\), where \( N \) is an integer \( \geq 1 \), are used for representing the created edges.

\( \text{add_edge}(G, V_1, V_2) \) is equivalent to \( \text{add_edge}(G, V_1, V_2, []) \).

If the edge would create a cycle in an acyclic digraph [page 87], then \( \{\text{error}, \{\text{bad_edge}, \text{Path}\}\} \) is returned. If either of \( V_1 \) or \( V_2 \) is not a vertex of the digraph \( G \), then \( \{\text{error}, \{\text{bad_vertex}, V\}\} \) is returned, \( V = V_1 \) or \( V = V_2 \).

\( \text{add_vertex}(G, V, \text{Label}) \) -> \( \text{vertex}() \)
\( \text{add_vertex}(G, V) \) -> \( \text{vertex}() \)
\( \text{add_vertex}(G) \) -> \( \text{vertex}() \)

Types:
- \( G = \text{digraph}() \)
- \( V = \text{vertex}() \)
- \( \text{Label} = \text{label}() \)

\( \text{add_vertex}/3 \) creates (or modifies) the vertex \( V \) of the digraph \( G \), using Label as the (new) label [page 87] of the vertex. Returns \( V \).

\( \text{add_vertex}(G, V) \) is equivalent to \( \text{add_vertex}(G, V, []) \).

\( \text{add_vertex}/1 \) 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 \( \geq 1 \), are used for representing the created vertices.

\( \text{del_edge}(G, E) \) -> \( \text{true} \)

Types:
- \( G = \text{digraph}() \)
- \( E = \text{edge}() \)

Deletes the edge \( E \) from the digraph \( G \).

\( \text{del_edges}(G, \text{Edges}) \) -> \( \text{true} \)

Types:
- \( G = \text{digraph}() \)
- \( \text{Edges} = [\text{edge}()] \)

Deletes the edges in the list \( \text{Edges} \) from the digraph \( G \).

\( \text{del_path}(G, V_1, V_2) \) -> \( \text{true} \)

Types:
- \( G = \text{digraph}() \)
- \( V_1 = V_2 = \text{vertex}() \)
Deletes edges from the digraph \( G \) until there are no paths [page 87] from the vertex \( V_1 \) to the vertex \( V_2 \).

A sketch of the procedure employed: Find an arbitrary simple path [page 87] \( v[1], v[2], ..., v[k] \) from \( V_1 \) to \( V_2 \) in \( G \). Remove all edges of \( G \) emanating [page 87] from \( v[i] \) and incident [page 87] to \( v[i+1] \) for \( 1 \leq i < k \) (including multiple edges). Repeat until there is no path between \( V_1 \) and \( V_2 \).

\[\text{del_vertex}(G, V) \rightarrow \text{true}\]

Types:
- \( G = \text{digraph}() \)
- \( V = \text{vertex}() \)

Deletes the vertex \( V \) from the digraph \( G \). Any edges emanating [page 87] from \( V \) or incident [page 87] on \( V \) are also deleted.

\[\text{del_vertices}(G, V) \rightarrow \text{true}\]

Types:
- \( G = \text{digraph}() \)
- \( V = \text{vertex}() \)

Deletes the vertices in the list \( V \) from the digraph \( G \).

\[\text{delete}(G) \rightarrow \text{true}\]

Types:
- \( G = \text{digraph}() \)

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.

\[\text{edge}(G, E) \rightarrow \{E, V_1, V_2, Label\} \mid \text{false}\]

Types:
- \( G = \text{digraph}() \)
- \( E = \text{edge}() \)
- \( V_1 = V_2 = \text{vertex}() \)
- \( Label = \text{label}() \)

Returns \( \{E, V_1, V_2, Label\} \) where \( Label \) is the label [page 87] of the edge \( E \) emanating [page 87] from \( V_1 \) and incident [page 87] on \( V_2 \) of the digraph \( G \). If there is no edge \( E \) of the digraph \( G \), then \text{false} is returned.

\[\text{edges}(G) \rightarrow \text{Edges}\]

Types:
- \( G = \text{digraph}() \)
- \( Edges = [\text{edge}()] \)

Returns a list of all edges of the digraph \( G \), in some unspecified order.

\[\text{edges}(G, V) \rightarrow \text{Edges}\]
get_cycle(G, V) -> Vertices | false

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

If there is a simple cycle [page 87] 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 87] 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 87] 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 87] 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 87] 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 87] from the vertex \( V_1 \) to the vertex \( V_2 \) of the digraph \( G \). Returns the path as a list \([V_1, \ldots, V_2]\) of vertices, or false if no simple path from \( V_1 \) to \( V_2 \) of length one or more exists.
The digraph \( G \) is traversed in a breadth-first manner, and the first path found is returned.

\[
in\text{degree}(G, V) \rightarrow \text{integer}()
\]
Types:
- \( G = \text{digraph}() \)
- \( V = \text{vertex}() \)
Returns the in-degree [page 87] of the vertex \( V \) of the digraph \( G \).

\[
in\text{edges}(G, V) \rightarrow \text{Edges}
\]
Types:
- \( G = \text{digraph}() \)
- \( V = \text{vertex}() \)
- \( \text{Edges} = [\text{edge}()] \)
Returns a list of all edges incident [page 87] on \( V \) of the digraph \( G \), in some unspecified order.

\[
in\text{neighbours}(G, V) \rightarrow \text{Vertices}
\]
Types:
- \( G = \text{digraph}() \)
- \( V = \text{vertex}() \)
- \( \text{Vertices} = [\text{vertex}()] \)
Returns a list of all in-neighbours [page 87] of \( V \) of the digraph \( G \), in some unspecified order.

\[
\text{info}(G) \rightarrow \text{InfoList}
\]
Types:
- \( G = \text{digraph}() \)
- \( \text{InfoList} = [\{\text{cyclicity, Cyclicity}\}, \{\text{memory, NoWords}\}, \{\text{protection, Protection}\}] \)
- \( \text{Cyclicity} = \text{cyclic} \mid \text{acyclic} \)
- \( \text{Protection} = \text{public} \mid \text{protected} \mid \text{private} \)
- \( \text{NoWords} = \text{integer}() \geq 0 \)
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 public, protected or private, according to the options given to new.
new() -> digraph()
   Equivalent to new([]).

def new(Type) -> digraph() | {error, Reason}
Types:
   • Type = [cyclic | acyclic | public | private | protected]
   • Reason = {unknown_type, term()}
Returns an empty digraph [page 87] with properties according to the options in Type:

cyclic   Allow cycles [page 87] in the digraph (default).
acyclic  The digraph is to be kept acyclic [page 87].
public    The digraph may be read and modified by any process.
protected Other processes can only 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.

def no_edges(G) -> integer() >= 0
Types:
   • G = digraph()
Returns the number of edges of the digraph G.

def no_vertices(G) -> integer() >= 0
Types:
   • G = digraph()
Returns the number of vertices of the digraph G.

def out_degree(G, V) -> integer()
Types:
   • G = digraph()
   • V = vertex()
Returns the out-degree [page 87] of the vertex V of the digraph G.

def out_edges(G, V) -> Edges
Types:
   • G = digraph()
   • V = vertex()
   • Edges = [edge()]
Returns a list of all edges emanating [page 87] from V of the digraph G, in some unspecified order.

def out_neighbours(G, V) -> Vertices
Types:
• \( G = \text{digraph}() \)
• \( V = \text{vertex}() \)
• \( \text{Vertices} = \{\text{vertex}()\} \)

Returns a list of all out-neighbours [page 87] of \( V \) of the digraph \( G \), in some unspecified order.

\[
\text{vertex}(G, V) \rightarrow \{V, \text{Label}\} \mid \text{false}
\]

**Types:**
• \( G = \text{digraph}() \)
• \( V = \text{vertex}() \)
• \( \text{Label} = \text{label}() \)

Returns \( \{V, \text{Label}\} \) where \( \text{Label} \) is the label [page 87] of the vertex \( V \) of the digraph \( G \), or \( \text{false} \) if there is no vertex \( V \) of the digraph \( G \).

\[
\text{vertices}(G) \rightarrow \text{Vertices}
\]

**Types:**
• \( G = \text{digraph}() \)
• \( \text{Vertices} = \{\text{vertex}()\} \)

Returns a list of all vertices of the digraph \( G \), in some unspecified order.

**See Also**

digraph.util(3) [page 94], ets(3) [page 122]
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 \(v\). 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 \(1 \leq i < k\). 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.
Exports

components(Digraph) -> [Component]

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

Returns a list of connected components [page 94]. 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 94] 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 94] from x and incident [page 94] 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 94] [].

Each and every cycle [page 94] is included in some strongly connected component, which implies that there always exists a topological ordering [page 94] of the created digraph.

cyclic_strong_components(Digraph) -> [StrongComponent]

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

Returns a list of strongly connected components [page 94]. 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 94] 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 94].

loop_vertices(Digraph) -> Vertices

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

Returns a list of all vertices of Digraph that are included in some loop [page 94].
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 94] 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 94] 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 94] 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 94] 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 94] 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 94] 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 94) 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 94) are returned.

strong_components(Digraph) -> [StrongComponent]

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

Returns a list of strongly connected components (page 94). 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 94) 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 94) 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 94) 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 94) that occur earlier in the list.

See Also
digraph(3) [page 87]
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()

0 pens 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 \texttt{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:

\[
\text{ErrorLine, Module, ErrorDescriptor}
\]

A string which describes the error is obtained with the following call:

apply(Module, format\_error, ErrorDescriptor)

See Also

erl\_parse(3) [page 108]
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_term/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_term/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_term/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_term/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_term/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. Exceptions are function calls in guard tests and 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.

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_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

\[\text{parse_transform(Forms, Options)} \rightarrow \text{Forms}\]

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.

\textbf{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 108], 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
- 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

\[
\text{module(AbsForms)} \rightarrow \{\text{ok}, \text{Warnings}\} \mid \{\text{error}, \text{Errors}, \text{Warnings}\} \\
\text{module(AbsForms, FileName)} \rightarrow \{\text{ok}, \text{Warnings}\} \mid \{\text{error}, \text{Errors}, \text{Warnings}\} \\
\text{module(AbsForms, FileName, CompileOptions)} \rightarrow \{\text{ok}, \text{Warnings}\} \mid \{\text{error}, \text{Errors}, \text{Warnings}\}
\]

Types:

- AbsForms = [term()]
- FileName = FileName2 = atom() \mid \text{string()}
- 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 maintainance nightmare, the elements of Options that control the warnings are only described in the manual page for the compile module.

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 98], or parser erl_parse(3) [page 108]). The warnings and errors returned have the following format:

```erlang
{{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.

```erlang
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.

```erlang
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:

```erlang
{ErrorLine, Module, ErrorDescriptor}
```

A string which describes the error is obtained with the following call:

```erlang
apply(Module, format_error, ErrorDescriptor)
```

### See Also

erl_parse(3) [page 108], epp(3) [page 98]
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.

\texttt{format\_error(\texttt{ErrorDescriptor}) \rightarrow Chars}

Types:
- ErrorDescriptor = error\_desc()
- 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).

\texttt{tokens(AbsTerm) \rightarrow Tokens}
\texttt{tokens(AbsTerm, MoreTokens) \rightarrow 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.

\texttt{normalise(AbsTerm) \rightarrow 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 \texttt{abstract/1}.

\texttt{abstract(Data) \rightarrow AbsTerm}

Types:
- Data = AbsTerm = term()

Converts the Erlang data structure Data into an abstract form of type AbsTerm. This is the inverse of \texttt{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 197], `erl_scan(3)` [page 114], 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.

eprs(Expressions) -> DeepCharList
eprs(Expressions, HookFunction) -> DeepCharList
eprs(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.

epr(Expression) -> DeepCharList
epr(Expression, HookFunction) -> DeepCharList
epr(Expression, Indent, HookFunction) -> DeepCharList
epr(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 197], erl_parse(3) [page 108], erl_eval(3) [page 100]
erl_scan

Erlang Module

This module contains functions for tokenizing characters into Erlang tokens.

Exports

\[
\text{string(CharList,StartLine)} \rightarrow \{\text{ok, Tokens, EndLine}\} \mid \text{Error}
\]

\[
\text{string(CharList)} \rightarrow \{\text{ok, Tokens, EndLine}\} \mid \text{Error}
\]

Types:
- \(\text{CharList} = \text{string()}\)
- \(\text{StartLine} = \text{EndLine} = \text{Line} = \text{integer()}\)
- \(\text{Tokens} = \{\text{atom()},\text{Line}\} \mid \{\text{atom()},\text{Line},\text{term()}\}\)
- \(\text{Error} = \{\text{error, ErrorInfo, EndLine}\}\)

Takes the list of characters \(\text{CharList}\) and tries to scan (tokenize) them. Returns \(\{\text{ok, Tokens, EndLine}\}\), where Tokens are the Erlang tokens from \(\text{CharList}\). EndLine is the last line where a token was found.

StartLine indicates the initial line when scanning starts. \text{string/1} is equivalent to \text{string(CharList,1)}.

\{\text{error, ErrorInfo, EndLine}\} is returned if an error occurs. EndLine indicates where the error occurred.

\[
\text{tokens(Continuation, CharList, StartLine)} \rightarrow \text{Return}
\]

Types:
- \(\text{Return} = \{\text{done, Result, LeftOverChars}\} \mid \{\text{more, Continuation}\}\)
- \(\text{Continuation} = [\text{}] \mid \text{string()}\)
- \(\text{CharList} = \text{string()}\)
- \(\text{StartLine} = \text{EndLine} = \text{integer()}\)
- \(\text{Result} = \{\text{ok, Tokens, EndLine}\} \mid \{\text{eof, EndLine}\}\)
- \(\text{Tokens} = \{\text{atom()},\text{Line}\} \mid \{\text{atom()},\text{Line},\text{term()}\}\)

This is the re-entrant scanner which scans characters until a dot ('.') whitespace) has been reached. It returns:

\{\text{done, Result, LeftOverChars}\} This return indicates that there is sufficient input data to get an input. Result is:

\{\text{ok, Tokens, EndLine}\} The scanning was successful. Tokens is the list of tokens including dot.

\{\text{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(Errordesc) -> string()
Types:
- Errordesc = errordesc()
Takes an Errordesc 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 197], erl_parse(3) [page 108]
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 118] or create/3 [page 118] function.

Alternatively, for more control, the open [page 120], add/3,4 [page 117], and close/1 [page 117] functions can be used.

To extract all files from a tar file, use the extract/1 [page 118] function. To extract only some files or to be able to specify some more options, use the extract/2 [page 119] function.

To return a list of the files in a tar file, use either the table/1 [page 120] or table/2 [page 120] function. To print a list of files to the Erlang shell, use either the t/1 [page 120] or tt/1 [page 121] function.

To convert an error term returned from one of the functions above to a readable message, use the format_error/1 [page 119] 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

\[
\text{add(TarDescriptor, Filename, Options)} \rightarrow \text{RetVal}
\]

Types:
- TarDescriptor = term()
- Filename = filename()
- Options = [Option]
  - Option = dereference \| verbose
  - RetValue = ok \| [error,\{Filename,Reason\}]
- Reason = term()

The \text{add/3} function adds a file to a tar file that has been opened for writing by \text{open/1} [page 120].

- 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.

\[
\text{add(TarDescriptor, Filename, NameInArchive, Options)} \rightarrow \text{RetVal}
\]

Types:
- TarDescriptor = term()
- Filename = filename()
- NameInArchive = filename()
- Options = [Option]
  - Option = dereference \| verbose
  - RetValue = ok \| [error,\{Filename,Reason\}]
- Reason = term()

The \text{add/4} function adds a file to a tar file that has been opened for writing by \text{open/1} [page 120]. It accepts the same options as \text{add/3} [page 117]. \text{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.

\[
\text{close(TarDescriptor)}
\]

Types:
- TarDescriptor = term()

The \text{close/1} function closes a tar file opened by \text{open/1} [page 120].

\[
\text{create(Name, FileList)} \rightarrow \text{RetVal}
\]

Types:
- Name = filename()
- FileList = [filename()]
- RetValue = ok \| [error,\{Name,Reason\}] <\text{\{V\}> Reason = term()
The `create/2` function creates a tar file and archives the files whose names are given in `FileList` into it.

\[\text{create(Name, FileList, OptionList)}\]

**Types:**
- `Name = filename()`
- `FileList = [filename()]`
- `OptionList = [Option]`
- `Option = compressed | cooked | dereference | verbose`
- `ReturnValue = ok | {error, {Name, Reason}} | {Reason = term()}`

The `create/3` function creates a tar file and archives the files whose names are given in `FileList` into it.

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.

\[\text{extract(Name)} \rightarrow \text{ReturnValue}\]

**Types:**
- `Name = filename()`
- `ReturnValue = 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.

\[\text{extract(Name, OptionList)}\]

**Types:**
- `Name = filename() | {binary, Binary} | {file, Fd}`
- `Binary = binary()`
- `Fd = file:descriptor()`
- `OptionList = [Option]`
- `Option = {cwd, Cwd} | {files, FileList} | keep_old_files | verbose`
- `Cwd = [dirname()]`
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, FileList}` 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.
- `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.

The `format_error/1` converts an error reason term to a human-readable error message string.

```
format_error(Reason) -> string()
```

Types:

- Reason = term()

```
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 117] functions to add one file at the time into an opened tar file. When you are finished adding files, use the `close` [page 117] 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".)
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 one 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 even if another process (or the same process) simultaneously deletes or inserts objects in the table.
- '$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.

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 136]. 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()
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.

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.

```erlang
from_dets(Tab, DetsTab) -> Tab
Types:
  • Tab = tid() | atom()
  • DetsTab = atom()
Fills an already created ETS table with the objects in the already opened Dets table named DetsTab. The ETS table is emptied before the objects are inserted.
```

```erlang
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 122]. 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:

```erlang
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:

```erlang
2> X=3.
3> ets:fun2ms(fun([{M,N}) when N > X -> M end).
[[{"$1'"'},{"$2'}"]],[{">'",{"$2'},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:
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 231].

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) -> tuple() | undefined
   Types:
   • Tab = tid() | atom()
   Returns information about the table Tab as a tuple with {Item, Value} elements as specified below.

Warning:
In Erlang/OTP R11B, this function will be corrected to return a list of tuples instead.

• 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.

\[\text{info}(\text{Tab}, \text{Item}) \rightarrow \text{Value} \mid \text{undefined}\]

Types:
- Tab = tid() \mid atom()
- Item, Value - see below

Returns the information associated with Item for the table Tab. 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 \text{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.

\[\text{init_table}(\text{Name}, \text{InitFun}) \rightarrow \text{true}\]

Types:
- Name = atom()
- InitFun = fun(\text{Arg}) \rightarrow \text{Res}
- Arg = read \mid close
- Res = end_of_input \mid \{\text{object()}, \text{InitFun}\} \mid \text{term()}
Replaces the existing objects of the table \texttt{Tab} with objects created by calling the input function \texttt{InitFun}, see below. This function is provided for compatibility with the \texttt{dets} module, it is not more efficient than filling a table by using \texttt{ets:insert/2}.

When called with the argument \texttt{read} the function \texttt{InitFun} is assumed to return \texttt{end_of_input} when there is no more input, or \{\texttt{Objects}, \texttt{Fun}\}, where \texttt{Objects} is a list of objects and \texttt{Fun} is a new input function. Any other value \texttt{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 \texttt{close}, the reply of which is ignored.

If the type of the table is \texttt{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 \texttt{duplicate_bag}.

\begin{verbatim}
insert(Tab, ObjectOrObjects) -> true
Types:
  • Tab = tid() | atom()
  • ObjectOrObjects = tuple() | [tuple()]
Inserts the object or all of the objects in the list \texttt{ObjectOrObjects} into the table \texttt{Tab}. If there already exists an object with the same key as one of the objects, and the table is a \texttt{set} or \texttt{ordered_set}, the old object will be replaced. If the list contains more than one object with the same key and the table is a \texttt{set/ordered_set}, one will be inserted, which one is not defined.

insert_new(Tab, ObjectOrObjects) -> bool()
Types:
  • Tab = tid() | atom()
  • ObjectOrObjects = tuple() | [tuple()]
This function works exactly like \texttt{insert/2}, with the exception that instead of overwriting objects with the same key (in the case of \texttt{set} or \texttt{ordered_set}) or adding more objects with keys already existing in the table (in the case of \texttt{bag} and \texttt{duplicate_bag}), it simply returns \texttt{false}. If \texttt{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 122]. The compiled match_spec is a 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 (\textbackslash \	extbackslash /\textbackslash /). As an example, the following expression:
\begin{verbatim}
ets:is_compiled_ms(ets:match_spec_compile([{'_',[],[true]}]))
\end{verbatim}
will yield \texttt{true}, while the following expressions:
STDLIB Reference Manual

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.
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.

\[
\text{match(Tab, Pattern) -> [Match]}
\]

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.

\[
\text{match(Tab, Pattern, Limit) -> } \{[Match],Continuation\} | \'$end\_of\_table\'$
\]

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:
**ets Reference Manual**

- `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.

```erlang
match_spec_compile(MatchSpec) -> CompiledMatchSpec
```

**Types:**
- `MatchSpec = match_spec()`  
- `CompiledMatchSpec = comp_match_spec()`

This function transforms a `match_spec` [page 122] 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.

```erlang
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 122] 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...
exts:match_spec_run(ets:tab2list(Table),
exts:match_spec_compile(MatchSpec)),
% ...will give the same result as the more common (and more efficient)
exts: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 an 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.
- 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\} Specifies 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 sequence of calls will fail:

\[
T=\text{ets:new(x,[])}, \\
... \\
\{_,C\} = \text{ets:select}(T,\text{ets:fun2ms}((\{N,\_\}=A) \text{ when } (N \text{ rem } 10) =:= 0 \rightarrow A \text{ end}),10), \\
\text{Broken} = \text{binary_to_term}(\text{term_to_binary}(C)), \\
\text{ets:select}(\text{Broken}).
\]

...while the following sequence will work:

\[
T=\text{ets:new(x,[])}, \\
... \\
MS = \text{ets:fun2ms}((\{N,\_\}=A) \text{ when } (N \text{ rem } 10) =:= 0 \rightarrow A \text{ end}), \\
\{_,C\} = \text{ets:select}(T,MS,10), \\
\text{Broken} = \text{binary_to_term}(\text{term_to_binary}(C)), \\
\text{ets:select}(\text{ets:repair_continuation}(\text{Broken},MS)).
\]

...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 even if objects are removed during the traversal. An example:

```erl
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.

```erl
select(Tab, MatchSpec) ->
    Types:
    - Tab = tid() | atom()
    - Object = tuple()
    - MatchSpec = match_spec()

Matches the objects in the table `Tab` using a `match_spec` [page 122]. 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", ...}`
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:

\[
\text{ets:match}(\text{Tab},\{\text{'$1'}, \text{'$2'}, \text{'$3'}\})
\]

is exactly equivalent to:

\[
\text{ets:select}(\text{Tab},\{\{\text{'$1'}, \text{'$2'}, \text{'$3'}\}, [], ['$$'])\})
\]

- and the following ets:match_object/2 call:

\[
\text{ets:match_object}(\text{Tab},\{\text{'$1'}, \text{'$2'}, \text{'$1'}\})
\]

is exactly equivalent to

\[
\text{ets:select}(\text{Tab},\{\{\text{'$1'}, \text{'$2'}, \text{'$1'}\}, [], ['$_'])\})
\]

Composite terms can be constructed in the Result part either by simply writing a list, so that this code:

\[
\text{ets:select}(\text{Tab},\{\{\text{'$1'}, \text{'$2'}, \text{'$3'}\}, [], ['$$'])\})
\]

gives the same output as:

\[
\text{ets:select}(\text{Tab},\{\{\text{'$1'}, \text{'$2'}, \text{'$3'}\}, [], [\{\text{'$_', '$1'}, \text{'$2'}, \text{'$3'}\}])\})
\]

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:

\[
\text{ets:select}(\text{Tab},\{\{\text{'$1'}, \text{'$2'}, \text{'$1'}\}, [], ['$_'])\})
\]

gives the same output as:

\[
\text{ets:select}(\text{Tab},\{\{\text{'$1'}, \text{'$2'}, \text{'$1'}\}, [], [\{\text{'$_', '$1'}, \text{'$2'}, \text{'$3'}\}])\})
\]

- 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:

\[
is_integer(X), \text{is_integer}(Y), X + Y < 4711
\]

is expressed like this (X replaced with ‘$1’ and Y with ‘$2’):

\[
\{\{\text{is_integer, '$1'}\}, \{\text{is_integer, '$2'}\}, \{'<', \{'+', '$1', '$2'}\}, 4711\}]
\]
select(Tab, MatchSpec, Limit) -> [[Match], Continuation] | '$end of table'

Types:
- Tab = tid() | atom()
- Object = tuple()
- 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 122]. 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.

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 122]. 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. The implementation of this function is not efficient.
```

```
tab2list(Tab) -> [Object]
Types:
  • Tab = tid() | atom()
  • Object = tuple()
Returns a list of all objects in the table Tab.
```

```
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
```

STDLIB Reference Manual
TraverseMethod = \( \text{first} \mid \text{next} \mid \text{prev} \mid \text{select} \mid \{ \text{select}, \text{MatchSpec} \} \)

\( \text{MatchSpec} = \text{match\_spec}() \)

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 \( \text{ets:table}/1,2 \) is the means to make the ETS table \( \text{Tab} \) usable to QLC.

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

- \( \text{first\_next} \). The table is traversed one key at a time by calling \( \text{ets:first}/1 \) and \( \text{ets:next}/2 \).
- \( \text{last\_prev} \). The table is traversed one key at a time by calling \( \text{ets:last}/1 \) and \( \text{ets:prev}/2 \).
- \( \text{select} \). The table is traversed by calling \( \text{ets:select}/3 \) and \( \text{ets:select}/1 \). The option \( \text{n\_objects} \) 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 122] (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.
- \( \{ \text{select}, \text{MatchSpec} \} \). As for select the table is traversed by calling \( \text{ets:select}/3 \) and \( \text{ets:select}/1 \). 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> \( \text{ets:insert(Tab = ets:new(t, []), [[1,a],[2,b],[3,c],[4,d]]}, \)
\( \text{MS = ets:fun2ms(fun([X,Y]) when (X > 1) or (X < 5) -> Y end)}, \)
\( \text{QH1 = ets:table(Tab, [[\text{traverse}, \{ \text{select}, \text{MS} \}]]).} \)

An example with implicit match\_spec:

10> \( \text{QH2 = qlc:q(\\{Y | | X \leq 5 \}, X > 1 \} \).} \)

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

11> \( \text{qlc:info(QH1) == qlc:info(QH2).} \)
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()
This function is a utility to test a match_spec [page 122] 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, \{Pos,Incr,Threshold,SetValue\}) -> Result
update_counter(Tab, Key, \{Pos,Incr\}) -> Result
update_counter(Tab, Key, Incr) -> Result

Types:
- Tab = tid() | atom()
- Key = term()
- Pos = Incr = Threshold = SetValue = Result = int()

This function provides an efficient way to update a counter, 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 \(\langle \text{keypos}+1\rangle\) 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

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, or,
- any of Pos, Incr, Threshold or SetValue is not an integer
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
  
  \[
  \langle\langle\text{Size:HeaderLength/unit\_8}\rangle\rangle = B.
  \]

- \{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.
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_266005.17, where 17 is a sequence number. Existing files will be overwritten. Temporary files are deleted unless some uncaught EXIT signal occurs.

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.

By default approximately 512*1024 bytes read from files are sorted internally. This option should rarely be needed.

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 = \(\text{int()} > 0\)
  - Format = binary_term | term | binary | FormatFun
  - FormatFun = fun(\(\text{binary}()) -> \text{Term}\)
  - Order = ascending | descending | OrderFun
  - OrderFun = fun(\(\text{Term, Term}\) -> bool())
  - TempDirectory = "" | filename()
  - Size = \(\text{int()} > 0\)
  - NoFiles = \(\text{int()} > 1\)

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.

\begin{verbatim}
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.
\end{verbatim}

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.
- \{not_a_directory, FileName\}. The file supplied with the tmpdir option is not a
directory.
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.

Exports

ensure_dir(Name) -> true

Types:
- Name = filename() | dirname()

The ensure_dir/1 function checks that all parent directories for the given file or directory name exist, creating them if not.

filesize(Filename) -> integer()

The filesize 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.

\[
\text{is\_regular(Name)} \rightarrow \text{true} \mid \text{false}
\]

Types:
- `Name = filename()`

The `is\_regular/1` function returns `true` if `Name` refers to a file (regular file), and `false` otherwise.

\[
\text{last\_modified(Name)} \rightarrow \{\{\text{Year},\text{Month},\text{Day}\},\{\text{Hour},\text{Min},\text{Sec}\}\}
\]

Types:
- `Name = filename() \mid \text{dirname()}

The `last\_modified/1` function returns the date and time the given file or directory was last modified.

\[
\text{wildcard(Wildcard)} \rightarrow \text{list()}
\]

Types:
- `Wildcard = filename() \mid \text{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:

\[
\text{filelib:Wildcard("lib/*/ebin/\*.beam")}
\]

To find either `.erl` or `.hrl` in all applications `src` directories, the following

\[
\text{filelib:Wildcard("lib/*/src/\*.\?rl")}
\]

or the following line

\[
\text{filelib:Wildcard("lib/*/src/\*.\{erl,hrl\}")}
\]
can be used.
To find all `.hrl` files in either `src` or `include` directories, use:

```erlang
filelib:wildcard("lib/*/\{src,include}/\*.hrl").
```

To find all `.erl` or `.hrl` files in either `src` or `include` directories, use:

```erlang
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
The path name is relative to the current working directory on the current volume.
Example: foo/bar, ../src

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

```
filename:rootname("/beam.src/kalle").
"/beam.src/kalle"

filename:rootname("/beam.src/foo.erl").
"/beam.src/foo"

filename:rootname("/beam.src/foo.erl", ".erl").
"/beam.src/foo"

filename:rootname("/beam.src/foo.beam", ".erl").
"/beam.src/foo.beam"
```

```
filename:split("/usr/local/bin").
["/", "usr", "local", "bin"]

filename:split("foo/bar").
["foo", "bar"]

filename:split("a:\msdev\include").
["a:/", ".msdev", "include"]
```

```
find_src(Beam) -> {SourceFile, Options}
find_src(Beam, Rules) -> {SourceFile, Options}
```

```
Types:
• Beam = Module | Filename
• Module = atom()
• Filename = string() | atom()
• SourceFile = string()
• Options = [Opt]
• Opt = [i, string()] | [outdir, string()] | [d, atom()]
```
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}. 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

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 $O(|T| \times \log(|S|))$, 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
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. Assumes that Element is
present in Set1.

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()
**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()
gb_sets

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.

large_set(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) -> integer()

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) \rightarrow List
Types:
• Set = gb_set()
• List = [term()]
  Returns the elements of Set as a list.

union(Set1, Set2) \rightarrow Set3
Types:
• Set1 = Set2 = Set3 = gb_set()
  Returns the merged (union) gb_set of Set1 and Set2.

union(SetList) \rightarrow 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 162], ordsets(3) [page 243], sets(3) [page 283]
gb_trees

Erlang Module

An efficient implementation of Prof. Arne Andersson’s General Balanced Trees. These have no storage overhead compared to unbalanced 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 O(K).

Original balance condition \( h(T) \leq \text{ceil}(c \times \log(|T|)) \) has been changed to the similar (but not quite equivalent) condition \( 2^{-h(T)} \leq |T| \leq c \). This should also be O(K).

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 logarithmic (as it should be).

DATA TYPES

\( \text{gb_tree}() = \text{a GB tree} \)

Exports

\( \text{balance(Tree1)} -> \text{Tree2} \)

Types:
- Tree1 = Tree2 = \text{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.

\( \text{delete(Key, Tree1)} -> \text{Tree2} \)

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.

\textbf{insert(Key, Val, Tree1) \rightarrow Tree2}

\begin{itemize}
  \item Key = Val = term()
  \item Tree1 = Tree2 = gb_tree()
\end{itemize}

Inserts Key with value Val into Tree1; returns the new tree. Assumes that the key is not present in the tree, crashes otherwise.

\textbf{is_defined(Key, Tree) \rightarrow bool()}

\begin{itemize}
  \item Tree = gb_tree()
\end{itemize}

Returns true if Key is present in Tree, otherwise false.

\textbf{is_empty(Tree) \rightarrow bool()}

\begin{itemize}
  \item Tree = gb_tree()
\end{itemize}

Returns true if Tree is an empty tree, and false otherwise.

\textbf{iterator(Tree) \rightarrow Iter}

\begin{itemize}
  \item Tree = gb_tree()
  \item Iter = term()
\end{itemize}

Returns an iterator that can be used for traversing the entries of Tree; see \texttt{next/1}. The implementation of this is very efficient; traversing the whole tree using \texttt{next/1} is only slightly slower than getting the list of all elements using \texttt{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.

\textbf{keys(Tree) \rightarrow [Key]}

\begin{itemize}
  \item Tree = gb_tree()
  \item Key = term()
\end{itemize}

Returns the keys in Tree as an ordered list.

\textbf{largest(Tree) \rightarrow \{Key, Val\}}

\begin{itemize}
  \item Tree = gb_tree()
  \item Key = Val = term()
\end{itemize}

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.

\textbf{next(Iter1) \rightarrow \{Key, Val, Iter2\}}
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.

```plaintext
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 156], `dict(3)` [page 82]
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:

<table>
<thead>
<tr>
<th>gen_event module</th>
<th>Callback module</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen_event:start_link</td>
<td>Module:init/1</td>
</tr>
<tr>
<td>gen_event:add_handler</td>
<td></td>
</tr>
<tr>
<td>gen_event:swap_handler</td>
<td></td>
</tr>
<tr>
<td>gen_event:notify</td>
<td>Module:handle_event/2</td>
</tr>
<tr>
<td>gen_event:sync_notify</td>
<td>Module:handle_call/2</td>
</tr>
<tr>
<td>gen_event:call</td>
<td>Module:handle_info/2</td>
</tr>
<tr>
<td>gen_event:delete_handler</td>
<td>Module:terminate/2</td>
</tr>
<tr>
<td>gen_event:swap_handler</td>
<td>Module1:terminate/2</td>
</tr>
<tr>
<td>gen_event:swap_sup_handler</td>
<td>Module2:init/1</td>
</tr>
<tr>
<td>gen_event:which_handlers</td>
<td></td>
</tr>
<tr>
<td>gen_event:stop</td>
<td>Module:terminate/2</td>
</tr>
<tr>
<td>gen_event:stop</td>
<td>Module:code_change/3</td>
</tr>
</tbody>
</table>

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.

Unless otherwise stated, all functions in this module fail if the specified event manager does not exist or if bad arguments are given.

Exports

```prolog
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/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.

```prolog
start() -> Result
start(EventMgrName) -> Result
```

Types:
- `EventMgrName = {local, Name} | {global, Name} | pid()`
- `Name = Node = 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.

```prolog
add_handler(EventMgrRef, Handler, Args) -> Result
```

Types:
- `EventMgr = Name | {Name, Node} | {global, Name} | pid()`
- `Name = Node = atom()`
- `Handler = Module | {Module, Id}`
---

**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.

---

**add_sup_handler(EventMgrRef, Handler, Args) -> Result**
- \{swapped,NewHandler,Pid\}, if the process Pid has replaced the event
  handler with another event handler NewHandler using a call to
  \texttt{swap\_handler/3} or \texttt{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.

\begin{verbatim}
notify(EventMgrRef, Event) \rightarrow ok
sync\_notify(EventMgrRef, Event) \rightarrow ok
\end{verbatim}

Types:
- \texttt{EventMgrRef = Name | \{Name,Node\} | \{global,Name\} | pid()}
- \texttt{Name = Node = atom()}
- \texttt{Event = term()}

Sends an event notification to the event manager \texttt{EventMgrRef}. The event manager will
\texttt{call Module:handle\_event/2} for each installed event handler to handle the event.
\texttt{notify} is asynchronous and will return immediately after the event notification has
been sent. \texttt{sync\_notify} is synchronous in the sense that it will return \texttt{ok} after the event
has been handled by all event handlers.

See add\_handler/3 for a description of \texttt{EventMgrRef}.

\texttt{Event} is an arbitrary term which is passed as one of the arguments to
\texttt{Module:handle\_event/2}.

\texttt{notify} will not fail even if the specified event manager does not exist, unless it is
specified as \texttt{Name}.

\begin{verbatim}
call(EventMgrRef, Handler, Request) \rightarrow Result
call(EventMgrRef, Handler, Request, Timeout) \rightarrow Result
\end{verbatim}

Types:
- \texttt{EventMgrRef = Name | \{Name,Node\} | \{global,Name\} | pid()}
- \texttt{Name = Node = atom()}
- \texttt{Handler = Module | \{Module,Id\}}
- \texttt{Module = atom()}
- \texttt{Id = term()}
- \texttt{Request = term()}
- \texttt{Timeout = int() > 0 | infinity}
- \texttt{Result = Reply | \{error,Error\}}
- \texttt{Reply = term()}
- \texttt{Error = bad\_module | \{'EXIT',Reason\} | term()}
- \texttt{Reason = term()}

170 STDLIB
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.

```
define 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]}.```

```
define 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()
```

STDLIB Reference Manual  

STDLIB Reference Manual  

gen_event

STDLIB Reference Manual  

gen_event

STDLIB Reference Manual  

gen_event
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(Arg1, ...)`, 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([Arg2, 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 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 function returns `{error, ['EXIT', Reason]} or {error, Term}`, respectively.

```erlang
swap_sup_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 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.

```erlang
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`.

---

```
172
STDLIB
```
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}

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} where State is the initial internal state of the event handler.

Module:handle_event(Event, State) -> Result

Types:
- Event = term()
- State = term()
- Result = {ok,NewState} | {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}` the event handler will remain in the event manager with the possible updated internal state `NewState`.

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}
    | {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}
    | {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 328], sys(3) [page 338]
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:

```
<table>
<thead>
<tr>
<th>gen_fsm module</th>
<th>Callback module</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen_fsm:start_link</td>
<td>-----&gt; Module:init/1</td>
</tr>
<tr>
<td>gen_fsm:send_event</td>
<td>-----&gt; Module:StateName/2</td>
</tr>
<tr>
<td>gen_fsm:send_all_state_event</td>
<td>-----&gt; Module:handle_event/3</td>
</tr>
<tr>
<td>gen_fsm:sync_send_event</td>
<td>-----&gt; Module:StateName/3</td>
</tr>
<tr>
<td>gen_fsm:sync_send_all_state_event</td>
<td>-----&gt; Module:handle_sync_event/4</td>
</tr>
<tr>
<td></td>
<td>-----&gt; Module:handle_info/3</td>
</tr>
<tr>
<td></td>
<td>-----&gt; Module:terminate/3</td>
</tr>
<tr>
<td></td>
<td>-----&gt; Module:code_change/4</td>
</tr>
</tbody>
</table>
```

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.
Exports

\[
\text{start\_link(Module, Args, Options) -> Result}
\]
\[
\text{start\_link(FsmName, Module, Args, Options) -> Result}
\]

Types:
- \( \text{FsmName = \{local, Name\} | \{global, GlobalName\} } \)
- \( \text{Name = atom() } \)
- \( \text{GlobalName = term() } \)
- \( \text{Module = atom() } \)
- \( \text{Args = term() } \)
- \( \text{Options = \[Option\] } \)
  - \( \text{Option = \{debug, Dbgs\} | \{timeout, Time\} | \{spawn\_opt, SOpts\} } \)
- \( \text{Dbgs = \[Dbg\] } \)
  - \( \text{Dbg = trace | log | statistics} \)
- \( \text{SOpts = \[SOpt\] } \)
  - \( \text{SOpt - see erlang:spawn\_opt/2,3,4,5} \)

\( \text{Result = \{ok, Pid\} | ignore | \{error, Error\} } \)
- \( \text{Pid = pid()} \)
- \( \text{Error = \{already\_started, Pid\} | term()} \)

Creates a \text{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 \text{gen\_fsm} is linked to the supervisor.

The \text{gen\_fsm} process calls \text{Module: init/1} to initialize. To ensure a synchronized startup procedure, \text{start\_link/3,4} does not return until \text{Module: init/1} has returned.

If \( \text{FsmName=\{local, Name\} } \), the \text{gen\_fsm} is registered locally as \( \text{Name} \) using \text{register/2}. If \( \text{FsmName=\{global, GlobalName\} } \), the \text{gen\_fsm} is registered globally as \( \text{GlobalName} \) using \text{global:register\_name/2}. If no name is provided, the \text{gen\_fsm} is not registered.

\( \text{Module} \) is the name of the callback module.

\( \text{Args} \) is an arbitrary term which is passed as the argument to \text{Module: init/1}.

If the option \( \{\text{timeout, Time} \} \) is present, the \text{gen\_fsm} is allowed to spend \( \text{Time} \) milliseconds initializing or it will be terminated and the start function will return \( \{\text{error, timeout}\} \).

If the option \( \{\text{debug, Dbgs} \} \) is present, the corresponding \text{sys} function will be called for each item in \( \text{Dbgs} \). Refer to \text{sys(3)} for more information.

If the option \( \{\text{spawn\_opt, SOpts} \} \) is present, \( \text{SOpts} \) will be passed as option list to the \text{spawn\_opt} BIF which is used to spawn the \text{gen\_fsm} process. Refer to \text{erlang(3)} for information about the \text{spawn\_opt} options.

If the \text{gen\_fsm} is successfully created and initialized the function returns \( \{\text{ok, Pid}\} \), where \( \text{Pid} \) is the pid of the \text{gen\_fsm}. If there already exists a process with the specified \( \text{FsmName} \), the function returns \( \{\text{error, \{already\_started, Pid\}} \} \) where \( \text{Pid} \) is the pid of that process.

If Module: init/1 fails with Reason, the function returns \( \{\text{error, Reason}\} \). If Module: init/1 returns \( \{\text{stop, Reason}\} \) or \text{ignore}, the process is terminated and the function returns \( \{\text{error, Reason}\} \) or \text{ignore}, respectively.
start(Module, Args, Options) \rightarrow Result
start(FsmName, Module, Args, Options) \rightarrow 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
- 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 for a description of arguments and return values.

send_event(FsmRef, Event) \rightarrow 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) \rightarrow 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 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 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.

In the case where the gen_fsm terminates during the handling of the event and the caller is linked to the gen_fsm and trapping exits, the exit message is removed from the caller's receive queue before the function call fails.

This behaviour is retained for backwards compatibility only and may change in the future. Note that if the gen_fsm crashes in between calls, a linked process must take care of the exit message anyway.

Warning: Under certain circumstances (e.g. FsmRef = {Name,Node}, and Node goes down) the exit message cannot be removed.

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 \texttt{gen\_fsm FsmRef} and waits until a reply arrives or a timeout occurs. The \texttt{gen\_fsm} will call \texttt{Module:handle\_sync\_event/4} to handle the event.

See \texttt{send\_event/2} for a description of \texttt{FsmRef} and \texttt{Event}. See \texttt{sync\_send\_event/3} for a description of \texttt{Timeout} and \texttt{Reply}.

See \texttt{send\_all\_state\_event/2} for a discussion about the difference between \texttt{sync\_send\_event} and \texttt{sync\_send\_all\_state\_event}.

\begin{verbatim}
reply(Caller, Reply) -> true

Types:
  • Caller - see below
  • Reply = term()

This function can be used by a \texttt{gen\_fsm} to explicitly send a reply to a client process that called \texttt{sync\_send\_event} or \texttt{sync\_send\_all\_state\_event}, when the reply cannot be defined in the return value of \texttt{Module:State/3} or \texttt{Module:handle\_sync\_event/4}.

\texttt{Caller} must be the \texttt{From} argument provided to the callback function. \texttt{Reply} is an arbitrary term, which will be given back to the client as the return value of \texttt{sync\_send\_event} or \texttt{sync\_send\_all\_state\_event}.
\end{verbatim}

\begin{verbatim}
send\_event\_after(Time, Event) -> Ref

Types:
  • Time = integer()
  • Event = term()
  • Ref = reference()

Sends a delayed event internally in the \texttt{gen\_fsm} that calls this function after \texttt{Time} ms. Returns immediately a reference that can be used to cancel the delayed send using \texttt{cancel\_timer/1}.

The \texttt{gen\_fsm} will call \texttt{Module:StateName/2} to handle the event, where \texttt{StateName} is the name of the current state of the \texttt{gen\_fsm} at the time the delayed event is delivered.

\texttt{Event} is an arbitrary term which is passed as one of the arguments to \texttt{Module:StateName/2}.
\end{verbatim}

\begin{verbatim}
start\_timer(Time, Msg) -> Ref

Types:
  • Time = integer()
  • Msg = term()
  • Ref = reference()

Sends a timeout event internally in the \texttt{gen\_fsm} that calls this function after \texttt{Time} ms. Returns immediately a reference that can be used to cancel the timer using \texttt{cancel\_timer/1}.

The \texttt{gen\_fsm} will call \texttt{Module:StateName/2} to handle the event, where \texttt{StateName} is the name of the current state of the \texttt{gen\_fsm} at the time the timeout message is delivered.

\texttt{Msg} is an arbitrary term which is passed in the timeout message, \{timeout, Ref, Msg\}, as one of the arguments to \texttt{Module:StateName/2}.
\end{verbatim}

\begin{verbatim}
cancel\_timer(Ref) -> RemainingTime | false
\end{verbatim}
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 or start_timer/2.

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 248]. 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 178]. 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 ??]. 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(Arg0) -> Result
  
Types:
  • Args = term()
  • Return = {ok,StateName,StateData} | {ok,StateName,StateData,Timeout}
  • | {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 or gen_fsm:start_link/3,4,
this function is called by the new process to initialize.
Arg0 is the Args argument provided to the start function.
If initialization is successful, the function should return {ok,StateName,StateData} or
{ok,StateName,StateData,Timeout}, 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 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,NewStateName,NewStateData} |
    {next_state,NewStateName,NewStateData,Timeout}
  • | {stop,Reason,NewStateData}
  • NewStateName = 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, 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.

StateData is the state data of the gen_fsm.

If the function returns \{next_state,NextStateName,NewStateData\} or \{next_state,NextStateName,NewStateData,Timeout\}, 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.

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\} | \{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, 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\} | \{next_state,NextStateName,NewStateData\} | \{next_state,NextStateName,NewStateData,Timeout\} | \{stop,Reason,Reply,NewStateData\} | \{stop,Reason,NewStateData\}
- Reply = term()
- NextStateName = atom()
- NewStateData = term()
- Timeout = int() > 0 | infinity
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, 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 and Tag is a unique tag.

StateData is the state data of the gen_fsm.

If the function returns {reply,Reply,NextStateName,NewStateData} or {reply,Reply,NextStateName,NewStateData,Timeout}, Reply will be given back to From as the return value of sync_send_event. 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.

If the function returns {next_state,NextStateName,NewStateData} or {next_state,NextStateName,NewStateData,Timeout}, the gen_fsm will continue executing in NextStateName with NewStateData. Any reply to From must be given explicitly using gen_fsm:reply/2.

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} |
  {next_state,NextStateName,NewStateData} |
  {next_state,NextStateName,NewStateData,Timeout} |
  {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, 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}  
- | {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 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 \texttt{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

supervisor(3) [page 328], sys(3) [page 338]
**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:

\[
\begin{array}{c|c}
\text{gen_server module} & \text{Callback module} \\
\hline
\text{gen_server:start_link} & \text{Module:init/1} \\
\text{gen_server:call} & \\
\text{gen_server:multi_call} & \text{Module:handle_call/3} \\
\text{gen_server:cast} & \text{Module:handle_cast/2} \\
\text{gen_server:abcast} & \text{Module:handle_info/2} \\
\text{-} & \text{Module:terminate/2} \\
\text{-} & \text{Module:code_change/3}
\end{array}
\]

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.

**Exports**

\[
\begin{align*}
\text{start_link} & (\text{Module, Args, Options}) \rightarrow \text{Result} \\
\text{start_link} & (\text{ServerName, Module, Args, Options}) \rightarrow \text{Result}
\end{align*}
\]

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** is **local,Name** the gen_server is registered locally as Name using **register/2**. If **ServerName** is **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. Refer to **sys(3)** for more information.

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. Refer to **erlang(3)** for information about the spawn_opt options.

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]
gen_server STDLIB Reference Manual

- Option = \{debug,Dbgs\} \{timeout,Time\} \{spawn_opt,SOpts\}
- Dbgs = \{Dbg\}
- Dbg = trace \{log, log, log\} \{statistics\} \{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` for a description of arguments and return values.

call(ServerRef, Request) \rightarrow Reply
call(ServerRef, Request, Timeout) \rightarrow 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.

There is a special case for backwards compatibility. If

- the client is linked to the gen_server, and
- the client is trapping exits, and
- the gen_server terminates while handling the request
then the exit message is removed from the client's receive queue before the function call fails. This special-case behaviour may be removed in the future because it is inconsistent with the behaviour when a gen_server dies between calls and also because the exit message cannot be removed in some circumstances, for instance when ServerRef = {Name, Node} and Node goes down.

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 running Erlang/OTP R6B or older, 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 running Erlang/OTP R7B or later.

This function does not read out any exit messages like call/2,3 does.
The previously undocumented functions `safe_multi_call/2,3,4` were removed in OTP R7B/Erlang 5.0 since this function is now safe, except in the case mentioned above.

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` 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_server` locally registered as Name at the specified nodes. The function returns immediately and ignores nodes that does not exist, or where the `gen_server` Name does not exist. The `gen_server` will call `Module:handle_cast/2` to handle the request.

See `multi_call/2,3,4` for a description of the arguments.

reply(Client, Reply) -> true

Types:
- Client - see below
- Reply = term()

This function can be used by a `gen_server` to explicitly send a reply to a client that called `call` or `multi_call`, 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` or `multi_call`.

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, D bgs}
- D bgs = [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 248]. 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 188]. 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 ??]. 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(Arg) -> Result

Types:
- Arg = term()
- Result = {ok, State} | {ok, State, Timeout}
- | {stop, Reason} | ignore
- State = term()
- Timeout = int() >= 0 | infinity
Whenever a gen_server is started using \texttt{gen_server:start/3,4} or \texttt{gen_server:start_link/3,4}, this function is called by the new process to initialize. \texttt{Args} is the \texttt{Args} argument provided to the start function.

If the initialization is successful, the function should return \{\texttt{ok}, \texttt{State}\} or \{\texttt{ok}, \texttt{State}, \texttt{Timeout}\}, where \texttt{State} is the internal state of the \texttt{gen_server}.

If an integer \texttt{timeout} value is provided, a \texttt{timeout} will occur unless a request or a message is received within \texttt{Timeout} milliseconds. A \texttt{timeout} is represented by the atom \texttt{timeout} which should be handled by the \texttt{handle_info/2} callback function. The atom \texttt{infinity} can be used to wait indefinitely, this is the default value.

If something goes wrong during the initialization the function should return \{\texttt{stop}, \texttt{Reason}\} where \texttt{Reason} is any term, or \texttt{ignore}.

\textbf{Module:handle_call(Request, From, State) --> Result}

\begin{itemize}
\item \texttt{Request} = \texttt{term()}
\item \texttt{From} = \{\texttt{pid()}, \texttt{Tag}\}
\item \texttt{State} = \texttt{term()}
\item \texttt{Result} = \{\texttt{reply,Reply,NewState}\} | \{\texttt{reply,Reply,NewState,Timeout}\}
\item \texttt{Reply} = \texttt{term()}\texttt{NewState} = \texttt{term()}
\item \texttt{Timeout} = \texttt{int()}>0 | \texttt{infinity}
\item \texttt{Reason} = \texttt{term()}
\end{itemize}

Whenever a \texttt{gen_server} receives a request sent using \texttt{gen_server:call/2,3} or \texttt{gen_server:multi_call/2,3,4}, this function is called to handle the request. \texttt{Request} is the \texttt{Request} argument provided to \texttt{call} or \texttt{multi_call}. \texttt{From} is a tuple \{\texttt{Pid,Tag}\} where \texttt{Pid} is the pid of the client and \texttt{Tag} is a unique tag. \texttt{State} is the internal state of the \texttt{gen_server}.

If the function returns \{\texttt{reply,Reply,NewState}\} or \{\texttt{reply,Reply,NewState,Timeout}\}, \texttt{Reply} will be given back to \texttt{From} as the return value of \texttt{call} or included in the return value of \texttt{multi_call}. The \texttt{gen_server} then continues executing with the possibly updated internal state \texttt{NewState}. See \texttt{Module:init/1} for a description of \texttt{Timeout}.

If the functions returns \{\texttt{noreply,NewState}\} or \{\texttt{noreply,NewState,Timeout}\}, the \texttt{gen_server} will continue executing with \texttt{NewState}. Any reply to \texttt{From} must be given explicitly using \texttt{gen_server:reply/2}.

If the function returns \{\texttt{stop,Reason,Reply,NewState}\}, \texttt{Reply} will be given back to \texttt{From}. If the function returns \{\texttt{stop,Reason,NewState}\}, any reply to \texttt{From} must be given explicitly using \texttt{gen_server:reply/2}. The \texttt{gen_server} will then call \texttt{Module:terminate(Reason,NewState)} and terminate.

\textbf{Module:handle_cast(Request, State) --> Result}

\begin{itemize}
\item \texttt{Types:}
\end{itemize}
Whenever a gen_server receives a request sent using gen_server:cast/2 or gen_server:abcast/2,3, 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}
• {stop,Reason,NewState}
• NewState = term()
• Timeout = int() >= 0 | infinity
• Reason = 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 \texttt{error_logger:format/2}.

\textbf{Module:code_change} (OldVsn, State, Extra) \rightarrow \{ok, NewState\}

\textbf{Types}:
\begin{itemize}
  \item \texttt{OldVsn = Vsn | \{down, Vsn\}}
  \item \texttt{Vsn = term()}
  \item \texttt{State = NewState = term()}
  \item \texttt{Extra = term()}
\end{itemize}

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 \{\texttt{update, Module, Change, \ldots}\} where \texttt{Change=\{advanced, Extra\}} is given in the appup file. See \texttt{OTP Design Principles} for more information.

In the case of an upgrade, \texttt{OldVsn} is \texttt{Vsn}, and in the case of a downgrade, \texttt{OldVsn} is \texttt{\{down, Vsn\}}. \texttt{Vsn} is defined by the \texttt{vsn} attribute(s) of the old version of the callback module \texttt{Module}. If no such attribute is defined, the version is the checksum of the BEAM file.

\texttt{State} is the internal state of the gen_server.
\texttt{Extra} is passed as-is from the \texttt{\{advanced, Extra\}} part of the update instruction.

The function should return the updated internal state.

\textbf{SEE ALSO}

\texttt{supervisor(3)} [page 328], \texttt{sys(3)} [page 338]
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

`put_chars([IoDevice,] IoData) -> ok`

Types:
   - `IoDevice = io_device()`
   - `IoData = IoList | binary()`
   - `IoList = [char() | binary() | IoList]`

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.

```erlang
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.

```erlang
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.

**Note:** The binary option does not work against IO servers on remote nodes running an older version of Erlang/OTP than R9C.

```erlang
write([IoDevice,] Term) -> ok
```

Types:

- IoDevice = io_device()
- Term = term()

Writes the term Term to the standard output (IoDevice).

```erlang
read([IoDevice,] Prompt) -> Result
```

Types:

- IoDevice = io_device()
- Prompt = atom() | string()
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} The parsing was successful.
{eof} End of file was encountered.
{error, ErrorInfo} The parsing failed.

fwrite(Format) ->
fwrite([IoDevice,] Format, Data) -> ok
format(Format) ->
format([IoDevice,] Format, Data) -> ok

Types:
- IoDevice = io_device()
- Format = atom() | string()
- 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, it is first converted to a list with the aid of atom_to_list/1.

1> io:fwrite("Hello world!\n", []).
Hello world!
ok
The general format of a control sequence is \texttt{~F.P.PadC}. The character \texttt{C} determines the type of control sequence to be used, \texttt{F} and \texttt{P} are optional numeric arguments. If \texttt{F}, \texttt{P}, or \texttt{Pad} is *, the next argument in Data is used as the numeric value of \texttt{F} or \texttt{P}.

\texttt{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.

\texttt{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.

\texttt{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.
- \texttt{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:

  \begin{verbatim}
  2> io:fwrite("|~10.5c|~10.5c|~5c|~n", [$a, $b, $c]).
  | aaaaa|aaaaa |ccccc|
  ok
  \end{verbatim}

- \texttt{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.

- \texttt{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.

- \texttt{g} The argument is a float which is written as \texttt{f}, if it is \texttt{>= 0.1} and \texttt{< 10000.0}. Otherwise, it is written in the \texttt{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 \texttt{f} format with the desired number of significant digits, it is also written in the \texttt{e} format.

- \texttt{s} Prints the argument with the \texttt{string} syntax. The argument is a list of characters (possibly not a flat list), 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:

  \begin{verbatim}
  3> io:fwrite("|~10w|~n", [[hey, hey, hey]]).
  ***********
  ok
  4> io:fwrite("|~10s|~n", [io:lib:write([hey, hey, hey])]).
  |hey,hey,h|
  ok
  \end{verbatim}
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 in the default \texttt{g} format.

p Writes the data with standard syntax in the same way as \texttt{~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:

```erlang
5> T = [{attributes,[[{id,age,1.50000},{mode,explicit}],
                     {typename,"INTEGER"}],
       [{id,cho},{mode,explicit},{typename,'Cho']},
       {typename,'Person'},
       {tag,['PRIVATE',3]},
       {mode,implicit}].
ok
6> io:fwrite("~w~n", [T]).
{attributes,[[[id,age,1.50000},{mode,explicit}]
            ,{typename,"INTEGER"}],
       [{id,cho},{mode,explicit}]
       ,{typename,'Cho']},
       {typename,'Person'},
       {tag,['PRIVATE',3]},
       {mode,implicit}]
ok
7> io:fwrite("~p~n", [T]).
{attributes,[[[id,age,1.50000}],
            {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 \texttt{io:fwrite} or \texttt{io:format}. For example, using \texttt{T} above:

```erlang
8> io:fwrite("Here T = ~p~n", [T]).
Here T = [{attributes,[[[id,age,1.50000],
                     {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 \texttt{~w}, but takes an extra argument which is the maximum depth to which terms are printed. Anything below this depth is replaced with \ldots\ For example, using \texttt{T} above:

```erlang
9> io:fwrite("~W~n", [T,9]).
```
ok

If the maximum depth has been reached, then it is impossible to read in the resultant output. Also, the \ldots 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 \texttt{~p}, but takes an extra argument which is the maximum depth to which terms are printed. Anything below this depth is replaced with \ldots. For example:

\begin{verbatim}
10> io:fwrite("~P~n", [T,9]).
\end{verbatim}

\begin{verbatim}
ok
\end{verbatim}

**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:

\begin{verbatim}
11> io:format("~.16B~n", [31]).
1F
ok
12> io:format("~.2B~n", [-19]).
-10011
ok
13> io:format("~.36B~n", [5*36+35]).
5Z
ok
\end{verbatim}

**X** Like \texttt{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.

\begin{verbatim}
14> io:format("~X~n", [31,"10#"]).
10#31
ok
15> io:format("~.16X~n", [-31,"0x"]).
-0x1F
ok
\end{verbatim}

**#** Like \texttt{B}, but prints the number with an Erlang style '\#'-separated base prefix.

\begin{verbatim}
\end{verbatim}
16> io:format("-.10#n", [31]).
 10#31
ok
17> io:format("-.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'
ok
19> io:fwrite("s", [65]).
** exited: [badarg, [{io, format, [<0.22.0>, "s", "A"]},
                  {erl_eval, do_apply, 5},
                  {shell, exprs, 6},
                  {shell, eval_loop, 2}]] **

In this example, an attempt was made to output the single character "65" with the aid of
the string formatting directive "s".

The two functions fwrite and format are identical. The old name format has been
retained for backwards compatibility, while the new name fwrite has been added as a
logical complement to fread.

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 direct 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.
- A decimal integer is expected.
- 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.
- A floating point number is expected. It must follow the Erlang floating point number syntax.
- 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.
- Similar to s, but the resulting string is converted into an atom.
- 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.
- 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,[1.90000,3.55000e+4,15.0000]}
21> io:fread('enter>', "~10f~d").
enter>5.67899
{ok,[5.67800,99]}
22> io:fread('enter>', ":"~10s:"~10c:").
enter>1 alan : joe :
{ok,["alan", " joe "]}

scan_erl_exprs(Prompt) ->
scan_erl_exprs([IoDevice,] Prompt, StartLine) -> Result
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 last 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, ExprList, EndLine} | {eof, EndLine} | {error, ErrorInfo, EndLine}
• ExprList – 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, ExprList, 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 = LineLength = 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 = string()
- Data = [term()]

Returns a character list which represents Data formatted in accordance with Format. See io:fwrite/2 [page ??] 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 203] 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,[15.6000,1.73000e-5,24.50000],[[]]}

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.

\hfill
\begin{verbatim}
write_atom(Atom) -\rightarrow chars()
Types:
\bullet Atom = atom()
Returns the list of characters needed to print the atom Atom.
\hfill
write_string(String) -\rightarrow chars()
Types:
\bullet String = string()
Returns the list of characters needed to print String as a string.
\hfill
write_char(Integer) -\rightarrow chars()
Types:
\bullet Integer = int()
Returns the list of characters needed to print a character constant.
\hfill
indentation(String, StartIndent) -\rightarrow int()
Types:
\bullet String = string()
\bullet StartIndent = int()
\end{verbatim}
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.

dee 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.
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 = string()
- Args = [term()]
Prints error message Args in accordance with Format. Similar to io:format/2, see io(3) [page ??].

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, 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.

**Exports**

`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:

```erlang
groups: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:

```erlang
groups:append("abc", "def").
"abcdef"
```

`groups: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.
delete(Elem, List1) -> List2

Types:
- Elem = term()
- List1 = List2 = [term()]

Returns a copy of List1, but the first occurrence of Elem, if present, is deleted.

duplicate(N, Elem) -> List

Types:
- N = int()
- Elem = term()
- List = [Elem]

Returns a list which contains N copies of the term Elem. For example:

> lists:duplicate(5, xx).

[xx, xx, xx, xx, xx]

flatlength(DeepList) -> int()

Types:
- DeepList = [term() | DeepList]

Equivalent to length(flatten(DeepList)), but more efficient.

flatten(DeepList) -> List

Types:
- DeepList = [Elem | DeepList]
- Elem = term()
- List = [Elem]

Returns a flattened version of DeepList.

flatten(DeepList, Tail) -> List

Types:
- DeepList = [Elem | DeepList]
- Elem = term()
- Tail = [term()]
- List = [Elem]

Returns a flattened version of DeepList with the tail Tail appended.

keydelete(Key, N, TupleList1) -> TupleList2

Types:
- Key = term()
keymember(Key, N, TupleList) -> bool()

Types:
- Key = term()
- N = 1..size(Tuple)
- TupleList = [Tuple]
- Tuple = tuple()

Returns true if there is a tuple in TupleList whose Nth element is Key, otherwise false.

keymerge(N, TupleList1, TupleList2) -> TupleList3

Types:
- N = 1..size(Tuple)
- TupleList1 = TupleList2 = TupleList3 = [Tuple]
- Tuple = tuple()

Returns the sorted list formed by merging TupleList1 and TupleList2. The sorting is performed on the Nth element of each tuple. Both TupleList1 and TupleList2 must be key-sorted prior to evaluating this function. When two keys are equal, elements from TupleList1 are picked before elements from TupleList2.

keyreplace(Key, N, TupleList1, NewTuple) -> TupleList2

Types:
- Key = term()
- N = 1..size(Tuple)
- TupleList1 = TupleList2 = [Tuple]
- NewTuple = Tuple = tuple()

Returns a copy of TupleList1, where the first occurrence of a tuple whose Nth element is Key, if present, is replaced with NewTuple.

keysearch(Key, N, TupleList) -> {value, Tuple} | false

Types:
- Key = term()
- N = 1..size(Tuple)
- TupleList = [Tuple]
- Tuple = tuple()

Searches the list of the tuples TupleList for a tuple whose Nth element is Key. Returns {value, Tuple} if such a tuple is found, or false otherwise.

keysort(N, TupleList1) -> TupleList2
Types:
- \( N = 1..\text{size}(\text{Tuple}) \)
- \( \text{TupleList1} = \text{TupleList2} = [\text{Tuple}] \)
- \( \text{Tuple} = \text{tuple()} \)

Returns a list containing the sorted elements of \( \text{TupleList1} \). Sorting is performed on the \( N \)th element of the tuples.

\( \text{last(List)} \rightarrow \text{Last} \)

Types:
- \( \text{List} = [\text{term()}] \)
- \( \text{Last} = \text{term()} \)

Returns the last element in \( \text{List} \).

\( \text{max(List)} \rightarrow \text{Max} \)

Types:
- \( \text{List} = [\text{term()}] \)
- \( \text{Max} = \text{term()} \)

Returns the maximum element of \( \text{List} \).

\( \text{member(Elem, List)} \rightarrow \text{bool()} \)

Types:
- \( \text{Elem} = \text{term()} \)
- \( \text{List} = [\text{term()}] \)

Returns true if \( \text{Elem} \) is an element of \( \text{List} \), otherwise false.

\( \text{merge(ListOfLists)} \rightarrow \text{List1} \)

Types:
- \( \text{ListOfLists} = [\text{List}] \)
- \( \text{List} = \text{List1} = [\text{term()}] \)

Returns the sorted list formed by merging all the sub-lists of \( \text{ListOfLists} \). All sub-lists must be sorted prior to evaluating this function.

\( \text{merge(List1, List2)} \rightarrow \text{List3} \)

Types:
- \( \text{List1} = \text{List2} = \text{List3} = [\text{term()}] \)

Returns the sorted list formed by merging \( \text{List1} \) and \( \text{List2} \). Both \( \text{List1} \) and \( \text{List2} \) must be sorted prior to evaluating this function.

\( \text{merge(Fun, List1, List2)} \rightarrow \text{List3} \)

Types:
- \( \text{Fun} = \text{fun}(A, B) \rightarrow \text{bool()} \)
- \( \text{List1} = [A] \)
- \( \text{List2} = [B] \)
- 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.

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.

min(List) -> Min

Types:
- List = [term()]
- Min = term()

Returns the minimum element of List.

nth(N, List) -> Elem

Types:
- N = int()
- 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 = int()
- List1 = Tail = [term()]

Returns the Nth tail of List. For example:

> lists:nthtail(3, [a, b, c, d, e]).
  [d, e]

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.
Limitations: A failure will occur if To < From and Incr is positive, or if To > From and Incr is negative, or if Incr == 0 andFrom /= 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 \( \text{Fun} \). \( \text{Fun}(A, B) \) should return true if \( A \) comes before \( B \) in the ordering, false otherwise.

**split**(\( N \), List1) \rightarrow \{\text{List2}, \text{List3}\)

Types:
- \( N = 1..\text{length}(\text{List1}) \)
- \( \text{List1} = \text{List2} = \text{List3} = [\text{term()}] \)

Splits List1 into List2 and List3. List2 contains the first \( N \) elements and List3 the rest of the elements.

**sublist**(List1, \( N \)) \rightarrow \text{List2}

Types:
- \( \text{List1} = \text{List2} = [\text{term()}] \)
- \( N = \text{int()} \)

Returns the first \( N \) elements of List1. It is not an error for \( N \) to exceed the length of the list - in that case the whole list is returned.

**sublist**(List1, Start, Length) \rightarrow \text{List2}

Types:
- \( \text{List1} = \text{List2} = [\text{term()}] \)
- \( \text{Start} = \text{Length} = \text{int()} \)

Returns the sub-list of List1 starting at \( \text{Start} \) and of length \( \text{Length} \). It is not an error if \( \text{Start} > \text{length}(\text{List1}) \) or if \( (\text{Start}+\text{Length}) > \text{length}(\text{List1}) \).

\[
> \text{lists:sublist([1,2,3,4], 2, 2)}.
[2,3]
> \text{lists:sublist([1,2,3,4], 2, 5)}.
[2,3,4]
> \text{lists:sublist([1,2,3,4], 5, 2)}.
[]
\]

**subtract**(List1, List2) \rightarrow \text{List3}

Types:
- \( \text{List1} = \text{List2} = \text{List3} = [\text{term()}] \)

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 removed. For example:

\[
> \text{lists:subtract("123212", "212")}.
"312".
\]

\( \text{lists:subtract}(A,B) \) is equivalent to \( A -- B \).

**suffix**(List1, List2) \rightarrow \text{bool()}

\[
220
\]
Returns true if List1 is a suffix of List2, otherwise false.

\text{sum(List)} \rightarrow \text{number()}

Types:
- \text{List} = \{\text{number()}\}
Returns the sum of the elements in \text{List}.

\text{ukeymerge(N, TupleList1, TupleList2)} \rightarrow \text{TupleList3}

Types:
- \text{N} = 1..\text{size(Tuple)}
- \text{TupleList1} = \text{TupleList2} = \text{TupleList3} = \{\text{Tuple}\}
- \text{Tuple} = \text{tuple()}
Returns the sorted list formed by merging \text{TupleList1} and \text{TupleList2} while removing consecutive duplicates. The sorting is performed on the \text{N}th element of each tuple. Both \text{TupleList1} and \text{TupleList2} must be key-sorted and contain no duplicates prior to evaluating this function. When two keys are equal, elements from \text{TupleList1} are picked before elements from \text{TupleList2}.

\text{ukeysort(N, TupleList1)} \rightarrow \text{TupleList2}

Types:
- \text{N} = 1..\text{size(Tuple)}
- \text{TupleList1} = \text{TupleList2} = \{\text{Tuple}\}
- \text{Tuple} = \text{tuple()}
Returns a list containing the sorted elements of \text{TupleList1} with consecutive duplicates removed. Sorting is performed on the \text{N}th element of the tuples.

\text{umerge(ListOfLists)} \rightarrow \text{List1}

Types:
- \text{ListOfLists} = \{\text{List}\}
- \text{List} = \text{List1} = \{\text{term()}\}
Returns the sorted list formed by merging all the sub-lists of \text{ListOfLists} while removing duplicates. All sub-lists must be sorted and contain no duplicates prior to evaluating this function.

\text{umerge(List1, List2)} \rightarrow \text{List3}

Types:
- \text{List1} = \text{List2} = \text{List3} = \{\text{term()}\}
Returns the sorted list formed by merging \text{List1} and \text{List2} while removing duplicates. Both \text{List1} and \text{List2} must be sorted and contain no duplicates prior to evaluating this function.

\text{umerge(Fun, List1, List2)} \rightarrow \text{List3}

Types:
- \text{Fun} = \text{fun(A, B) -> bool()}

\text{STDLIB Reference Manual}

\text{lists}
- List1 = [A]
- List2 = [B]
- List3 = [A | B]
- A = B = term()

Returns the sorted list formed by merging List1 and List2 while removing consecutive duplicates. 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 equals or comes before B in the ordering, false otherwise.

\[ \text{umerge3(List1, List2, List3)} \rightarrow \text{List4} \]

Types:
- List1 = List2 = List3 = List4 = [term()]

Returns the sorted list formed by merging List1, List2 and List3 while removing duplicates. All of List1, List2 and List3 must be sorted and contain no duplicates prior to evaluating this function.

\[ \text{unzip(List1)} \rightarrow \{\text{List2, List3}\} \]

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.

\[ \text{unzip3(List1)} \rightarrow \{\text{List2, List3, List4}\} \]

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.

\[ \text{usort(List1)} \rightarrow \text{List2} \]

Types:
- List1 = [term()]

Returns a list containing the sorted elements of List1 without duplicates.

\[ \text{usort(Fun, List1)} \rightarrow \text{List2} \]

Types:
- Fun = fun(Elem1, Elem2) -&gt; bool()
STDLIB Reference Manual

- Elem1 = Elem2 = term()
- List1 = List2 = [term()]

Returns a list containing the sorted elements of List1 with consecutive duplicates removed, according to the ordering function Fun. Fun(A, B) should return true if A equals or 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).

Examples:
> lists: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]]

call(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.

call(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.

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.

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.

```erlang
flatmap(Fun, List1) -> List2
```

**Types:**

- Fun = fun(A) -> B
- List1 = [A]
- List2 = [B]
- A = B = term()

`flatmap` behaves as if it had been defined as follows:

```
flatmap(Fun, List1) ->
    append(map(Fun, List1))
```

```erlang
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 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
```

```erlang
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.

map(Fun, List1) -> List2

Types:
  • Fun = fun(A) -> B
  • List1 = [A]
  • List2 = [B]
  • A = B = term()

Takes a function from A to B, and a list of A's and produces a list of B's 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.
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,1,c,d],[2,3,4]

See also splitwith/2 for a different way to partition a list.

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:

splitwidth(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.

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.
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) -> Args
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 167], `rb(3)`
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**

```erlang
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
\[ \text{erf}(X) = \frac{2}{\sqrt{\pi}} \int_0^X e^{-t^2} dt. \]

erfc(X) -> float()

Types:
- X = number()

erfc(X) returns \( 1.0 - \text{erf}(X) \), computed by methods that avoid cancellation for large X.

Bugs

As these are the C library, the bugs are the same.
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 straightforward 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:

```erlang
-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:

```erlang
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 too small to justify complicated ets searches, but on real tables, using select to get exactly the data you want will increase efficiency remarkably.

Let's 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,[#{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").

% ...  

est:select(emp_tab, ests: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 ests:foldr example. In fact it's even easier to read than the ests: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 ests:fun2ms example that one needs to include ms_transform.hrl in the source code, as this is what triggers the parse transformation of the ests: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 ests examples. Let's say one wants to get all the employee numbers of any employee hired before the year 2000. Using ests:match isn't an alternative here as relational operators cannot be expressed there. Once again, an ests:foldr could do it (slowly, but correct):

est: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:

est: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 fold1 example. Let's jump on to writing it using ests:fun2ms
-include_lib("stdlib/include/ms_transform.hrl").

% ...

es: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:

es: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:

es:select(emp_tab, ets:fun2ms(
    fun(#emp{empno = [$0 | Rest]}) ->
    {[$0|Rest],[$1|Rest]}
    end)).

As a matter of fact, this query hits 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}:

es:select(emp_tab, ets:fun2ms(
    fun(#emp{empno = E, surname = "Smith"}) ->
    {guru,E};
    (#emp{empno = E, empyear = Y}) when Y < 1997 ->
    {inventory, E};
    ...)
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 being 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
heets: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,pall).
{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]]}

STDLIB Reference Manual
ms_transform

STDLIB

237
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)).
```

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 esoteric 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 \texttt{is\_list(X)} will be translated to \{\texttt{is\_list, '}$1'$\} \{\texttt{'$1'$} 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 \texttt{fun} will be replaced by match_spec variables in the order of occurrence, so that the fragment \texttt{fun({A,B,C})} will be replaced by \{\texttt{'$1'$, '$2'$, '$3'$} 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 \texttt{fun({A,B}) when is\_atom(A) -> B} end will be translated into \[\{\{\texttt{'$1'$, '$2'$}, \{\texttt{is\_atom, '$1'$}\}, \{\texttt{'$2'$}\}\}\].

Variables that are not appearing in the head are imported from the environment and made into match_spec \texttt{const} expressions. Example from the shell:

\begin{verbatim}
1> X = 25.
25
2> ets:fun2ms(fun({A,B}) when A > X -> B end).
\[
[[\{\texttt{'$1'$, '$2'$}\}, [[\texttt{'>'$, \texttt{'$1'$}, \{\texttt{const}, 25\}\}], \{\texttt{'$2'$}\}]]
\end{verbatim}

Matching with \texttt{=} cannot be used in the body. It can only be used on the top level in the head of the \texttt{fun}. Example from the shell again:

\begin{verbatim}
1> ets:fun2ms(fun({A,[B|C]} = D) when A > B -> D end).
\[
[[\{\texttt{'$1'$, '$2'$, '$3'$}\}, [[\texttt{'>'$, \texttt{'$1'$}, \texttt{\{\texttt{const}, 25\}\}], \{\texttt{'$2'$}\}]]
\end{verbatim}

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 \texttt{$_'} in the resulting match_spec, it is to allow a more natural access to the whole matched object. The pseudo function \texttt{object()} could be used instead, see below. The following expressions are translated equally:

\begin{verbatim}
ets:fun2ms(fun({a,,} = A) -> A end).
ets:fun2ms(fun({a,,}) -> object() end).
\end{verbatim}

The special match_spec variables \texttt{$_'} and \texttt{'$\ast$'} can be accessed through the pseudo functions \texttt{object()} (for \texttt{$_'$}) and \texttt{bindings()} (for \texttt{'$\ast$'}). as an example, one could translate the following \texttt{ets:match\_object/2} call to a \texttt{ets:select} call:

\texttt{ets:match\_object(Table, \{\$1', test, '}$2'$\}).

...is the same as...

\texttt{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 \texttt{ets:select/2} call will conceptually look like this in the resulting code:

\texttt{ets:select(Table, \[\{\{\texttt{'$1'$, test, '$2'$}, [], \{\texttt{$_'$}\}\}\}}}.

Matching on the top level of the \texttt{fun} head might feel like a more natural way to access \texttt{$_'$}, 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

\[
\text{parse_transform}(\text{Forms, Options}) \rightarrow \text{Forms}
\]

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.

\[
\text{transform_from_shell}(\text{Dialect, Clauses, BoundEnvironment}) \rightarrow \text{term()}
\]

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 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.

**See Also**

dict(3) [page 82], gb_trees(3) [page 162]
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.

See Also

gb_sets(3) [page 156], sets(3) [page 283]
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 \texttt{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 \texttt{slave} module. This effects, tty IO, file IO, and code loading.

If the master node fails, the entire pool will exit.

Exports

\begin{verbatim}
start(Name) ->
start(Name, Args) -> Nodes
\end{verbatim}

Types:
- \texttt{Name = atom()}
- \texttt{Args = string()}
- \texttt{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 \texttt{Files} [page 247] below. The start-up procedure fails if the file is not found.

The slave nodes are started with \texttt{slave:start/2,3}, passing along \texttt{Name} and, if provided, \texttt{Args}. \texttt{Name} is used as the first part of the node names, \texttt{Args} is used to specify command line arguments. See \texttt{slave(3)} [page 296].

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.

\begin{verbatim}
attach(Node) -> all\_ready\_attached | attached
\end{verbatim}

Types:
- \texttt{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 [netadm(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.
The `proc_lib` module is used to initialize some useful information when a process starts. The registered names, or the process identities, of the parent process, and the parent ancestors, are stored together with information about the function initially called in the process.

A crash report is generated if the process terminates with a reason other than `normal` or `shutdown`. `shutdown` is used to terminate an abnormal process in a controlled manner. A 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.

The crash report is sent to the `error_logger`. An event handler has to be installed in the `error_logger` event manager in order to handle these reports. The crash report is tagged `crash_report` and the `format/1` function should be called in order to format the report.

### Exports

- `spawn(Fun) -> Pid`
- `spawn(Node,Fun) -> Pid`
- `spawn(Module,Func,Args) -> Pid`
- `spawn(Node,Module,Func,Args) -> Pid`

**Types:**
- `Fun = fun() -> void()`
- `Module = atom()`
- `Func = atom()`
- `Args = [Arg]`
- `Arg = term()`
- `Node = atom()`
- `Pid = pid()`

Spawns a new process and initializes it as described above. The process is spawned using the `spawn` BIFs. The process can be spawned on another `Node`.

- `spawn_link(Fun) -> Pid`
- `spawn_link(Node,Fun) -> Pid`
- `spawn_link(Module,Func,Args) -> Pid`
- `spawn_link(Node,Module,Func,Args) -> Pid`

**Types:**
spawns a new process and initializes it as described above. The process is spawned using the
spawn_opt BIFs. The process can be spawned on another Node.

Types:
- Fun = fun() -> void()
- Module = atom()
- Func = atom()
- Args = [Arg]
- Node = atom()
- Pid = pid()

starts a function, initializes it as described above. The process is then spawned using the
start BIFs. The process can be spawned on another Node.

Types:
- Module = atom()
- Func = atom()
- Args = [Arg]
- Time = integer >= 0 | infinity
- SpawnOpts = list()
- Ret = term() | {error, Reason}
proc_lib:spawn/3 or proc_lib:spawn_link/3, and waits for the process to start. When the process has
started, it must call proc_lib:init_ack(Parent, Ret) or proc_lib:init_ack(Ret),
where Parent is the process that evaluates start. At this time, Ret is returned from
start.

If the start_link function is used and the process crashes before proc_lib:init_ack is
called, {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 process
to start (proc_lib:init_ack). If it has not started within this time, {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/4 BIF. Refer to the erlang module for information about the spawn_opt
options.

init_ack(Parent, Ret) -> void()
init_ack(Ret) -> void()

Types:
• Parent = pid()
• Ret = term()

This function is used by a process that has been started by a proc_lib:start 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
proc_lib:start function. If the init_ack function is not called (e.g. if the init function
fails) and proc_lib:start/3 is used, that function never returns and the parent
hangs forever. This can be avoided by using a time out in the call to start, or by using
start_link.

The following example illustrates how this function and proc_lib:start_link are used.

-module(my_proc).
-export([start_link/0]).
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().
loop() ->
  receive
    ....

format(CrashReport) -> string()

Types:
• CrashReport = void()
Formats a previously generated crash report. The formatted report is returned as a string.

\[
\text{initial\_call(PidOrPinfo)} \rightarrow \{\text{Module, Function, Args} \mid \text{Fun} \mid \text{false}
\]

Types:
- \(\text{PidOrPinfo} = \text{pid()} \mid \{X, Y, Z\} \mid \text{ProcInfo}\)
- \(X = Y = Z = \text{int()}\)
- \(\text{ProcInfo} = \text{[void()]}\)
- \(\text{Module} = \text{atom()}\)
- \(\text{Fun} = \text{fun()} \rightarrow \text{void()}\)
- \(\text{Function} = \text{atom()}\)
- \(\text{Args} = \text{[term()]}\)

Extracts the initial call of a process that was spawned using the spawn functions described above. \(\text{PidOrPinfo}\) can either be a Pid, an integer tuple (from which a pid can be created), or the process information of a process (fetched through an \text{erlang:process\_info/1} function call).

\[
\text{translate\_initial\_call(PidOrPinfo)} \rightarrow \{\text{Module, Function, Arity} \mid \text{Fun}\}
\]

Types:
- \(\text{PidOrPinfo} = \text{pid()} \mid \{X, Y, Z\} \mid \text{ProcInfo}\)
- \(X = Y = Z = \text{int()}\)
- \(\text{ProcInfo} = \text{[void()]}\)
- \(\text{Module} = \text{atom()}\)
- \(\text{Fun} = \text{fun()} \rightarrow \text{void()}\)
- \(\text{Function} = \text{atom()}\)
- \(\text{Arity} = \text{int()}\)

Extracts the initial call of a process which was spawned using the spawn functions described above. If the initial call is to one of the system defined behaviours such as \text{gen\_server} or \text{gen\_event}, it is translated to more useful information. If a \text{gen\_server} is spawned, the returned \text{Module} is the name of the callback module and \text{Function} is \text{init} (the function that initiates the new server).

A supervisor and a supervisor bridge are also \text{gen\_server} processes. In order to return information that this process is a supervisor and the name of the call-back module, \text{Module} is \text{supervisor} and \text{Function} is the name of the supervisor callback module. \text{Arity} is 1 since the \text{init/1} function is called initially in the callback module.

By default, \{\text{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 \text{proc\_lib} module.

\(\text{PidOrPinfo}\) can either be a Pid, an integer tuple (from which a pid can be created), or the process information of a process (fetched through an \text{erlang:process\_info/1} function call).

This function is used by the \text{c:i/0} and \text{c:regs/0} functions in order to present process information.

\[
\text{hibernate(Module, Function, Arguments)}
\]

Types:
hibernate/3 gives a way to put a process started using one of the functions in the proc_lib module into a wait state where its memory allocation has been reduced as much as possible, which is useful if the process does not expect to receive any messages in the near future.

The process will be awakened when a message is sent to it, and control will resume in Module:Function with the arguments given by ArgumentList.

If the process has any message in its message queue, the process will be awakened immediately in the same way as described above.

Note: The actual work is done by the erlang:hibernate/3 BIF. To ensure that exception handling and logging continues to work in a process started by proc_lib, always use proc_lib:hibernate rather than erlang:hibernate/3.

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 occurrence 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.

\[
\text{get\_value(Key, List) -> term()}
\]

Types:
- Key = term()
- List = [term()]

Equivalent to \text{get\_value(Key, List, undefined)}.

\[
\text{get\_value(Key, List, Default) -> term()}
\]

Types:
- Key = term()
- Default = term()
- List = [term()]

Returns the value of a simple key/value property in List. If \text{lookup(Key, List)} would yield \{Key, Value\}, this function returns the corresponding Value, otherwise Default is returned.

See also: \text{get\_all\_values/2, get\_bool/2, get\_value/1, lookup/2}.

\[
\text{is\_defined(Key, List) -> bool()}
\]

Types:
- Key = term()
- List = [term()]

Returns true if List contains at least one entry associated with Key, otherwise false is returned.

\[
\text{lookup(Key, List) -> none | tuple()}
\]

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: \text{get\_bool/2, get\_value/2, lookup\_all/2}.

\[
\text{lookup\_all(Key, List) -> [tuple()}}
\]

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: \text{lookup/2}.

\[
\text{normalize(List, Stages) -> List}
\]

Types:
List = [term()]
Stages = [Operation]
Operation = {aliases, Aliases} | {negations, Negations} | {expand, Expansions}
Aliases = [{Key, Key}]
Negations = [{Key, Key}]
Key = term()
Expansions = [{Property, [term()]}, {Property, [tuple()]}
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()]

property(Property) -> Property
Types:
  • Property = atom() | tuple()

property(Key, Value) -> Property
Types:
  • Key = term()
  • Value = term()
  • Property = atom() | tuple()

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 occurrences of atoms in List to tuples \{Atom, true\}.
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 262] section. A query is stated using Query List Comprehensions (QLCs). These 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. The answers to a query are determined by data in QLC tables that fulfill the constraints expressed by the QLCs of the query.

QLCs should not be confused with the language construct query ListComprehension end used by Mnemosyne. The qlc module recognizes the first argument of every call to qlc:q/1,2 as QLCs, and nothing else. The semantics are very different: Mnemosyne uses ideas borrowed from Prolog while the QLCs introduced in this module are all Erlang. 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. It is the aim of this module to replace Mnemosyne and to be more versatile by means of QLC tables.

While ordinary list comprehensions evaluate to lists, calling qlc:q/1,2 [page 267] returns a Query Handle. To obtain all the answers to a query, qlc:eval/1,2 [page 265] should be called with the query handle as first argument. Query handles are essentially 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 264] 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 \texttt{bool()}. Generators have the form \texttt{Pattern\/-ListExpression}, where \texttt{ListExpression} is an expression evaluating to a query handle or a list. Query handles are returned from \texttt{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 \texttt{false}, or collecting the template when all filters return \texttt{true}.

**Common options**

The following options are accepted by \texttt{cursor/2, eval/2, fold/4, and info/2}:

- \{\texttt{unique\_all, true}\} adds a \{\texttt{unique, true}\} option to every list expression of the query. Default is \{\texttt{unique\_all, false}\}. The option \texttt{unique\_all} is equivalent to \{\texttt{unique, true}\}.
- \{\texttt{cache\_all, true}\} adds a \{\texttt{cache, true}\} option to every list expression of the query except tables and lists. Default is \{\texttt{cache\_all, false}\}. The option \texttt{cache\_all} is equivalent to \{\texttt{cache, true}\}.

**Common data types**

- \texttt{QueryCursor = \{qlc\_cursor, term()\}}
- \texttt{QueryHandle = \{qlc\_handle, term()\}}
- \texttt{QueryHandleOrList = QueryHandle | list()}
- \texttt{Answers = \{Answer\}}
- \texttt{Answer = term()}
- \texttt{AbstractExpression = -parse trees for Erlang expressions, see the \{abstract format\} documentation in ERTS User's Guide-}
- \texttt{MatchExpression = -match specifications, see the \{match specification\} documentation in the ERTS User's Guide and ms\_transform(3) [page 231]-}
- \texttt{SpawnOptions = default | spawn options()}
- \texttt{SortOptions = \{SortOption\} | SortOption}
- \texttt{SortOption = \{compressed, bool()\} | \{no\_files, NoFiles\} | \{order, Order\} | \{size, Size\} | \{tmpdir, TempDirectory\} | \{unique, bool()\} -see file\_sorter(3) [page 142]-}
- \texttt{Order = ascending | descending | OrderFun}
• OrderFun = fun(Term, Term) -> bool()
• TempDirectory = "" | filename()
• Size = int() > 0
• NoFiles = int() > 1
• KeyPos = int() > 0 | [int() > 0]
• bool() = true | false
• filename() = -see filename(3) [page 150]-
• spawn_options() = -see [ erlang(3)]-

Future plans

Support for faster join of two tables will be added not later than in R11. Depending on preferences and priorities some high level optimizations may be added in the future.

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 262] 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
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:

\[
\text{qlc}:\text{q}([X \mid X <- \text{qlc:append}(\text{QH1}, \text{QH2})], \{\text{unique}, \text{true}\})
\]

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:

\[
\text{qlc:sort}(\text{qlc:q}([X \mid X <- \text{qlc:append}(\text{QH1}, \text{QH2})], \{\text{unique}, \text{true}\}))
\]

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:

\[
\begin{align*}
C &= \text{qlc:cursor}(\text{qlc:q}([X \mid X <- \text{qlc:append}(\text{QH1}, \text{QH2})], \{\text{unique}, \text{true}\})) , \\
R &= \text{qlc:next_answers}(C, 5), \\
\text{ok} &= \text{qlc:delete_cursor}(C), \\
R.
\end{align*}
\]

Query list comprehensions are convenient for stating conditions on data from two or more tables. An example that does a natural join on two tables on position 2:

\[
\text{qlc:q}([\{X1, X2, X3, Y1\} \mid \{X1, X2, X3\} <- \text{QH1}, \\
\{Y1, Y2\} <- \text{QH2}, \\
X2 =:= Y2])
\]

If QH1 and QH2 both are tables and X2 or Y2 is a key or index position then the join can be done quickly by looking up key values. In this first version of the `qlc` module this has not yet been implemented. Instead the filter will always 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 262] 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 next 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:

\[
\text{QH2}' = \text{qlc:q}([X \mid X <- \text{QH2}], \{\text{cache}, \text{true}\})
\]

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.

There is an option cache_all that can be set to true when evaluating a query. It adds a cache 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 267] function the implementation of a QLC table for the gb_trees [page 162] module is given:

```erlang
-module(gb_table).
-import(gb_trees, [iterator/1, lookup/2, next/1]).
-export([table/1]).

table(T) ->
    TF = fun() -> qlc_next(next(iterator(T))) end,
    InfoFun = fun(num_of_objects) -> 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) ->
                Vals = a_few(T),
                    [{gb_trees, from_orrdict, [Vals]}];
                ({lookup, 1, KeyValues}) ->
                    ValsS = io_lib:format("gb_trees:from_orrdict(~w)", [a_few(T)]),
                        io_lib:format("lists:flatmap(fun(K) -> "
                                "case gb_trees:lookup(K, ~s) of 
                                "{value, V} -> [{K,V}];none -> [] end 
                                "end, ~w", [ValsS, KeyValues])
                            end,
                qlc:table(TF, [{info_fun, InfoFun}, {format_fun, FormatFun},
                                {lookup_fun, LookupFun}]).
```
qlc_next([{X, V, S}]) ->
    [{X, V} | fun() -> qlc_next(next(S)) end];
qlc_next(none) ->
    [].

a_few(T) ->
    a_few(iterator(T), 7).
a_few(_I, 0) -> more;
a_few(I0, N) ->
    case next(I0) of
    {X, V, I} ->
        [{X, V} | a_few(I, N-1)];
    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. It 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 index 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 key is always preferred 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, bool()] | cache_all | {spawn_options, SpawnOptions} | {unique_all, bool()}

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, bool()] | cache_all | {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, bool()] | cache_all | {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, bool()] | cache_all | {unique_all, bool()} | unique_all
  ReturnOption = [flat, bool()] | {format, Format} | {n_elements, NElements}
  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. 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.

```erlang
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.

```erlang
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 \texttt{file_sorter:keysort/4} [page 142] 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.

```erlang
next_answers(QueryCursor [, NumberOfAnswers]) -> Answers | Error
```

Types:
- \(NumberOfAnswers\) = all_remaining | int() > 0
- \(Error\) = \{error, module(), Reason\}
- \(Reason\) = as returned by \texttt{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 [].

\[
\text{q(QueryListComprehension [, Options])} \rightarrow \text{QueryHandle}
\]

Types:
- QueryListComprehension = literal query list comprehension-
- Options = [Option] | Option
- Option = {maxLookup, MaxLookup} | {cache, bool()} | cache | {unique, bool()} | unique
- MaxLookup = int() >= 0 | infinity

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

\[
\text{-include_lib("stdlib/include/qlc.hrl").}
\]

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:

\[
\ldots
A = [X || \{X\} <- [[1],[2]]],
QH = qlc:q(A),
\ldots
\]

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, true\} 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, true\}.

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
If the unique option is combined with the cache option, two ETS tables are used, but the full answers are stored in one table only.

Sometimes (see qlc:table/2 [page 270] below) traversal of tables can be done by looking up key values, which is supposed 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.

```prolog
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)]).
   qlc:q([ X ||
             [{X,Y},_] <-
               lists:flatmap(fun (K) ->
                             case gb_trees:lookup(K,gb_trees:from_orddict([])) of
                             {value,V} -> [{K,V}];
                             none -> []
                           end,
                           [{1,a},{1,b},{1,c},{2,a},{2,b},{2,c}]),
             (X =:= 1) or (X =:= 2),
             (Y =:= a) or (Y =:= b) or (Y =:= c)
           ]).
ok
```

In this example using the gb_table module from the Implementing a QLC table [page 262] section there are six keys to look up: \{a\}, \{b\}, \{c\}, \{a\}, \{b\}, and \{c\}. The reason is that the two elements of the key \{X,Y\} are matched separately.

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.

```prolog
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_sort:sort/3 [page 142] 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.

```prolog
string_to_handle(QueryString [], Options [], Bindings]) -> QueryHandle | Error
```

Types:
- QueryString = string()
- Options = [Option] | Option
STDLIB Reference Manual

Option = \{max_lookup, MaxLookup\} \{cache, bool()\} \{unique, bool()\} \{lookup, MaxLookup\}

Option = f\max lookup, MaxLookupg \| f\cache, bool()g \| f\cache, bool()g \| f\unique, bool()g

MaxLookup = int() \(\geq 0\) or infinity

Bindings = as returned by erl_eval:bindings/1-

Error = f\error, module(), Reasong

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 \| fun(MatchExpression) -> Objects
- TraverseFun0 = fun() -> Objects
- Objects = [] \| [term()] \| Objects
- 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(SelectedO objects) -> FormattedTable
- SelectedO objects = all \| \{match_spec, MatchExpression\} \| \{lookup, \{Position, Keys\}\}
- FormattedTable = \{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) -> [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. 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 262] 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) funs for different tables are evaluated is not specified. Other table access than reading, such as calling `InfoFun`, is assumed to be O K 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 index 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`. `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 index positions by calling `InfoFun(indices)`. If the key position can be used for lookup it is always chosen, otherwise the index position requiring the least number of lookups is chosen. If there is a tie between two index positions the one occurring first in the list returned by `InfoFun` is chosen. Positions requiring more than max_lookup [page 268] 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.
• **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 for displaying the call that created the table's query handle. The default value undefined is displayed as a call to ‘$MOD’:'$FUN'/0, otherwise it is up to FormatFun to present the selected objects in a suitable way. 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). The argument to FormatFun describes the optimizations done as a result of analyzing the filter(s). The possible values are:

• `{lookup, Position, Keys}`. 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`. No optimization was found. A match specification matching all objects will be used if TraverseFun is unary.

See ets(3) [page 140], dets(3) [page 80] and [mnesia(3)] for the various options recognized by table/1,2 in respective module.

**See Also**

dets(3) [page 66], [Erlang Reference Manual], erl_eval(3) [page 100], [erlang(3)], ets(3) [page 122], [file(3)], file_sorter(3) [page 142], [mnemosyne(3)], [mnesia(3)], [Programming Examples], shell(3) [page 286]
queue

Erlang Module

This module implements FIFO queues in an efficient manner. All operations has an amortised $O(1)$ running time, except \texttt{len/1}, \texttt{reverse/1}, \texttt{join/2} and \texttt{split/2} that probably are $O(n)$.

Exports

\begin{verbatim}
cons(Item, Q1) -> Q2
  Types:
  • Item = term()
  • Q1 = Q2 = queue()
  Inserts Item at the head of queue Q1. Returns the new queue Q2.

daeh(Q) -> Item
  The same as \texttt{last(Q)} and the opposite of \texttt{head(Q)}.

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 be the head item of the queue.

head(Q) -> Item
  Types:
  • Item = term()
  • Q = queue()
  Returns Item from the head of queue Q.
  Fails with reason \texttt{empty} if Q is empty.

in(Item, Q1) -> Q2
  Types:
  • Item = term()
  • Q1 = Q2 = queue()
\end{verbatim}
Inserts `Item` at the tail of queue `Q1`. Returns a new queue `Q2`. This is the same as `snoc(Q1, Item)`.

```q
in_r(Item, Q1) -> Q2
```

Types:
- `Item = term()`
- `Q1 = Q2 = queue()`

Inserts `Item` at the head of queue `Q1`. Returns a new queue `Q2`. This is the same as `cons(Item, Q1)`.

```
init(Q1) -> Q2
```

Types:
- `Item = term()`
- `Q1 = Q2 = queue()`

Returns a queue `Q2` that is the result of removing the last item from `Q1`. This is the opposite of `tail(Q1)`.

Fails with reason `empty` if `Q1` is empty.

```
is_empty(Q) -> true | false
```

Types:
- `Q = queue()`

Tests if `Q` is empty and returns `true` if so and `false` otherwise.

```
join(Q1, Q2) -> Q3
```

Types:
- `Q1 = Q2 = Q3 = queue()`

Returns a queue `Q3` that is the result of joining `Q1` and `Q2` with `Q1` before (at the head) `Q2`.

```
lait(Q1) -> Q2
```

The same as `init(Q1)` and the opposite of `tail(Q1)`.

```
last(Q) -> Item
```

Types:
- `Item = term()`
- `Q = queue()`

Returns the last item of queue `Q`. This is the opposite of `head(Q)`.

Fails with reason `empty` if `Q` is empty.

```
len(Q) -> N
```

Types:
- `Q = queue()`
- `N = integer()`
Calculates and returns the length of queue Q.

new() -> Q

Types:
- Q = queue()

Returns an empty queue.

out(Q1) -> Result

Types:
- Result = \{\{value, Item\}, Q2\} \mid \{empty, Q1\}
- Q1 = Q2 = queue()

Removes the head item from 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.

out_r(Q1) -> Result

Types:
- Result = \{\{value, Item\}, Q2\} \mid \{empty, Q1\}
- Q1 = Q2 = queue()

Removes the last item from 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.

reverse(Q1) -> Q2

Types:
- Q1 = Q2 = queue()

Returns a queue Q2 that contains the items of Q1 in the reverse order.

snoc(Q1, Item) -> Q2

Types:
- Item = term()
- Q1 = Q2 = queue()

Inserts Item as the last item of queue Q1. Returns the new queue Q2. This is the opposite of cons(Item, Q1).

split(N, Q1) -> {Q2, Q3}

Types:
- N = integer()
- Q1 = Q2 = Q3 = queue()

Splits Q1 into a queue Q2 of length N with items from the head end, and a queue Q3 with the rest of the items.
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.

to_list(Q) -> list()

Types:
- Q = queue()

Returns a list of the items in the queue, with the head item of the queue as the head of the list.
random

Erlang Module


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.

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 $N \geq 1$ 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.
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 = \{ \text{match, Matches} \} \lor \{ \text{error, errordesc()} \}

Finds all non-overlapping matches of the expression RegExp in String. It returns as follows:

\{ \text{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 \{ \text{Start, Length} \}, where \text{Start} is the starting position of the match, and \text{Length} is the length of the matching string.

\{ \text{error, Error} \} if there was an error in RegExp.

\texttt{sub(String, RegExp, New)} -> \texttt{SubRes}

Types:
- String = RegExp = New = string()
- SubRes = \{ \text{ok, NewString, RepCount} \} \lor \{ \text{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:

\{ \text{ok, NewString, RepCount} \} if RegExp is correct. RepCount is the number of replacements which have been made (this will be either 0 or 1).

\{ \text{error, Error} \} if there is an error in RegExp.

\texttt{gsub(String, RegExp, New)} -> \texttt{SubRes}

Types:
- String = RegExp = New = string()
- SubRes = \{ \text{ok, NewString, RepCount} \} \lor \{ \text{error, errordesc()} \}
- RepCount = integer()

The same as \texttt{sub}, except that all non-overlapping occurrences of a substring matching RegExp in String are replaced by the string New. It returns:

\{ \text{ok, NewString, RepCount} \} if RegExp is correct. RepCount is the number of replacements which have been made.

\{ \text{error, Error} \} if there is an error in RegExp.

\texttt{split(String, RegExp)} -> \texttt{SplitRes}

Types:
- String = RegExp = string()
- SubRes = \{ \text{ok, FieldList} \} \lor \{ \text{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 = 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 RegExp is correct and RE is 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...
| 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
\n 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-zA-Z][0-9a-zA-Z]*
Variables [A-Z][0-9a-zA-Z]*
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 also be useful when generating regular expressions, or when they are entered other than with Erlang strings.
Sets are collections of elements with no duplicate elements. The representation of a set is not defined.

Exports

\[
\text{new()} -> \text{Set}
\]

Types:
- \(\text{Set} = \text{set}()\)

Returns a new empty ordered set.

\[
\text{is_set(Set)} -> \text{bool()}
\]

Types:
- \(\text{Set} = \text{term}()\)

Returns true if Set is an ordered set of elements, otherwise false.

\[
\text{size(Set)} -> \text{int()}
\]

Types:
- \(\text{Set} = \text{term}()\)

Returns the number of elements in Set.

\[
\text{to_list(Set)} -> \text{List}
\]

Types:
- \(\text{Set} = \text{set}()\)
- \(\text{List} = [\text{term}()]\)

Returns the elements of Set as a list.

\[
\text{from_list(List)} -> \text{Set}
\]

Types:
- \(\text{List} = [\text{term}()]\)
- \(\text{Set} = \text{set}()\)

Returns an ordered set of the elements in List.

\[
\text{is_element(Element, Set)} -> \text{bool()}
\]
Types:
- **Element** = term()
- **Set** = set()

Returns true if Element is an element of Set, otherwise false.

```erlang
add_element(Element, Set1) -> Set2
```
Types:
- **Element** = term()
- **Set1 = Set2 = set()**

Returns a new ordered set formed from Set1 with Element inserted.

```erlang
del_element(Element, Set1) -> Set2
```
Types:
- **Element** = term()
- **Set1 = Set2 = set()**

Returns Set1, but with Element removed.

```erlang
union(Set1, Set2) -> Set3
```
Types:
- **Set1 = Set2 = Set3 = set()**

Returns the merged (union) set of Set1 and Set2.

```erlang
union(SetList) -> Set
```
Types:
- **SetList = [set()]**
- **Set = set()**

Returns the merged (union) set of the list of sets.

```erlang
intersection(Set1, Set2) -> Set3
```
Types:
- **Set1 = Set2 = Set3 = set()**

Returns the intersection of Set1 and Set2.

```erlang
intersection(SetList) -> Set
```
Types:
- **SetList = [set()]**
- **Set = set()**

Returns the intersection of the non-empty list of sets.

```erlang
subtract(Set1, Set2) -> Set3
```
Types:
- **Set1 = Set2 = Set3 = set()**
Returns only the elements of Set1 which are not also elements of Set2.

```prolog
is_subset(Set1, Set2) -> bool()
```

Types:
- Set1 = Set2 = set()
Returns true when every element of Set1 is also a member of Set2, otherwise false.

```prolog
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.

```prolog
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 243], gb_sets(3) [page 156]
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`.

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

```erlang
-inclue_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.

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. 

```
estrider 1> erl
Erlang (BEAM) emulator version 5.3 [hipe] [threads:0]

Eshell V5.3 (abort with ^Q)
1> Str = "abcd".
    "abcd"
2> L = length(Str).
    4
3> Descriptor = {L, list_to_atom(Str)}.
    {4, abcd}
4> L.
    4
5> b().
    Descripor = {4, abcd}
    L = 4
    Str = "abcd"
    ok
6> f(L).
    ok
7> b().
    Descripor = {4, abcd}
    Str = "abcd"
    ok
8> f(L).
    ** 1: variable 'L' is unbound **
9> {L, _} = Descripor.
    {4, abcd}
10> L.
    ** exited: [{badmatch,[{4,abcd}],[{erl_eval,expr,3}]}] **
12> P.
    ** 1: variable 'P' is unbound **
13> Descripor.
    {4, abcd}
14> [P, Q] = Descripor.
    {4, abcd}
15> P.
    4
16> f().
    ok
17> put(aa, hello).
    undefined
18> get(aa).
    hello
19> Y = test1:demo(1).
```


20> get().
   [{aa,worked}]
21> put(aa, hello).
   worked
22> Z = test1:demo(2).
   =ERROR REPORT==== 19-Feb-2003::10:04:14 ===
   Error in process 0.40.0 with exit value: {{badmatch,1},
   [{test1 демо,1},
    {erl_eval,expr,4},{shell,eval_loop,2}]}
   ** exited: {{badmatch,1},
    [{test1 демо,1},
     {erl_eval,expr,4},{shell,eval_loop,2}]} **
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\n").
   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{}.
   ** exited: {undef,[{shell_default,val,[]}
    {erl_eval,do_apply,5},
    {erl_eval,expr_list,6},
    {erl_eval,expr,5},
    {shell,eval_loop,2}]} **
35> #rec[b = 3].
   {rec,undefined,3}
36> rp(v(-1)).
   #rec[a = undefined, 
    b = 3]
   ok
37> rd(rec, {f = orddict:new()}).
   rec
38> rp(#rec{}).
   #rec[f = []]
ok
39> rd(rec, [c]), A.
** 1: variable 'A' is unbound **
40> rp(#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
** exited: killed **
42> halt().

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 shows that L is no longer bound to a 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 "** exited:" 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:

```erlang
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:

```erlang
-record(rec, {a, b = val()}).
```

```
v() ->
    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:

```erlang
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 "** exited: killed **" message is shown.

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 behaviour 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 behaviour). The latter sets `^G` to terminate the current shell and start a new one. JCL mode can not 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, it calls a predicate function (with information about the function call in question). This predicate function returns `true` to let the shell go ahead with the evaluation, or `false` to abort it. There are two possible predicate functions for the user to implement:

```erlang
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.

```erlang
non_local_allowed(FuncSpec, ArgList, State) -> {true,NewState} | {false,NewState}
```

to determine if the call to non-local function `FuncSpec` (`{Module,Func}` or a fun) with arguments `ArgList` should be allowed.

These predicate functions are in fact called from local and non-local evaluation function handlers, described in the `erl_eval` manual page. (Arguments in `ArgList` are evaluated before the predicates 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 predicate
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 predicate function module. Example (with predicate functions implemented in `pred.module`):
  
  ```
  $ erl -stdlib restricted_shell pred
  ```

- 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 predicate functions can not be used to allow or disallow execution of functions called from compiled code (only functions called from expressions entered at the shell prompt).

Exports

```prolog
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.

```prolog
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.

```prolog
start_restricted(Module) -> ok
```  
Types:

- `Module = atom()`  
Exits a normal shell and starts a restricted shell. `Module` specifies the module for the predicate functions `local_allowed/3` and `non_local_allowed/3`. The function is meant to be called from the shell.

```prolog
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 \texttt{rsh}. The user must be allowed to \texttt{rsh} to the remote hosts without being prompted for a password. This can be arranged in a number of ways (refer to the \texttt{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 \texttt{rsh} program.

An alternative to the \texttt{rsh} program can be specified on the command line to \texttt{erl} as follows: \texttt{-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

\begin{verbatim}
start(Host) ->
start(Host, Name) ->
start(Host, Name, Args) -> \{ok, Node\} | \{error, Reason\}
\end{verbatim}

Types:
\begin{itemize}
\item \texttt{Host} = Name = atom()
\item \texttt{Args} = string()
\item \texttt{Node} = node()
\item \texttt{Reason} = timeout | \texttt{no_rsh} | \{already_running, Node\}
\end{itemize}
Starts a slave node on the host \texttt{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 \texttt{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 \texttt{Args} argument is used to set \texttt{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 \texttt{H} with the node name \texttt{Name@H}, and we also want the slave node to have the following properties:

- directory \texttt{Dir} should be added to the code path;
- the Mnesia directory should be set to \texttt{M};
- the unix \texttt{DISPLAY} environment variable should be set to the display of the master node.

The following code is executed to achieve this:

\begin{verbatim}
E = " -env DISPLAY " ++ net_adm:localhost() ++ ":0 ",
Arg = "-mnesia_dir " ++ M ++ " -pa " ++ Dir ++ E,
slave:start(H, Name, Arg).
\end{verbatim}

If successful, the function returns \{ok, Node\}, where \texttt{Node} is the name of the new node. Otherwise it returns \{error, Reason\}, where \texttt{Reason} can be one of:

- \texttt{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.

- \texttt{no_rsh} There is no \texttt{rsh} program on the computer.

- \texttt{already_running, Node} A node with the name \texttt{Name@Host} already exists.

\begin{verbatim}
start_link(Host) ->
start_link(Host, Name) ->
start_link(Host, Name, Args) -> {ok, Node} | {error, Reason}
\end{verbatim}

Types:

- Host = Name = atom()
- Args = string()
- Node = node()
- Reason = timeout | no_rsh | \{already_running, Node\}

Starts a slave node in the same way as \texttt{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 \texttt{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.
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 \( B = \{ x : S(x) \} \). 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 \( X \times Y \), is the set \( \{ a : a = (x, y) \text{ for some } x \in X \text{ and } y \in Y \} \). A relation is a subset of \( X \times Y \). Let \( R \) be a relation. The fact that \( (x, y) \) belongs to \( R \) is written as \( x \text{R}_x \). 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 \( \{ x : x \text{R}_x \text{ for some } x \} \). The converse of \( R \) is the set \( \{ a : a = (y, x) \text{ for some } (x, y) \in R \} \). If \( A \) is a subset of \( X \), then the image of \( A \) under \( R \) is the set \( \{ y : x \text{R}_x \text{ for some } x \in A \} \), and if \( B \) is a subset of \( Y \), then the inverse image of \( B \) is the set \( \{ x : x \text{R}_x \text{ for some } y \in B \} \). 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 \( x \text{R}_x \) if and only if there exists an element \( y \in Y \) such that \( x \text{R}_y \) and \( y \text{S}_x \). The restriction of \( R \) to \( A \) is the set \( S \) defined so that \( x \text{R}_x \) if and only if there exists an element \( x \in A \) such that \( x \text{R}_y \). 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 \((x,y)\in F \) 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 \(\{x[i]: (x[1],...,x[i],...,x[n]) \in R \text{ for some } x[j]\in X[j], 1<=j<=n \text{ and not } i=j\}\). 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 \(S\) is defined to be the set \(\{z: z=((x[1],...,x[n]), (y[1],...,y[n])) \text{ for some } (x[1],...,x[n])\in S \text{ and for some } (x[i],y[i]) \in R[i], 1<=i<=n\}\). 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]) in R and for some (y[1],...,y[m]) in S 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 310] 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 317], 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 = 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(I, 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}

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,

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

\texttt{a\_function(Tuples [, Type]) \rightarrow Function}

Types:
- Function = function()
- Tuples = [tuple()]
- Type = type()

Creates a function [page 300]. \texttt{a\_function(F,T)} is equivalent to \texttt{from\_term(F,T)}, if the result is a function. If no type [page 301] is explicitly given, \texttt{([atom,atom])} is used as type of the function.

\texttt{canonical\_relation(SetOfSets) \rightarrow 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 300] of a set \(X\) and \(R\) is the equivalence relation in \(X\) induced by SetOfSets, then the returned relation is the canonical map [page 300] from \(X\) onto the equivalence classes with respect to \(R\).

\begin{verbatim}
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]]]
\end{verbatim}

\texttt{composite(Function1, Function2) \rightarrow Function3}

Types:
- Function1 = Function2 = Function3 = function()

Returns the composite [page 300] of the functions Function1 and Function2.

\begin{verbatim}
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]]
\end{verbatim}

\texttt{constant\_function(Set, AnySet) \rightarrow Function}

Types:
- AnySet = anyset()
- Function = function()
- Set = set()

Creates the function [page 300] 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 299] 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 299] 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 300] 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 301] 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 299] 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 299] 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).

[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).

[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 301]. 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 299] of BinRel1 such that for each element E in Set that does not belong to the domain [page 299] 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 300]. family(F,T) is equivalent to from_term(F,T), if the result is a family. If no type [page 301] 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 300], then Family3 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 300] 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 299] 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 300] 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 299] 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 300] 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 299] of Family1[i].

If Family1[i] is an empty set for some i, then the process exits with a badarg message.

```prolog
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 300], 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].

```prolog
1> F1 = sofs:family([[a,[[1,2]],[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 300] 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.

```prolog
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 300] 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 299] of Family1[i].
family_specification(Fun, Family1) -> Family2

Types:
- Fun = specFun()
- Family1 = Family2 = family()

If Family1 is a family [page 300], then Family2 is the restriction [page 299] 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 301] of Family1[i], otherwise Fun is applied to Family1[i].

family_to_digraph(Family [, GraphType]) -> Graph

Types:
- Graph = digraph()
- Family = family()
- GraphType = -see digraph(3)-

Creates a directed graph from the family [page 300] Family. For each pair \((a, b[1],...,b[n])\) of Family, the vertex a as well the edges \((a,b[i])\) for \(1\leq i\leq n\) 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 300], 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([1],{1},2,[3]),[c,2,3]), R = sofs:family_to_relation(F), sofs:to_external(R), [{b,1},{c,2},{c,3}]
family_union(Family1) \rightarrow Family2

Types:
\begin{itemize}
  \item Family1 = Family2 = family()
\end{itemize}

If Family1 is a family [page 300] 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 299] of Family1[i].

\begin{verbatim}
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,[]}]
\end{verbatim}

family_union(F) is equivalent to family_projection({sofs,union},F).

family_union(Family1, Family2) \rightarrow Family3

Types:
\begin{itemize}
  \item Family1 = Family2 = Family3 = family()
\end{itemize}

If Family1 and Family2 are families [page 300], 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.

\begin{verbatim}
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]]}]
\end{verbatim}

field(BinRel) \rightarrow Set

Types:
\begin{itemize}
  \item BinRel = binary_relation()
  \item Set = set()
\end{itemize}

Returns the field [page 299] of the binary relation BinRel.

\begin{verbatim}
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}]
\end{verbatim}

field(R) is equivalent to union(domain(R), range(R)).

from_external(ExternalSet, Type) \rightarrow AnySet

Types:
\begin{itemize}
  \item ExternalSet = external_set()
  \item AnySet = anyset()
  \item Type = type()
\end{itemize}

Creates a set from the external set [page 301] ExternalSet and the type [page 301] Type. It is assumed that Type is a valid type [page 301] of ExternalSet.

from_sets(ListOfSets) \rightarrow Set
Types:
- Set = set()
-ListOfSets = [anyset()]

Returns the unordered set [page 301] containing the sets of the list ListOfSets.

```erlang
define(S1, sofs:relation([[f, a, 1], [g]]),
       S2, sofs:relation([[f, b, 2], [g]]),
       S, sofs:from_sets([S1, S2]),
       to_external(S).
```

from_sets(TupleOfSets) -> Ordset

Types:
- Ordset = ordset()
- TupleOfSets = tuple-of(anyset())

Returns the ordered set [page 301] containing the sets of the non-empty tuple TupleOfSets.

```erlang
from_term(Term [, Type]) -> AnySet
```

Types:
- AnySet = anyset()
- Term = term()
- Type = type()

Creates an element of Sets [page 301] by traversing the term Term, sorting lists, removing duplicates and deriving or verifying a valid type [page 301] for the so obtained external set. An explicitly given type [page 301] 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:

```erlang
define(S, sofs:from_term(["foo", [1, 1]], ["foo", [2, 2]], [atom, [atom]]),
       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”:

```erlang
define(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]),
       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`.

```erlang
image(BinRel, Set1) -> Set2
Types:
  • BinRel = binary_relation()
  • Set1 = Set2 = set()
Returns the image [page 299] 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]
```

```erlang
intersection(SetOfSets) -> Set
Types:
  • Set = set()
  • SetOfSets = set_of_sets()
Returns the intersection [page 299] 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 299] of Set1 and Set2.

intersection_of_family(Family) -> Set
Types:
  • Family = family()
  • Set = set()
Returns the intersection of the family [page 300] 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]
```

```erlang
inverse(Function1) -> Function2
Types:
  • Function1 = Function2 = function()
Returns the inverse [page 300] 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 299] 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 300] 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 299], 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 299], false otherwise.

is_set(AnySet) -> Bool
Types:
* AnySet = anyset()
  * Bool = bool()

Returns true if AnySet is an unordered set [page 301], 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 301], 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 299] of Set2, false otherwise.

**is_type**(Term) -> Bool

Types:
* Bool = bool()
* Term = term()

Returns true if the term Term is a type [page 301].

**join**(Relation1, I, Relation2, J) -> Relation3

Types:
* Relation1 = Relation2 = Relation3 = relation()
* I = J = integer() > 0

Returns the natural join [page 300] of the relations Relation1 and Relation2 on coordinates I and J.

```prolog
R1 = sofs:relation([f(a,x,1), f(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).
```

**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 300] of the ordered set \{R[i],...,R[n]\} and BinRel1.
1> Ri = sofs:relation([[a,i],[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:
- A Set = set() | ordset()
- NoElements = integer() \geq 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 300] 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 300] 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 300] 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 \{\text{restriction}(F,S1,S2),
\text{drestriction}(F,S1,S2)\}.

\text{partition_family}(\text{SetFun}, \text{Set}) \to \text{Family}

\textbf{Types:}
- \text{Family} = \text{family()}
- \text{SetFun} = \text{set_fun()}
- \text{Set} = \text{set()}

\textbf{Returns the family [page 300]} \text{Family where the indexed set is a partition [page 300]} of
\text{Set such that two elements are considered equal if the results of applying SetFun are the}
\text{same value i. This i is the index that Family maps onto the equivalence class [page 300].}

1> S = sofs:relation([[a,a,a,a],[a,a,b,b],[a,b,b,b]]),
    SetFun = \{\text{external}, \text{fun}(A,\_\_,\_\_) -> [A,\_\_]\ end\},
    F = sofs:partition_family(SetFun, S),
    sofs:to_external(F).
    [[[a,a],[a,a,a]],[[a,b],[a,a,b],[a,b,b,b]]]

\text{product(TupleOfSets)} \to \text{Relation}

\textbf{Types:}
- \text{Relation} = \text{relation()}
- \text{TupleOfSets} = \text{tuple-of(set())}

\textbf{Returns the Cartesian product [page 300]} of the non-empty tuple of sets \text{TupleOfSets}
If (x[1],...,x[n]) is an element of the n-ary relation \text{Relation}, then x[i] is drawn from
\text{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],[a,1],[a,2],[a,2],[b,1],[b,1],[b,2],[b,2]]

\text{product(Set1, Set2)} \to \text{BinRel}

\textbf{Types:}
- \text{BinRel} = \text{binary_relation()}
- \text{Set1} = \text{Set2} = \text{set()}

\textbf{Returns the Cartesian product [page 299]} 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 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 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. relation(R,T) is equivalent to from_term(R,T), if T is a type 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(
)
```

relation_to_family(BinRel) -> Family

Types:
- Family = family()
- BinRel = binary_relation()

Returns the family such that the index set is equal to the domain of the binary relation BinRel, and Family[i] is the image of the set of i under BinRel.

```
316 STDLIB
```
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 300] 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 300] 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 299] of the binary relations BinRel1 and BinRel2.

relative_product1(BinRel1, BinRel2) -> BinRel3

Types:
• BinRel1 = BinRel2 = BinRel3 = binary_relation()
Returns the relative product [page 299] of the converse [page 299] 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 299] 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).

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).

set(Terms [, Type]) -> Set

Types:
• Set = set()
• Terms = [term()]
• Type = type()

Creates an unordered set [page 301]. set(L,T) is equivalent to from_term(L,T), if the result is an unordered set. If no type [page 301] 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 301] 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).

strict_relation(BinRel1) -> BinRel2

Types:
• BinRel1 = BinRel2 = binary_relation()

Returns the strict relation [page 300] 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) -> [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])).
   [[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 299] 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 299] (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 301] 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 301] of an atomic, ordered or unordered set.

union(SetOfSets) -> Set

Types:
- Set = set()
- SetOfSets = set_of_sets()

Returns the union [page 299] of the set of sets SetOfSets.

union(Set1, Set2) -> Set3

Types:
- Set1 = Set2 = Set3 = set()

Returns the union [page 299] of Set1 and Set2.
union_of_family(Family) -> Set

Types:
- Family = family()
- Set = set()

Returns the union of the family [page 300] 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 300] W corresponding to the binary relation BinRel1. Let F be the field [page 299] 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 82], digraph(3) [page 87], orddict(3) [page 242], ordsets(3) [page 243], sets(3) [page 283]
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"]
  ```

- `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()
string

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.

substr(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.  
0.900000  
> string:to_float("3/2=1.5").  
{error,no_float}  
> string:to_float("-1.5eX").  
{-1.50000,"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}

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.

The regular expression functions have been moved to their own module regexp (see
regexp(3) [page 278]). The old entry points still exist for backwards compatibility, but
will be removed in a future release so that users are encouraged to use the module
regexp.

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 $\text{MaxR}$ and $\text{MaxT}$. If more than $\text{MaxR}$ restarts occur within $\text{MaxT}$ seconds, the supervisor terminates all child processes and then itself.

This is the type definition of a child specification:

```plaintext
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()
```

- $\text{Id}$ is a name that is used to identify the child specification internally by the supervisor.
- $\text{StartFunc}$ defines the function call used to start the child process. It should be a module-function-arguments tuple $\{M,F,A\}$ used as $\text{apply}(M,F,A)$.

The start function must create and link to the child process, and should return $\{\text{ok},\text{Child}\}$ or $\{\text{ok},\text{Child},\text{Info}\}$ where $\text{Child}$ is the pid of the child process and $\text{Info}$ an arbitrary term which is ignored by the supervisor.

The start function can also return $\text{ignore}$ 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 $\{\text{error},\text{Error}\}$.

Note that the $\text{start_link}$ functions of the different behaviour modules fulfill the above requirements.

- $\text{Restart}$ 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.
- $\text{Shutdown}$ defines how a child process should be terminated. $\text{brutal_kill}$ means the child process will be unconditionally terminated using $\text{exit}(\text{Child},\text{kill})$. An integer timeout value means that the supervisor will tell the child process to terminate by calling $\text{exit}(\text{Child},\text{shutdown})$ and then wait for an exit signal with reason $\text{shutdown}$ back from the child process. If no exit signal is received within the specified time, the child process is unconditionally terminated using $\text{exit}(\text{Child},\text{kill})$.

If the child process is another supervisor, $\text{Shutdown}$ should be set to $\text{infinity}$ to give the subtree ample time to shutdown.

Note that child processes implemented using the behaviour 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}`.

```erlang
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}`.

```erlang
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 \( \text{error,Error} \) where \( \text{Error} \) is a term containing information about the error.

\[
\text{which_children} (\text{SupRef}) \to \text{[[Id,Child,Type,Modules]]}
\]

**Types:**
- \( \text{SupRef} = \text{Name} \mid \{ \text{Name,Node} \} \mid \{ \text{global,Name} \} \mid \text{pid}() \)
- \( \text{Name} = \text{Node} = \text{atom}() \)
- \( \text{Id} = \text{term()} \mid \text{undefined} \)
- \( \text{Child} = \text{pid()} \mid \text{undefined} \)
- \( \text{Type} = \text{worker} \mid \text{supervisor} \)
- \( \text{Modules} = [\text{Module}] \mid \text{dynamic} \)
- \( \text{Module} = \text{atom}() \)

Returns a list with information about all child specifications and child processes belonging to the supervisor \( \text{SupRef} \).
See `start_child/2` for a description of \( \text{SupRef} \).

The information given for each child specification/process is:

- \( \text{Id} \) - as defined in the child specification or undefined in the case of a simple one for one supervisor.
- \( \text{Child} \) - the pid of the corresponding child process, or undefined if there is no such process.
- \( \text{Type} \) - as defined in the child specification.
- \( \text{Modules} \) - as defined in the child specification.

\[
\text{check_childspecs} ([\text{ChildSpec}]) \to \text{Result}
\]

**Types:**
- \( \text{ChildSpec} = \text{child_spec}() \)
- \( \text{Result} = \text{ok} \mid \{ \text{error,Error} \} \)
- \( \text{Error} = \text{term}() \)

This function takes a list of child specification as argument and returns \( \text{ok} \) if all of them are syntactically correct, or \( \{ \text{error,Error} \} \) otherwise.

**CALLBACK FUNCTIONS**

The following functions should be exported from a supervisor callback module.
Exports

Module: init(Args) \rightarrow 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() \geq 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 167], gen_fsm(3) [page 177], gen_server(3) [page 188], sys(3) [page 338]
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

\[
\begin{align*}
\text{start_link}(\text{Module}, \text{Args}) &\rightarrow \text{Result} \\
\text{start_link}(\text{SupBridgeName}, \text{Module}, \text{Args}) &\rightarrow \text{Result}
\end{align*}
\]

Types:
- \(\text{SupBridgeName} = \{\text{local}, \text{Name}\} \mid \{\text{global}, \text{Name}\}\)
- \(\text{Name} = \text{atom}()\)
- \(\text{Module} = \text{atom}()\)
- \(\text{Args} = \text{term}()\)
- \(\text{Result} = \{\text{ok}, \text{Pid}\} \mid \text{ignore} \mid \{\text{error}, \text{Error}\}\)
- \(\text{Pid} = \text{pid}()\)
- \(\text{Error} = \{\text{already_started}, \text{Pid}\} \mid \text{term}()\)

Creates a supervisor_bridge process, linked to the calling process, which calls \text{Module}:\text{init}/1 to start the subsystem. To ensure a synchronized start-up procedure, this function does not return until \text{Module}:\text{init}/1 has returned.

If \text{SupBridgeName} = \{\text{local}, \text{Name}\} the supervisor_bridge is registered locally as \text{Name using register/2}. If \text{SupBridgeName} = \{\text{global}, \text{Name}\} the supervisor_bridge is registered globally as \text{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 \text{SupBridgeName} the function returns \{\text{error}, \{\text{already_started}, \text{Pid}\}\}, where \text{Pid} is the pid of that process.

\text{Module} is the name of the callback module.

\text{Args} is an arbitrary term which is passed as the argument to \text{Module}:\text{init}/1.

If the supervisor_bridge and the subsystem are successfully started the function returns \{\text{ok}, \text{Pid}\}, where \text{Pid} 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

```erlang
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`.

```erlang
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 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 328], sys(3) [page 338]
This module contains functions for sending system messages used by programs, and messages 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.

cache_code(Name, Module, OldVsn, Extra)
cache_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.

\texttt{print\_log(Debug) \rightarrow void()}

Types:
- \texttt{Debug = \{dbg\_opt()\}}

Prints the logged system events in the debug structure using \texttt{FormFunc} as defined when the event was generated by a call to handle\_debug/4.

\texttt{\Module:system\_continue(Parent, Debug, Misc)}

Types:
- \texttt{Parent = pid()}
- \texttt{Debug = \{dbg\_opt()\}}
- \texttt{Misc = term()}

This function is called from \texttt{sys:handle\_system\_msg/6} when the process should continue its execution (for example after it has been suspended). This function never returns.

\texttt{\Module:system\_terminate(Reason, Parent, Debug, Misc)}

Types:
- \texttt{Reason = term()}
- \texttt{Parent = pid()}
- \texttt{Debug = \{dbg\_opt()\}}
- \texttt{Misc = term()}

This function is called from \texttt{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.

\texttt{\Module:system\_code\_change(Misc, Module, OldVsn, Extra) \rightarrow \{ok, NMisc\}}

Types:
- \texttt{Misc = term()}
- \texttt{OldVsn = undefined | term()}
- \texttt{Module = atom()}
- \texttt{Extra = term()}
- \texttt{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.
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 \text{TRef} below. By using cancel/1, the returned reference can be used to cancel any requested action. A \text{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

\text{start() \rightarrow 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 \text{kernel} for this.

apply\_after(Time, Module, Function, Arguments) \rightarrow \{ok, TRef\} \mid \{error, Reason\}

Types:
- \text{Time = integer()} in Milliseconds
- \text{Module = Function = atom()}
- \text{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) \rightarrow \{ok, TRef\} \mid \{error, Reason\}

send\_after(Time, Message) \rightarrow \{ok, TRef\} \mid \{error, Reason\}

Types:
- \text{Time = integer()} in Milliseconds
- \text{Pid = \{pid() \mid atom()\}}
- \text{Message = \{term()\}}
- \text{Result = \{ok, TRef\} \mid \{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) -> {Time, Value}

Types:
- T1 = T2 = {MegaSecs, Secs, MicroSecs}
- MegaSecs = Secs = MicroSecs = integer()

Calculates the time difference 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

Returns 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!
2>
```

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:

<table>
<thead>
<tr>
<th>Abbrev.</th>
<th>Registry key</th>
</tr>
</thead>
<tbody>
<tr>
<td>hkcr</td>
<td>HKEY_CLASSES_ROOT</td>
</tr>
<tr>
<td>current_user</td>
<td>HKEY_CURRENT_USER</td>
</tr>
<tr>
<td>hkcu</td>
<td>HKEY_CURRENT_USER</td>
</tr>
<tr>
<td>local_machine</td>
<td>HKEY_LOCAL_MACHINE</td>
</tr>
<tr>
<td>hklm</td>
<td>HKEY_LOCAL_MACHINE</td>
</tr>
<tr>
<td>users</td>
<td>HKEY_USERS</td>
</tr>
<tr>
<td>hku</td>
<td>HKEY_USERS</td>
</tr>
<tr>
<td>current_config</td>
<td>HKEY_CURRENT_CONFIG</td>
</tr>
<tr>
<td>hkcc</td>
<td>HKEY_CURRENT_CONFIG</td>
</tr>
<tr>
<td>dyn_data</td>
<td>HKEY_DYN_DATA</td>
</tr>
<tr>
<td>hkdd</td>
<td>HKEY_DYN_DATA</td>
</tr>
</tbody>
</table>

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
- 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**

```prolog
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/2`). This means that after the call, the current key is invalid.

```prolog
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.

```prolog
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.

```prolog
format_error(ErrorId) -> ErrorString
```

Types:
- `ErrorId` = `atom()`
- `ErrorString` = `string()`

Convert an POSIX errorcode to a string (by calling `erl_posix_msg:message`).

```prolog
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`.

```prolog
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)
## Index of Modules and Functions

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Module(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_function/2</td>
<td>sofs, 303</td>
</tr>
<tr>
<td>abcast/2</td>
<td>gen_server, 192</td>
</tr>
<tr>
<td>abcast/3</td>
<td>gen_server, 192</td>
</tr>
<tr>
<td>absname/1</td>
<td>filename, 150</td>
</tr>
<tr>
<td>absname/2</td>
<td>filename, 151</td>
</tr>
<tr>
<td>absname_join/2</td>
<td>filename, 151</td>
</tr>
<tr>
<td>abstract/1</td>
<td>erl_parse, 109</td>
</tr>
<tr>
<td>acos/1</td>
<td>math, 229</td>
</tr>
<tr>
<td>acosh/1</td>
<td>math, 229</td>
</tr>
<tr>
<td>add/2</td>
<td>gb_sets, 157</td>
</tr>
<tr>
<td>add/3</td>
<td>erl_tar, 117</td>
</tr>
<tr>
<td>add/4</td>
<td>erl_tar, 117</td>
</tr>
<tr>
<td>add_binding/3</td>
<td>erl_eval, 101</td>
</tr>
<tr>
<td>add_edge/3</td>
<td>digraph, 87</td>
</tr>
<tr>
<td>add_edge/4</td>
<td>digraph, 87</td>
</tr>
<tr>
<td>add_edge/5</td>
<td>digraph, 87</td>
</tr>
<tr>
<td>add_element/2</td>
<td></td>
</tr>
</tbody>
</table>
timer, 346

arith_op/2
    erl_internal, 104

asin/1
    math, 229

asinh/1
    math, 229

atan/1
    math, 229

atan2/2
    math, 229

atanh/1
    math, 229

attach/1
    pool, 246

attribute/1
    erl_pp, 111

attribute/2
    erl_pp, 111

balance/1
    gb_sets, 157
    gb_trees, 162

basename/1
    filename, 151

basename/2
    filename, 151

bchunk/2
    dets, 67

beam_lib
    chunks/2, 52
    clear_crypto_key_fun/0, 55
    cmp/2, 53
    cmp_dirs/2, 53
    crypto_key_fun/1, 55
    diff_dirs/2, 54
    format_error/1, 55
    info/1, 53
    strip/1, 54
    strip_files/1, 54
    strip_release/1, 54
    version/1, 52

bif/2
    erl_internal, 104

binding/2
    erl_eval, 101

bindings/1
    erl_eval, 101

bool_op/2
    erl_internal, 104

bt/1
    c, 57

c
    bt/1, 57
    c/1, 57
    c/2, 57
    cd/1, 57
    flush/0, 58
    help/0, 58
    i/0, 58
    i/3, 58
    l/1, 58
    lc/1, 58
    ls/0, 58
    ls/1, 58
    m/0, 59
    m/1, 59
    memory/0, 59
    memory/1, 59
    nc/1, 59
    nc/2, 59
    ni/0, 58
    nl/1, 59
    nregs/0, 60
    pid/3, 59
    pwd/0, 60
    q/0, 60
    regs/0, 60
    xm/1, 60

c/1
    c, 57

c/2
    c, 57

calendar
    date_to_gregorian_days/1, 62
    date_to_gregorian_days/3, 62
    datetime_to_gregorian_seconds/2, 62
    day_of_the_week/1, 62
    day_of_the_week/3, 62
    gregorian_days_to_date/1, 62
    gregorian_seconds_to_datetime/1, 62
    is_leap_year/1, 62
    last_day_of_the_month/2, 62
    local_time/0, 63
    local_time_to_universal_time/2, 63
local_time_to_universal_time_dst/2, 63
call/2
   gen_server, 190
call/3
   gen_event, 170
call/4
   gen_event, 170
cancel/1
timer, 346
cancel_timer/1
gen_fsm, 181
cast/2
gen_server, 192
cd/1
c, 57
centre/2
   string, 325
centre/3
   string, 325
change_code/4
   sys, 340
change_code/5
   sys, 340
change_key/2
   win32reg, 350
change_key_create/2
   win32reg, 350
char_list/1
   io_lib, 211
chars/2
   string, 323
check/1
   file_sorter, 146
check/2
   file_sorter, 146
check_childspecs/1
   supervisor, 333
chr/2
   string, 322
chunks/2
   beam_lib, 52
clear_crypto_key_fun/0
   beam_lib, 55
close/1
dets, 68
epp, 98
erl_tar, 117
win32reg, 350
cmp/2
   beam_lib, 53
cmp_dirs/2
   beam_lib, 53
comp_op/2
   erl_internal, 105
compact/1
   proplists, 253
components/1
   digraph_utils, 95
composite/2
   sofs, 303
concat/1
   lists, 214
concat/2
   string, 322
condensation/1
   digraph_utils, 95
cons/2
   queue, 272
constant_function/2
   sofs, 303
converse/1
sofs, 304
copies/2
  string, 324
cos/1
  math, 229
cosh/1
  math, 229
create/1
  pg, 244
create/2
erl_tar, 117
  pg, 244
create/3
erl_tar, 118
crypto_key_fun/1
  beam_lib, 55
cspan/2
  string, 323
current_key/1
  win32reg, 350
cursor/2
  qlc, 264
cyclic_strong_components/1
  digraph_utils, 95
daeh/1
  queue, 272
date_to_gregorian_days/1
  calendar, 62
date_to_gregorian_days/3
  calendar, 62
datetime_to_gregorian_seconds/2
  calendar, 62
day_of_the_week/1
  calendar, 62
day_of_the_week/3
  calendar, 62
debug_options/1
  sys, 342
deep_char_list/1
  io_lib, 211
del_binding/2
  erl_eval, 101
del_edge/2
  digraph, 88
del_edges/2
  digraph, 88
del_element/2
  gb_sets, 157
  sets, 284
del_path/3
  digraph, 88
del_vertex/2
  digraph, 89
del_vertices/2
  digraph, 89
delete/1
  digraph, 89
  ets, 123
delete/2
  dets, 68
  ets, 123
  gb_sets, 157
  gb_trees, 162
  lists, 215
  proplists, 253
delete_all_objects/1
  dets, 68
  ets, 123
delete_any/2
  gb_sets, 157
  gb_trees, 163
delete_child/2
  supervisor, 332
delete_cursor/1
  qlc, 264
delete_handler/3
  gen_event, 171
delete_key/1
  win32reg, 350
delete_object/2
  dets, 68
  ets, 123
delete_value/2
  win32reg, 351
dets
  all/0, 67
  bchunk/2, 67
cyclic_strong_components/1, 95
is_acyclic/1, 95
loop_vertices/1, 95
postorder/1, 96
preorder/1, 96
reachable/2, 96
reachable_neighbours/2, 96
reaching/2, 96
reaching_neighbours/2, 96
strong_components/1, 97
subgraph/3, 97
topsort/1, 97
dirname/1
    filename, 152
domain/1
    sofs, 304
drestriction/2
    sofs, 304
drestriction/3
    sofs, 305
dropwhile/2
    lists, 224
duplicate/2
    lists, 215
e/2
    qlc, 264
edge/2
    digraph, 89
dedges/1
    digraph, 89
dedges/2
    digraph, 89
empty/0
    gb_sets, 158
gb_trees, 163
empty_set/0
    sofs, 305
ensure_dir/1
    filelib, 147
enter/3
    gb_trees, 163
enter_loop/3
    gen_server, 192
enter_loop/4
    gen_fsm, 182
gen_server, 193
enter_loop/5
    gen_fsm, 182
gen_server, 193
enter_loop/6
    gen_fsm, 182
epp
close/1, 98
open/2, 98
open/3, 98
parse_erl_form/1, 98
parse_file/3, 98
equal/2
    string, 322
erase/2
    dict, 82
erf/1
    math, 230
erfc/1
    math, 230
erl_eval
    add_binding/3, 101
    binding/2, 101
    bindings/1, 101
del_binding/2, 101
    expr/2, 100
    expr/3, 100
    expr/4, 100
    expr_list/2, 101
    expr_list/3, 101
    expr_list/4, 101
    exprs/2, 100
    exprs/3, 100
    exprs/4, 100
    new_bindings/0, 101
erl_id_trans
    parse_transform/2, 103
erl_internal
    arith_op/2, 104
    bif/2, 104
    bool_op/2, 104
    comp_op/2, 105
guard_bif/2, 104
    list_op/2, 105
    op_type/2, 105
    send_op/2, 105
type_test/2, 104
erl_list
table/2, 139
test_ms/2, 140
to_dets/2, 141
update_counter/3, 141
update_counter/4, 141
update_counter/6, 141
eval/2
qlc, 264
exit_after/2
timer, 346
exit_after/3
timer, 346
exp/1
math, 229
expand/1
win32reg, 351
expand/2
proplists, 253
expr/1
erl_pp, 112
expr/2
erl_eval, 100
erl_pp, 112
expr/3
erl_eval, 100
erl_pp, 112
expr/4
erl_eval, 100
erl_pp, 112
expr_list/2
erl_eval, 101
expr_list/3
erl_eval, 101
expr_list/4
erl_eval, 101
exprs/1
erl_pp, 112
exprs/2
erl_eval, 100
erl_pp, 112
exprs/3
erl_eval, 100
erl_pp, 112
exprs/4
erl_eval, 100
extension/1
filename, 152
extension/3
sofs, 305
extract/1
erl_tar, 118
extract/2
erl_tar, 118
family/2
sofs, 306
family_difference/2
sofs, 306
family_domain/1
sofs, 306
family_field/1
sofs, 306
family_intersection/1
sofs, 307
family_intersection/2
sofs, 307
family_projection/2
sofs, 307
family_range/1
sofs, 307
family_specification/2
sofs, 308
family_to_digraph/2
sofs, 308
family_to_relation/1
sofs, 308
family_union/1
sofs, 309
family_union/2
sofs, 309
fetch/2
dict, 82
fetch_keys/1
dict, 83
field/1
sofs, 309
file2tab/1
ets, 123
file_size/1
    filelib, 147

file_sorter
    check/1, 146
    check/2, 146
    keycheck/2, 146
    keycheck/3, 146
    keymerge/3, 146
    keymerge/4, 146
    keysort/2, 145
    keysort/3, 145
    keysort/4, 145
    merge/2, 145
    merge/3, 145
    sort/1, 145
    sort/2, 145
    sort/3, 145

filelib
    ensure_dir/1, 147
    file_size/1, 147
    fold_files/5, 147
    is_dir/1, 147
    is_file/1, 147
    is_regular/1, 148
    last_modified/1, 148
    wildcard/1, 148
    wildcard/2, 149

filename
    absname/1, 150
    absname/2, 151
    basename/1, 151
    basename/2, 151
    dirname/1, 152
    extension/1, 152
    first/1
dets, 68
    ets, 124
    first_match/2
regexp, 278
    first/2
filename, 154
    first_match/1
regexp, 278
    find/2
dict, 83
    find_src/1
filename, 154
    find_src/2
filename, 154
    fixtable/1
ets, 124
    fixtable/2
ets, 124
    flatten/1
filename, 154
    lists, 215
    flatten/2
lists, 215
    flatlength/1
lists, 215
    flatten/3
dict, 83
    gb_sets, 158
    sets, 285
    flatten/4
qlc, 265
    fold/3
dict, 83
    gb_sets, 158
    sets, 285
    fold/4
qlc, 265
    fold_files/5
filelib, 147
    foldl/3
dets, 69
    ets, 124
    lists, 225
    foldr/3
dets, 69
    ets, 124
    lists, 225
    foreach/2
lists, 226
form/1
erl_pp, 111

form/2
erl_pp, 111

format/1
io, 199
proc_lib, 250

format/2
io_lib, 209

format/3
io, 199

format_error/1
beam_lib, 55
erl_int, 107
erl_parse, 109
erl_scan, 115
erl_tar, 119
ms_transform, 241
qic, 265
regexp, 280
win32reg, 351

fwrite/1
io, 199

fwrite/2
io_lib, 209

fwrite/3
io, 199

gb_sets
add/2, 157
add_element/2, 157
balance/1, 157
del_element/2, 157
delete/2, 157
delete_any/2, 157
difference/2, 157
empty/0, 158
filter/2, 158
fold/3, 158
from_list/1, 158
from_ordset/1, 158
insert/2, 158
intersection/1, 159
intersection/2, 159
is_element/2, 159
is_empty/1, 159
is_member/2, 159
is_set/1, 159
is_subset/2, 159
iterator/1, 159
largest/1, 160
new/0, 158
next/1, 160
singleton/1, 160
size/1, 160
smallest/1, 160
subtract/2, 158
take_largest/1, 160
take_smallest/1, 160
to_list/1, 161
union/1, 161
union/2, 161

gb_trees
balance/1, 162
delete/2, 162
delete_any/2, 163
empty/0, 163
enter/3, 163
from_orddict/1, 163
get/2, 163
insert/3, 164
is_defined/2, 164
is_empty/1, 164
iterator/1, 164
keys/1, 164
largest/1, 164
lookup/2, 163
next/1, 164
size/1, 165
smallest/1, 165
take_largest/1, 165
take_smallest/1, 165
to_list/1, 165
update/3, 165
values/1, 166
gen_event
add_handler/3, 168
add_sup_handler/3, 169
call/3, 170
call/4, 170
delete_handler/3, 171
Module:code_change/3, 175
Module:handle_call/2, 174
Module:handle_event/2, 173
Module:handle_info/2, 174
Module:init/1, 173
Module:terminate/2, 175
notify/2, 170
start/0, 168
start/1, 168
start_link/0, 168
start_link/1, 168
stop/1, 173
swap_handler/5, 171
swap_sup_handler/5, 172
sync_notify/2, 170
which_handlers/1, 172
gen_fsm
cancel_timer/1, 181
enter_loop/4, 182
enter_loop/5, 182
enter_loop/6, 182
Module:code_change/4, 186
Module:handle_event/3, 184
Module:handle_info/3, 185
Module:handle_sync_event/4, 185
Module:init/1, 183
Module:StateName/2, 183
Module:StateName/3, 184
Module:StateName/4, 185
reply/2, 181
send_all_state_event/2, 179
send_event/2, 179
send_event_after/2, 181
start/3, 179
start/4, 179
start_link/3, 178
start_link/4, 178
start_timer/2, 181
sync_send_all_state_event/2, 180
sync_send_all_state_event/3, 180
sync_send_event/2, 180
sync_send_event/3, 180
gen_server
abcast/2, 192
abcast/3, 192
call/2, 190
call/3, 190
cast/2, 192
enter_loop/3, 192
enter_loop/4, 193
enter_loop/5, 193
Module:code_change/3, 196
Module:handle_call/3, 194
Module:handle_cast/2, 194
Module:handle_info/2, 195
Module:init/1, 193
Module:terminate/2, 195
multi_call/2, 191
multi_call/3, 191
multi_call/4, 191
reply/2, 192
start/3, 189
start/4, 189
start_link/3, 188
start_link/4, 188
get/2
gb_trees, 163
get_all_values/2
proplists, 254
get_bool/2
proplists, 254
get_chars/3
io, 197
get_cycle/2
digraph, 90
get_debug/3  sys, 342
get_keys/1  proplists, 254
get_line/2  io, 198
get_node/0  pool, 247
get_nodes/0  pool, 247
get_path/3  digraph, 90
get_short_cycle/2  digraph, 90
get_short_path/3  digraph, 90
get_status/1  sys, 340
get_status/2  sys, 340
get_value/2  proplists, 255
get_value/3  proplists, 255
gregorian_days_to_date/1  calendar, 62
gregorian_seconds_to_datetime/1  calendar, 62
gsub/3  regexp, 279
guard/1  erl_pp, 111
guard/2  erl_pp, 111
guard_bif/2  erl_internal, 104
handle_debug/1  sys, 342
handle_system_msg/6  sys, 342
head/1  queue, 272
help/0  c, 58
hibernate/3  proc_lib, 251
history/1  shell, 294
hms/3  timer, 347
hours/1  timer, 347
i/0  c, 58
ets, 126
i/1  ets, 126
i/3  c, 58
image/2  sofs, 311
in/2  queue, 272
in_degree/2  digraph, 91
in_edges/2  digraph, 91
in_neighbours/2  digraph, 91
in_r/2  queue, 273
indentation/2  io_lib, 210
info/1  beam_lib, 53
dets, 69
digraph, 91
ets, 126
info/2  dets, 70
ets, 127
qlc, 265
init/1  queue, 273
init/3  log_mf, 228
init/4
   log.mfh, 228

init_ack/1
   proc_lib, 250

init_ack/2
   proc_lib, 250

init_table/2
   ets, 127

init_table/3
   dets, 70

initial_call/1
   proc_lib, 251

insert/2
   dets, 71
   ets, 128
   gb_sets, 158

insert/3
   gb_trees, 164

insert_new/2
   dets, 71
   ets, 128

install/3
   sys, 341

install/4
   sys, 341

intersection/1
   gb_sets, 159
   sets, 284
   sofs, 311

intersection/2
   gb_sets, 159
   sets, 284
   sofs, 311

intersection_of_family/1
   sofs, 311

inverse/1
   sofs, 311

inverse_image/2
   sofs, 312

io
   format/1, 199
   format/3, 199
   fread/3, 203
   fwrite/1, 199
   fwrite/3, 199
   get_chars/3, 197
   get_line/2, 198
   nl/1, 197
   parse_erl_exprs/1, 205
   parse_erl_exprs/3, 205
   parse_erl_form/1, 206
   parse_erl_form/3, 206
   put_chars/2, 197
   read/2, 198
   read/3, 199
   scan_erl_exprs/1, 204
   scan_erl_exprs/3, 204
   scan_erl_form/1, 205
   scan_erl_form/3, 205
   setopts/2, 198
   write/2, 198

io_lib
   char_list/1, 211
   deep_char_list/1, 211
   format/2, 209
   fread/2, 209
   fread/3, 209
   fwrite/2, 209
   indentation/2, 210
   nl/0, 208
   print/1, 208
   print/4, 208
   printable_list/1, 211
   write/1, 208
   write/2, 208
   write_atom/1, 210
   write_char/1, 210
   write_string/1, 210

is_a_function/1
   sofs, 312

is_acyclic/1
   digraph_utils, 95

is_compatible_bchunk_format/2
   dets, 72

is_compiled_ms/1
   ets, 128

is_defined/2
   gb_trees, 164
   proplists, 255

is_dets_file/1
   dets, 72

is_dir/1
   filelib, 147

is_disjoint/2
ets, 129
lists, 217
queue, 273

last_day_of_the_month/2
   calendar, 62
last_modified/1
   filelib, 148

lc/1
   c, 58
left/2
   string, 325
left/3
   string, 325
len/1
   queue, 273
   string, 322
lib
   error_message/2, 212
   flush_receive/0, 212
   nonl/1, 212
   progranme/0, 212
   send/2, 212
   sendv/2, 213
list_op/2
   erl_internal, 105
lists
   all/2, 224
   any/2, 224
   append/1, 214
   append/2, 214
   concat/1, 214
   delete/2, 215
   dropwhile/2, 224
   duplicate/2, 215
   filter/2, 224
   flatlength/1, 215
   flatmap/2, 225
   flatten/1, 215
   flatten/2, 215
   foldl/3, 225
   foldr/3, 225
   foreach/2, 226
   keydelete/3, 215
   keymember/3, 216
   keymerge/3, 216
   keyreplace/4, 216
   keysearch/3, 216
   keysort/2, 216
   last/1, 217

map/2, 226
mapfoldl/3, 226
mapfoldr/3, 226
max/1, 217
member/2, 217
merge/1, 217
merge/2, 217
merge/3, 217
merge3/3, 218
min/1, 218
nth/2, 218
nthtail/2, 218
partition/2, 227
prefix/2, 218
reverse/1, 218
reverse/2, 219
seq/2, 219
seq/3, 219
sort/1, 219
sort/2, 219
split/2, 220
splitwith/2, 227
sublist/2, 220
sublist/3, 220
subtract/2, 220
suffix/2, 220
sum/1, 221
takewhile/2, 227
ukeymerge/3, 221
ukeysort/2, 221
umerge/1, 221
umerge/2, 221
umerge/3, 221
umerge3/3, 222
unzip/1, 222
unzip3/1, 222
usort/1, 222
usort/2, 222
zip/2, 223
zip3/3, 223
zipwith/3, 223
zipwith3/4, 223

local_time/0
   calendar, 63
local_time_to_universal_time/2
   calendar, 63
local_time_to_universal_time_dst/2
   calendar, 63
log/1
   math, 229

STDLIB 367
log/2
  sys, 339
log/3
  sys, 339
log10/1
  math, 229
log10/1
  math, 229
log1f
  init/3, 228
  init/4, 228
log_to_file/2
  sys, 339
log_to_file/3
  sys, 339
lookup/2
  dets, 72
  ets, 129
  gb_trees, 163
  proplists, 255
lookup_all/2
  proplists, 255
lookup_element/3
  ets, 129
loop_vertices/1
  digraph_utils, 95
ls/0
  c, 58
ls/1
  c, 58
m/0
  c, 59
m/1
  c, 59
map/2
  dict, 84
  lists, 226
mapfoldl/3
  lists, 226
mapfoldr/3
  lists, 226
match/1
  dets, 72
  ets, 131
match/2
  dets, 73
match/3
  dets, 73
  ets, 130
match_delete/2
  dets, 73
  ets, 131
match_object/1
  dets, 74
  ets, 131
match_object/2
  dets, 74
  ets, 131
match_object/3
  dets, 74
  ets, 131
match_spec_compile/1
  ets, 132
match_spec_run/2
  ets, 132
matches/2
  regexp, 278
math
  acos, 1, 229
  acosh, 1, 229
  asin, 1, 229
  asinh, 1, 229
  atan, 1, 229
  atanh, 1, 229
  cos, 1, 229
  cosh, 1, 229
  erf, 1, 230
  erfc, 1, 230
  exp, 1, 229
  log, 1, 229
  log10, 1, 229
  pi, 0, 229
  pow, 2, 229
  sin, 1, 229
  sinh, 1, 229
  sqrt, 1, 229
  tan, 1, 229
  tanh, 1, 229
max/1
  lists, 217
member/2
nc/1
  c, 59
nc/2
  c, 59
new/0
dict, 84
digraph, 92
gb_sets, 158
queue, 274
sets, 283
new/1
digraph, 92
new/2
dets, 133
new_bindings/0
erl_eval, 101
next/1
gb_sets, 160
gb_trees, 164
next/2
dets, 75
dets, 134
next_answers/2
qlc, 266
ni/0
c, 58
nl/0
io_lib, 208
nl/1
c, 59
io, 197
no_debug/1
sys, 340
no_debug/2
sys, 340
no_edges/1
digraph, 92
no_elements/1
sofs, 314
no_vertices/1
digraph, 92
nonl/1
lib, 212
normalise/1
erl_parse, 109
normalize/2
proplists, 255
notify/2
gen_event, 170
now_diff/2	imer, 347
now_to_datetime/1
calendar, 64
now_to_local_time/1
calendar, 63
now_to_universal_time/1
calendar, 64
nregs/0
c, 60
nth/2
lists, 218
nthtail/2
lists, 218
op_type/2
erl_internal, 105
open/1
win32reg, 351
open/2
epp, 98
erl_tar, 119
open/3
epp, 98
open_file/1
dets, 75
open_file/2
dets, 75
out/1
queue, 274
out_degree/2
digraph, 92
out_edges/2
digraph, 92
out_neighbours/2
digraph, 92
out_r/1
queue, 274

370

STDLIB
parse/1  
  regexp, 280
parse_erl_exprs/1  
  io, 205
parse_erl_exprs/3  
  io, 205
parse_erl_form/1  
  epp, 98 
  io, 206
parse_erl_form/3  
  io, 206
parse_exprs/1  
  erl_parse, 108
parse_file/3  
  epp, 98
parse_form/1  
  erl_parse, 108
parse_term/1  
  erl_parse, 108
parse_transform/2  
  erl_id_trans, 103  
  ms_transform, 240
partition/1  
  sofs, 314
partition/2  
  lists, 227  
  sofs, 314
partition/3  
  sofs, 314
partition_family/2  
  sofs, 315
pathtype/1  
  filename, 153
pg  
  create/1, 244  
  create/2, 244  
  esend/2, 245  
  join/2, 244  
  members/1, 245  
  send/2, 245
pi/o  
  math, 229
pid/3  
  c, 59
pid2name/1  
  dets, 77
pool  
  attach/1, 246  
  get_node/0, 247  
  get_nodes/0, 247  
  pspawn/3, 247  
  pspawn_link/3, 247  
  start/1, 246  
  start/2, 246  
  stop/0, 247
postorder/1  
  digraph_utils, 96
pow/2  
  math, 229
prefix/2  
  lists, 218
preorder/1  
  digraph_utils, 96
prev/2  
  ets, 134
print/1  
  io_lib, 208
print/4  
  io_lib, 208
print_log/1  
  sys, 343
printable_list/1  
  io_lib, 211
proc_lib  
  format/1, 250  
  hibernate/3, 251  
  init_ack/1, 250  
  init_ack/2, 250  
  initial_call/1, 251  
  spawn/1, 248  
  spawn/2, 248  
  spawn/3, 248  
  spawn/4, 248  
  spawn_link/1, 248  
  spawn_link/2, 248  
  spawn_link/3, 248  
  spawn_link/4, 248  
  spawn_opt/2, 249  
  spawn_opt/3, 249  
  spawn_opt/4, 249  
  spawn_opt/5, 249  
  start/3, 249
start/4, 249
start/5, 249
start_link/3, 249
start_link/4, 249
start_link/5, 249
translate_initial_call/1, 251

product/1
  sofs, 315

product/2
  sofs, 315

proname/0
  lib, 212

projection/2
  sofs, 316

property/1
  proplists, 256

property/2
  proplists, 256

proplists
  append_values/2, 253
  compact/1, 253
  delete/2, 253
  expand/2, 253
  get_all_values/2, 254
  get_bool/2, 254
  get_keys/1, 254
  get_value/2, 255
  get_value/3, 255
  is_defined/2, 255
  lookup/2, 255
  lookup_all/2, 255
  normalize/2, 255
  property/1, 256
  property/2, 256
  split/2, 256
  substitute_aliases/2, 257
  substitute_negations/2, 257
  unfold/1, 257

pseudo/1
  slave, 298

pseudo/2
  slave, 298

pspawn/3
  pool, 247

pspawn_link/3
  pool, 247

put_chars/2

io, 197

pwd/0
  c, 60

q/0
  c, 60

q/2
  q/c, 267

q/c
  append/1, 264
  append/2, 264
  cursor/2, 264
  delete_cursor/1, 264
  e/2, 264
  eval/2, 264
  fold/4, 265
  format_error/1, 265
  info/2, 265
  keysort/3, 266
  next_answers/2, 266
  q/2, 267
  sort/2, 268
  string_to_handle/3, 268
  table/2, 269

queue
  cons/2, 272
  dash/1, 272
  from_list/1, 272
  head/1, 272
  in/2, 272
  in_r/2, 273
  init/1, 273
  is_empty/1, 273
  join/2, 273
  lait/1, 273
  last/1, 273
  len/1, 273
  new/0, 274
  out/1, 274
  out_r/1, 274
  reverse/1, 274
  snoc/2, 274
  split/2, 274
  tail/1, 274
  to_list/1, 275

random
  seed/0, 276
  seed/3, 276
  seed0/0, 276
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Module</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan_erl_exprs/3</td>
<td>io , 204</td>
<td></td>
</tr>
<tr>
<td>scan_erl_form/1</td>
<td>io , 204</td>
<td></td>
</tr>
<tr>
<td>scan_erl_form/3</td>
<td>io , 204</td>
<td></td>
</tr>
<tr>
<td>seconds/1</td>
<td>timer , 347</td>
<td></td>
</tr>
<tr>
<td>seconds_to_daytime/1</td>
<td>calendar , 64</td>
<td></td>
</tr>
<tr>
<td>seconds_to_time/1</td>
<td>calendar , 64</td>
<td></td>
</tr>
<tr>
<td>seed/0</td>
<td>random , 276</td>
<td></td>
</tr>
<tr>
<td>seed/3</td>
<td>random , 276</td>
<td></td>
</tr>
<tr>
<td>seed0/0</td>
<td>random , 276</td>
<td></td>
</tr>
<tr>
<td>select/1</td>
<td>dets , 78</td>
<td></td>
</tr>
<tr>
<td>select/2</td>
<td>dets , 78</td>
<td></td>
</tr>
<tr>
<td>select/3</td>
<td>dets , 78</td>
<td></td>
</tr>
<tr>
<td>select_count/2</td>
<td>ets , 138</td>
<td></td>
</tr>
<tr>
<td>select_delete/2</td>
<td>ets , 138</td>
<td></td>
</tr>
<tr>
<td>send/2</td>
<td>lib , 212</td>
<td></td>
</tr>
<tr>
<td>send/2</td>
<td>pg , 245</td>
<td></td>
</tr>
<tr>
<td>send_after/2</td>
<td>timer , 345</td>
<td></td>
</tr>
<tr>
<td>send_after/3</td>
<td>timer , 345</td>
<td></td>
</tr>
<tr>
<td>send_all_state_event/2</td>
<td>gen fsm , 179</td>
<td></td>
</tr>
<tr>
<td>send_event/2</td>
<td>gen fsm , 179</td>
<td></td>
</tr>
<tr>
<td>send_event/2</td>
<td>gen fsm , 179</td>
<td></td>
</tr>
<tr>
<td>send_event_after/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gen fsm , 181</td>
<td></td>
<td></td>
</tr>
<tr>
<td>send_interval/2</td>
<td>timer , 346</td>
<td></td>
</tr>
<tr>
<td>send_interval/3</td>
<td>timer , 346</td>
<td></td>
</tr>
<tr>
<td>send_op/2</td>
<td>erl internal , 105</td>
<td></td>
</tr>
<tr>
<td>sendw/2</td>
<td>lib , 213</td>
<td></td>
</tr>
<tr>
<td>seq/2</td>
<td>lists , 219</td>
<td></td>
</tr>
<tr>
<td>seq/3</td>
<td>lists , 219</td>
<td></td>
</tr>
<tr>
<td>set/2</td>
<td>sofs , 318</td>
<td></td>
</tr>
<tr>
<td>set_value/3</td>
<td>win32reg , 351</td>
<td></td>
</tr>
<tr>
<td>setopts/2</td>
<td>io , 198</td>
<td></td>
</tr>
<tr>
<td>sets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add_element/2</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>del_element/2</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>filter/2</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>fold/3</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>from_list/1</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>intersection/1</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>intersection/2</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>is_element/2</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>is_set/1</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>is_subset/2</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>new/0</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>size/1</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>subtract/2</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>to_list/1</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>union/1</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>union/2</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>sh_to_awk/1</td>
<td>regexp , 280</td>
<td></td>
</tr>
<tr>
<td>shell</td>
<td></td>
<td></td>
</tr>
<tr>
<td>history/1</td>
<td>294</td>
<td></td>
</tr>
<tr>
<td>results/1</td>
<td>294</td>
<td></td>
</tr>
<tr>
<td>start_restricted/1</td>
<td>294</td>
<td></td>
</tr>
<tr>
<td>stop_restricted/0</td>
<td>294</td>
<td></td>
</tr>
<tr>
<td>sin/1</td>
<td>math , 229</td>
<td></td>
</tr>
<tr>
<td>singleton/1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
gb_sets, 160
sinh/1
  math, 229
size/1
  gb_sets, 160
gb_trees, 165
  sets, 283
slave
  pseudo/1, 298
  pseudo/2, 298
  relay/1, 298
  start/1, 296
  start/2, 296
  start/3, 296
  start_link/1, 297
  start_link/2, 297
  start_link/3, 297
  stop/1, 298
sleep
  timer, 346
slot/2
  dets, 79
gs, 139
smallest
  gb_sets, 160
  gb_trees, 165
snoc/2
  queue, 274
sofs
  a_function/2, 303
canonical_relation/1, 303
composite/2, 303
constant_function/2, 303
difference/1, 304
digraph_to_family/1, 304
domain/1, 304
drestriction/1, 304
drestriction/2, 305
drestriction/3, 305
empty_set/0, 305
extension/3, 305
family/2, 306
family_difference/2, 306
family_domain/1, 306
family_field/1, 306
family_intersection/1, 307
family_intersection/2, 307
family_projection/2, 307
family_range/1, 307
family_specification/2, 308
family_to_digraph/2, 308
family_to_relation/1, 308
family_union/1, 309
family_union/2, 309
field/1, 309
from_external/2, 309
from_sets/1, 309, 310
from_term/2, 310
image/2, 311
intersection/1, 311
intersection/2, 311
intersection_of_family/1, 311
inverse/1, 311
inverse_image/2, 312
is_a_function/1, 312
is_disjoint/2, 312
is_empty_set/1, 312
is_equal/2, 312
is_set/1, 312
is_sofs_set/1, 313
is_subset/2, 313
is_type/1, 313
join/4, 313
multiple_relative_product/2, 313
no_elements/1, 314
partition/1, 314
partition/2, 314
partition/3, 314
partition_family/2, 315
product/1, 315
product/2, 315
projection/2, 316
range/1, 316
relation/2, 316
relation_to_family/1, 316
relative_product/1, 317
relative_product/2, 317
restriction/2, 317
restriction/3, 318
relation/2, 316
set/2, 318
symdiff/2, 319
symmetric_partition/1, 318
to_external/1, 320
to_sets/1, 320
type/1, 320
union/1, 320
union/2, 320
union_of_family/1, 321
weak_relation/1, 321
lists, 220
substitute_aliases/2
  proplists, 257
substitute_negations/2
  proplists, 257
substitution/2
  sofs, 319
substr/2
  string, 323
substr/3
  string, 323
subtract/2
  gb_sets, 158
  lists, 220
  sets, 284
suffix/2
  lists, 220
sum/1
  lists, 221
supervisor
  check_childspecs/1, 333
  delete_child/2, 332
  Module:init/1, 334
  restart_child/2, 332
  start_child/2, 331
  start_link/2, 330
  start_link/3, 330
  terminate_child/2, 331
  which_children/1, 333
supervisor_bridge
  Module:init/1, 336
  Module:terminate/2, 336
  start_link/2, 335
  start_link/3, 335
suspend/1
  sys, 340
suspend/2
  sys, 340
swap_handler/5
  gen_event, 171
swap_sup_handler/5
  gen_event, 172
symdiff/2
  sofs, 319
symmetric_partition/2
sofs, 320
sync/1
  dets, 79
sync_notify/2
  gen_event, 170
sync_send_all_state_event/2
  gen_fsm, 180
sync_send_all_state_event/3
  gen_fsm, 180
sync_send_event/2
  gen_fsm, 180
sync_send_event/3
  gen_fsm, 180
sys
  change_code/4, 340
  change_code/5, 340
  debug_options/1, 342
  get_debug/3, 342
  get_status/1, 340
  get_status/2, 340
  handle_debug/1, 342
  handle_system_msg/6, 342
  install/3, 341
  install/4, 341
  log/2, 339
  log/3, 339
  log_to_file/2, 339
  log_to_file/3, 339
  Mod:system_code_change/4, 343
  Mod:system_continue/3, 343
  Mod:system_terminate/4, 343
  no_debug/1, 340
  no_debug/2, 340
  print_log/1, 343
  remove/2, 341
  remove/3, 341
  resume/1, 340
  resume/2, 340
  statistics/2, 339
  statistics/3, 339
  suspend/1, 340
  suspend/2, 340
  trace/2, 340
  trace/3, 340
t/1
  erl_tar, 120
tab2file/2
  ets, 139
tab2list/1
ets, 139

table/1
erl_tar, 120

table/2
dets, 79
erl_tar, 120
ets, 139
qlc, 269

tail/1
queue, 274

take_largest/1
gb_sets, 160
gb_trees, 165
take_smallest/1
gb_sets, 160
gb_trees, 165
takewhile/2
lists, 227
tan/1
math, 229
tanh/1
math, 229
tc/3
timer, 347
terminate_child/2
supervisor, 331
test_ms/2
ets, 140
time_difference/2
calendar, 64
time_to_seconds/1
calendar, 64
timer
apply_after/4, 345
apply_interval/4, 346
cancel/1, 346
exit_after/2, 346
exit_after/3, 346
hms/3, 347
hours/1, 347
kill_after/1, 346
kill_after/2, 346
minutes/1, 347
now_diff/2, 347
seconds/1, 347
send_after/2, 345
send_after/3, 345
send_interval/2, 346
send_interval/3, 346
sleep/1, 346
start/0, 345
tc/3, 347
to_dets/2
ets, 141
to_ets/2
dets, 80
to_external/1
sofs, 320
to_float/1
string, 326
to_integer/1
string, 326
to_list/1
dict, 84
gb_sets, 161
gb_trees, 165
queue, 275
sets, 283
to_sets/1
sofs, 320
tokens/1
erl_parse, 109
tokens/2
erl_parse, 109
string, 323
tokens/3
erl_scan, 114
topsort/1
digraph_utils, 97
trace/2
sys, 340
trace/3
sys, 340
transform_from_shell/3
ms_transform, 240
translate_initial_call/1
proc_lib, 251
traverse/2
dets, 81
tt/1

STDLIB 379
erl_tar, 120
type/1
  sofs, 320
type_test/2
  erl_internal, 104

ukeymerge/3
  lists, 221
ukeysort/2
  lists, 221
umerge/1
  lists, 221
umerge/2
  lists, 221
umerge/3
  lists, 221
umerge3/3
  lists, 222
unfold/1
  proplists, 257
uniform/0
  random, 276
uniform/1
  random, 276
uniform_s/1
  random, 277
uniform_s/2
  random, 277
union/1
  gb_sets, 161
    sets, 284
    sofs, 320
union/2
  gb_sets, 161
    sets, 284
    sofs, 320
union_of_family/1
  sofs, 321
universal_time/0
  calendar, 64
universal_time_to_local_time/2
  calendar, 65
unzip/1
  lists, 222
unzip3/1
  lists, 222
update/3
  dict, 85
    gb_trees, 165
update/4
  dict, 85
update_counter/3
  dets, 81
    dict, 85
    ets, 141
update_counter/4
  ets, 141
update_counter/6
  ets, 141
usort/1
  lists, 222
usort/2
  lists, 222
valid_date/1
  calendar, 65
valid_date/3
  calendar, 65
value/2
  win32reg, 352
values/1
  gb_trees, 166
    win32reg, 352
version/1
  beam_lib, 52
vertex/2
  digraph, 93
vertices/1
  digraph, 93
weak_relation/1
  sofs, 321
which_children/1
  supervisor, 333
which_handlers/1
  gen_event, 172
wildcard/1
  filelib, 148
wildcard/2
  filelib , 149

win32reg
  change_key/2, 350
  change_key_create/2, 350
  close/1, 350
  current_key/1, 350
  delete_key/1, 350
  delete_value/2, 351
  expand/1, 351
  format_error/1, 351
  open/1, 351
  set_value/3, 351
  sub_keys/1, 352
  value/2, 352
  values/1, 352

words/1
  string , 324

words/2
  string , 324

write/1
  io_lib , 208

write/2
  io , 198
  io_lib , 208

write_atom/1
  io_lib , 210

write_char/1
  io_lib , 210

write_string/1
  io_lib , 210

xm/1
  c , 60

zip/2
  lists , 223

zip3/3
  lists , 223

zipwith/3
  lists , 223

zipwith3/4
  lists , 223