STDLIB

version 1.14
Typeset in \( \LaTeX \) from SGML source using the DOCBUILDER 3.4.1 Document System.
## Contents

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

1.1 STD LIB ........................................... 52
1.2 base64 ............................................ 53
1.3 beam_lib ........................................... 54
1.4 c ............................................... 62
1.5 calendar ........................................... 66
1.6 dets .............................................. 71
1.7 dict ............................................. 87
1.8 digraph ............................................ 92
1.9 digraph_utils ..................................... 99
1.10 epp .............................................. 103
1.11 erl_eval ........................................ 105
1.12 erl_expand_records ................................ 108
1.13 erl_jd_trans ..................................... 109
1.14 erl_internal ..................................... 110
1.15 erl_int ......................................... 112
1.16 erl_parse ........................................ 114
1.17 erl_pp .......................................... 117
1.18 erl_scan ........................................ 120
1.19 erl_tar ......................................... 122
1.20 ets ............................................. 128
1.21 file_sorter ...................................... 148
1.22 filelib .......................................... 153
1.23 filename ........................................ 156
1.24 gb_sets ........................................ 162
1.25 gb_trees ........................................ 168
1.26 gen_event ....................................... 173
1.27 gen_fsm ......................................... 183
1.28 gen_server ..................................... 194
1.29 io ............................................... 203
<table>
<thead>
<tr>
<th>Section</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.30</td>
<td>io_lib</td>
<td>214</td>
</tr>
<tr>
<td>1.31</td>
<td>lib</td>
<td>218</td>
</tr>
<tr>
<td>1.32</td>
<td>lists</td>
<td>220</td>
</tr>
<tr>
<td>1.33</td>
<td>log_mf_h</td>
<td>235</td>
</tr>
<tr>
<td>1.34</td>
<td>math</td>
<td>236</td>
</tr>
<tr>
<td>1.35</td>
<td>ms_transform</td>
<td>238</td>
</tr>
<tr>
<td>1.36</td>
<td>orddict</td>
<td>249</td>
</tr>
<tr>
<td>1.37</td>
<td>ordsets</td>
<td>250</td>
</tr>
<tr>
<td>1.38</td>
<td>pg</td>
<td>251</td>
</tr>
<tr>
<td>1.39</td>
<td>pool</td>
<td>253</td>
</tr>
<tr>
<td>1.40</td>
<td>proc_lib</td>
<td>255</td>
</tr>
<tr>
<td>1.41</td>
<td>proplists</td>
<td>260</td>
</tr>
<tr>
<td>1.42</td>
<td>qlc</td>
<td>265</td>
</tr>
<tr>
<td>1.43</td>
<td>queue</td>
<td>282</td>
</tr>
<tr>
<td>1.44</td>
<td>random</td>
<td>286</td>
</tr>
<tr>
<td>1.45</td>
<td>regexp</td>
<td>288</td>
</tr>
<tr>
<td>1.46</td>
<td>sets</td>
<td>293</td>
</tr>
<tr>
<td>1.47</td>
<td>shell</td>
<td>296</td>
</tr>
<tr>
<td>1.48</td>
<td>shell_default</td>
<td>305</td>
</tr>
<tr>
<td>1.49</td>
<td>slave</td>
<td>306</td>
</tr>
<tr>
<td>1.50</td>
<td>sofs</td>
<td>309</td>
</tr>
<tr>
<td>1.51</td>
<td>string</td>
<td>332</td>
</tr>
<tr>
<td>1.52</td>
<td>supervisor</td>
<td>338</td>
</tr>
<tr>
<td>1.53</td>
<td>supervisor_bridge</td>
<td>345</td>
</tr>
<tr>
<td>1.54</td>
<td>sys</td>
<td>348</td>
</tr>
<tr>
<td>1.55</td>
<td>timer</td>
<td>355</td>
</tr>
<tr>
<td>1.56</td>
<td>win32reg</td>
<td>359</td>
</tr>
<tr>
<td>1.57</td>
<td>zip</td>
<td>363</td>
</tr>
</tbody>
</table>
STDLIB Reference Manual

Short Summaries

- Application **STDLIB** [page 52] - The STDLIB Application
- Erlang Module **base64** [page 53] - Implements base 64 encode and decode, see RFC2045.
- Erlang Module **beam_lib** [page 54] - An Interface To the BEAM File Format
- Erlang Module **c** [page 62] - Command Interface Module
- Erlang Module **calendar** [page 66] - Local and universal time, day-of-the-week, date and time conversions
- Erlang Module **dets** [page 71] - A Disk Based Term Storage
- Erlang Module **dict** [page 87] - Key-Value Dictionary
- Erlang Module **digraph** [page 92] - Directed Graphs
- Erlang Module **digraph_utils** [page 99] - Algorithms for Directed Graphs
- Erlang Module **epp** [page 103] - An Erlang Code Preprocessor
- Erlang Module **erl_eval** [page 105] - The Erlang Meta Interpreter
- Erlang Module **erl_expand_records** [page 108] - Expands Records in a Module
- Erlang Module **erl_id_trans** [page 109] - An Identity Parse Transform
- Erlang Module **erl_internal** [page 110] - Internal Erlang Definitions
- Erlang Module **erl_lint** [page 112] - The Erlang Code Linter
- Erlang Module **erl_parse** [page 114] - The Erlang Parser
- Erlang Module **erl_pp** [page 117] - The Erlang Pretty Printer
- Erlang Module **erl_scan** [page 120] - The Erlang Token Scanner
- Erlang Module **erl_tar** [page 122] - Unix ‘tar’ utility for reading and writing tar archives
- Erlang Module **ets** [page 128] - Built-In Term Storage
- Erlang Module **file_sorter** [page 148] - File Sorter
- Erlang Module **filelib** [page 153] - File utilities, such as wildcard matching of filenames
- Erlang Module **filename** [page 156] - Filename Manipulation Functions
- Erlang Module **gb_sets** [page 162] - General Balanced Trees
- Erlang Module **gb_trees** [page 168] - General Balanced Trees
- Erlang Module **gen_event** [page 173] - Generic Event Handling Behaviour
- Erlang Module **gen_fsm** [page 183] - Generic Finite State Machine Behaviour
STDLIB Reference Manual

- Erlang Module **gen_server** [page 194] – Generic Server Behaviour
- Erlang Module **io** [page 203] – Standard IO Server Interface Functions
- Erlang Module **io_lib** [page 214] – IO Library Functions
- Erlang Module **lib** [page 218] – A number of useful library functions
- Erlang Module **lists** [page 220] – List Processing Functions
- Erlang Module **log.mf_h** [page 235] – An Event Handler which Logs Events to Disk
- Erlang Module **math** [page 236] – Mathematical Functions
- Erlang Module **ms_transform** [page 238] – Parse transform that translates fun syntax into match specifications.
- Erlang Module **orddict** [page 249] – Key-Value Dictionary as Ordered List
- Erlang Module **ordsets** [page 250] – Functions for Manipulating Sets as Ordered Lists
- Erlang Module **pg** [page 251] – Distributed, Named Process Groups
- Erlang Module **pool** [page 253] – Load Distribution Facility
- Erlang Module **proc_lib** [page 255] – Functions for asynchronous and synchronous start of processes adhering to the OTP design principles.
- Erlang Module **proplists** [page 260] – Support functions for property lists
- Erlang Module **qlc** [page 265] – Query Interface to Mnesia, ETS, Dets, etc
- Erlang Module **queue** [page 282] – Abstract Data Type for FIFO Queues
- Erlang Module **random** [page 286] – Pseudo random number generation
- Erlang Module **regexp** [page 288] – Regular Expression Functions for Strings
- Erlang Module **sets** [page 293] – Functions for Set Manipulation
- Erlang Module **shell** [page 296] – The Erlang Shell
- Erlang Module **shell_default** [page 305] – Customizing the Erlang Environment
- Erlang Module **slave** [page 306] – Functions to Starting and Controlling Slave Nodes
- Erlang Module **sofs** [page 309] – Functions for Manipulating Sets of Sets
- Erlang Module **string** [page 332] – String Processing Functions
- Erlang Module **supervisor** [page 338] – Generic Supervisor Behaviour
- Erlang Module **supervisor_bridge** [page 345] – Generic Supervisor Bridge Behaviour.
- Erlang Module **sys** [page 348] – A Functional Interface to System Messages
- Erlang Module **timer** [page 355] – Timer Functions
- Erlang Module **win32reg** [page 359] – win32reg provides access to the registry on Windows
- Erlang Module **zip** [page 363] – Utility for reading and creating ‘zip’ archives.

STDLIB

No functions are exported.
base64

The following functions are exported:

- `encode(Data) -> Base64`
  [Page 53] Encodes data into base64.
- `encode_to_string(Data) -> Base64String`
  [Page 53] Encodes data into base64.
- `decode(Base64) -> Data`
  [Page 53] Decodes a base64 encoded string to data.
- `decode_to_string(Base64) -> DataString`
  [Page 53] Decodes a base64 encoded string to data.
- `mime_decode(Base64) -> Data`
  [Page 53] Decodes a base64 encoded string to data.
- `mime_decode_string(Base64) -> DataString`
  [Page 53] Decodes a base64 encoded string to data.

beam_lib

The following functions are exported:

- `chunks(Beam, [ChunkRef]) -> ok, {Module, [ChunkData]} | {error, beam_lib, Reason}`
  [Page 57] Read selected chunks from a BEAM file or binary
- `chunks(Beam, [ChunkRef], [Option]) -> ok, {Module, [ChunkResult]} | {error, beam_lib, Reason}`
  [Page 57] Read selected chunks from a BEAM file or binary
- `version(Beam) -> ok, {Module, [Version]} | {error, beam_lib, Reason}`
  [Page 57] Read the BEAM file's module version
- `md5(Beam) -> ok, {Module, MD5} | {error, beam_lib, Reason}`
  [Page 58] Read the BEAM file's module version
- `info(Beam) -> [{Item, Info}] | {error, beam_lib, Reason1}`
  [Page 58] Information about a BEAM file
- `cmp(Beam1, Beam2) -> ok | {error, beam_lib, Reason}`
  [Page 59] Compare two BEAM files
- `cmp_dirs(Dir1, Dir2) -> Only1, Only2, Different | {error, beam_lib, Reason1}`
  [Page 59] Compare the BEAM files in two directories
- `diff_dirs(Dir1, Dir2) -> ok | {error, beam_lib, Reason1}`
  [Page 59] Compare the BEAM files in two directories
- `strip(Beam1) -> ok, {Module, Beam2} | {error, beam_lib, Reason1}`
  [Page 59] Removes chunks not needed by the loader from a BEAM file
- `strip_files(Files) -> ok, [[Module, Beam2]] | {error, beam_lib, Reason1}`
  [Page 59] Removes chunks not needed by the loader from BEAM files
- `strip_release(Dir) -> {ok, [[Module, Filename]]} | {error, beam_lib, Reason1}`
  [page 60] Removes chunks not needed by the loader from all BEAM files of a release
- `format_error(Reason) -> Chars`
  [page 60] Return an English description of a BEAM read error reply
- `crypto_key_fun(CryptoKeyFun) -> ok | {error, Reason}`
  [page 60] Register a fun that provides a crypto key
- `clear_crypto_key_fun() -> {ok, Result}`
  [page 61] Unregister the current crypto key fun

C

The following functions are exported:

- `bt(Pid) -> void()
  [page 62] Stack backtrace for a process`
- `c(File) -> {ok, Module} | error`
  [page 62] Compile and load code in a file
- `c(File, Options) -> {ok, Module} | error`
  [page 62] Compile and load code in a file
- `cd(Dir) -> void()`
  [page 62] Change working directory
- `flush() -> void()`
  [page 63] Flush any messages sent to the shell
- `help() -> void()`
  [page 63] Help information
- `i() -> void()`
  [page 63] Information about the system
- `ni() -> void()`
  [page 63] Information about the system
- `i(X, Y, Z) -> void()`
  [page 63] Information about pid <X.Y.Z>
- `l(Module) -> void()`
  [page 63] Load or reload module
- `lc(Files) -> ok`
  [page 63] Compile a list of files
- `ls() -> void()`
  [page 63] List files in the current directory
- `ls(Dir) -> void()`
  [page 63] List files in a directory
- `m() -> void()`
  [page 64] Which modules are loaded
- `m(Module) -> void()`
  [page 64] Information about a module
- `memory() -> [{Type, Size}]`
  [page 64] Memory allocation information
• memory(Type) -> Size
  [page 64] Memory allocation information
• memory([Type]) -> [[Type, Size]]
  [page 64] Memory allocation information
• nc(File) -> {ok, Module} | error
  [page 64] Compile and load code in a file on all nodes
• nc(File, Options) -> {ok, Module} | error
  [page 64] Compile and load code in a file on all nodes
• nl(Module) -> void()
  [page 64] Load module on all nodes
• pid(X, Y, Z) -> pid()
  [page 64] Convert X,Y,Z to a pid
• pwd() -> void()
  [page 65] Print working directory
• q() -> void()
  [page 65] Quit - shorthand for init:stop()
• regs() -> void()
  [page 65] Information about registered processes
• nregs() -> void()
  [page 65] Information about registered processes
• xm(ModSpec) -> void()
  [page 65] Cross reference check a module
• y(File) -> YeccRet
  [page 65] Generate an LALR-1 parser
• y(File, Options) -> YeccRet
  [page 65] Generate an LALR-1 parser

calendar

The following functions are exported:

• date_to_gregorian_days(Date) -> Days
  [page 67] Compute the number of days from year 0 up to the given date
• date_to_gregorian_days(Year, Month, Day) -> Days
  [page 67] Compute the number of days from year 0 up to the given date
• datetime_to_gregorian_seconds([Date, Time]) -> Seconds
  [page 67] Compute the number of seconds from year 0 up to the given date and time
• day_of_the_week(Date) -> DayNumber
  [page 67] Compute the day of the week
• day_of_the_week(Year, Month, Day) -> DayNumber
  [page 67] Compute the day of the week
• gregorian_days_to_date(Days) -> Date
  [page 67] Compute the date given the number of gregorian days
• gregorian_seconds_to_datetime(Seconds) -> {Date, Time}
  [page 67] Compute the date given the number of gregorian days
STDLIB Reference Manual

- is_leap_year(Year) -> bool()
  [page 67] Check if a year is a leap year
- last_day_of_the_month(Year, Month) -> int()
  [page 67] Compute the number of days in a month
- local_time() -> {Date, Time}
  [page 68] Compute local time
- local_time_to_universal_time([Date1, Time1]) -> {Date2, Time2}
  [page 68] Convert from local time to universal time (deprecated)
- local_time_to_universal_time_dst([Date1, Time1]) -> [{Date, Time}]
  [page 68] Convert from local time to universal time(s)
- now_to_local_time(Now) -> {Date, Time}
  [page 68] Convert now to local date and time
- now_to_universal_time(Now) -> {Date, Time}
  [page 69] Convert now to date and time
- now_to_datetime(Now) -> {Date, Time}
  [page 69] Convert now to date and time
- seconds_to_daytime(Seconds) -> {Days, Time}
  [page 69] Compute days and time from seconds
- seconds_to_time(Seconds) -> Time
  [page 69] Compute time from seconds
- time_difference(T1, T2) -> {Days, Time}
  [page 69] Compute the difference between two times (deprecated)
- time_to_seconds(Time) -> Seconds
  [page 69] Compute the number of seconds since midnight up to the given time
- universal_time() -> {Date, Time}
  [page 69] Compute universal time
- universal_time_to_local_time([Date1, Time1]) -> {Date2, Time2}
  [page 70] Convert from universal time to local time
- valid_date(Date) -> bool()
  [page 70] Check if a date is valid
- valid_date(Year, Month, Day) -> bool()
  [page 70] Check if a date is valid

**dets**

The following functions are exported:

- all() -> [Name]
  [page 72] Return a list of the names of all open Dets tables on this node.
- bchunk(Name, Continuation) -> {Continuation2, Data} | "$end_of_table" | {error, Reason}
  [page 72] Return a chunk of objects stored in a Dets table.
- close(Name) -> ok | {error, Reason}
  [page 73] Close a Dets table.
- delete(Name, Key) -> ok | {error, Reason}
  [page 73] Delete all objects with a given key from a Dets table.
- delete_all_objects(Name) -> ok | {error, Reason}  
  [page 73] Delete all objects from a Dets table.
- delete_object(Name, Object) -> ok | {error, Reason}  
  [page 73] Delete a given object from a Dets table.
- first(Name) -> Key | '$end_of_table'  
  [page 73] Return the first key stored in a Dets table.
- foldl(Function, Acc0, Name) -> Acc1 | {error, Reason}  
  [page 74] Fold a function over a Dets table.
- foldr(Function, Acc0, Name) -> Acc1 | {error, Reason}  
  [page 74] Fold a function over a Dets table.
- from_ets(Name, EtsTab) -> ok | {error, Reason}  
  [page 74] Replace the objects of a Dets table with the objects of an Ets table.
- info(Name) -> InfoList | undefined  
  [page 74] Return information about a Dets table.
- info(Name, Item) -> Value | undefined  
  [page 75] Return the information associated with a given item for a Dets table.
- init_table(Name, InitFun [, Options]) -> ok | {error, Reason}  
  [page 75] Replace all objects of a Dets table.
- insert(Name, Objects) -> ok | {error, Reason}  
  [page 76] Insert one or more objects into a Dets table.
- insert_new(Name, Objects) -> Bool  
  [page 76] Insert one or more objects into a Dets table.
- is_compatible_bchunk_format(Name, BchunkFormat) -> Bool  
  [page 77] Test compatibility of a table's chunk data.
- is_dets_file(FileName) -> Bool | {error, Reason}  
  [page 77] Test for a Dets table.
- lookup(Name, Key) -> [Object] | {error, Reason}  
  [page 77] Return all objects with a given key stored in a Dets table.
- match(Continuation) -> {{Match}, Continuation2} | '$end_of_table' |  
  {error, Reason}  
  [page 77] Match a chunk of objects stored in a Dets table and return a list of  
  variable bindings.
- match(Name, Pattern) -> {Match} | {error, Reason}  
  [page 78] Match the objects stored in a Dets table and return a list of variable  
  bindings.
- match(Name, Pattern, N) -> {{Match}, Continuation} | '$end_of_table' |  
  {error, Reason}  
  [page 78] 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 78] Delete all objects that match a given pattern from a Dets table.
- match_object(Continuation) -> {{Object}, Continuation2} |  
  '$end_of_table' | {error, Reason}  
  [page 79] Match a chunk of objects stored in a Dets table and return a list of  
  objects.
- match_object(Name, Pattern) -> [Object] | {error, Reason}  
  [page 79] Match the objects stored in a Dets table and return a list of objects.
The following functions are exported:

- `match_object(Name, Pattern, N) -> {[Object], Continuation} | '$end_of_table' | {error, Reason}`
  [page 79] Match the first chunk of objects stored in a Det's table and return a list of objects.
- `member(Name, Key) -> Bool | {error, Reason}`
  [page 80] Test for occurrence of a key in a Det's table.
- `next(Name, Key1) -> Key2 | '$end_of_table'`
  [page 80] Return the next key in a Det's table.
- `open_file(Filename) -> {ok, Reference} | {error, Reason}`
  [page 80] Open an existing Det's table.
- `open_file(Name, Args) -> {ok, Name} | {error, Reason}`
  [page 80] Open a Det's table.
- `pid2name(Pid) -> {ok, Name} | undefined`
  [page 82] Return the name of the Det's table handled by a pid.
- `repair_continuation(Continuation, MatchSpec) -> Continuation2`
  [page 82] Repair a continuation from select/1 or select/3.
- `safe_fixtable(Name, Fix)`
  [page 82] Fix a Det's table for safe traversal.
- `select(Continuation) -> {Selection, Continuation2} | '$end_of_table' | {error, Reason}`
  [page 83] Apply a match specification to some objects stored in a Det's table.
- `select(Name, MatchSpec) -> Selection | {error, Reason}`
  [page 83] Apply a match specification to all objects stored in a Det's table.
- `select(Name, MatchSpec, N) -> {Selection, Continuation} | '$end_of_table' | {error, Reason}`
  [page 83] Apply a match specification to the first chunk of objects stored in a Det's table.
- `select_delete(Name, MatchSpec) -> N | {error, Reason}`
  [page 84] Delete all objects that match a given pattern from a Det's table.
- `slot(Name, I) -> '$end_of_table' | [Object] | {error, Reason}`
  [page 84] Return the list of objects associated with a slot of a Det's table.
- `sync(Name) -> ok | {error, Reason}`
  [page 84] Ensure that all updates made to a Det's table are written to disk.
- `table(Name [, Options]) -> QueryHandle`
  [page 85] Return a QLC query handle.
- `to_ets(Name, EtsTab) -> EtsTab | {error, Reason}`
  [page 85] Insert all objects of a Det's table into an Ets table.
- `traverse(Name, Fun) -> Return | {error, Reason}`
  [page 86] Apply a function to all or some objects stored in a Det's table.
- `update_counter(Name, Key, Increment) -> Result`
  [page 86] Update a counter object stored in a Det's table.

```
(dict)
```

The following functions are exported:

- `append(Key, Value, Dict1) -> Dict2`
  [page 87] Append a value to keys in a dictionary
- `append_list(Key, ValList, Dict1) -> Dict2`  
  [page 87] Append new values to keys in a dictionary

- `erase(Key, Dict1) -> Dict2`  
  [page 87] Erase a key from a dictionary

- `fetch(Key, Dict) -> Value`  
  [page 87] Look-up values in a dictionary

- `fetch_keys(Dict) -> Keys`  
  [page 88] Return all keys in a dictionary

- `filter(Pred, Dict1) -> Dict2`  
  [page 88] Choose elements which satisfy a predicate

- `find(Key, Dict) -> {ok, Value} | error`  
  [page 88] Search for a key in a dictionary

- `fold(Fun, Acc0, Dict) -> Acc1`  
  [page 88] Fold a function over a dictionary

- `from_list(List) -> Dict`  
  [page 88] Convert a list of pairs to a dictionary

- `is_key(Key, Dict) -> bool()`  
  [page 88] Test if a key is in a dictionary

- `map(Fun, Dict1) -> Dict2`  
  [page 89] Map a function over a dictionary

- `merge(Fun, Dict1, Dict2) -> Dict3`  
  [page 89] Merge two dictionaries

- `new() -> dictionary()`  
  [page 89] Create a dictionary

- `store(Key, Value, Dict1) -> Dict2`  
  [page 89] Store a value in a dictionary

- `to_list(Dict) -> List`  
  [page 89] Convert a dictionary to a list of pairs

- `update(Key, Fun, Dict1) -> Dict2`  
  [page 90] Update a value in a dictionary

- `update(Key, Fun, Initial, Dict1) -> Dict2`  
  [page 90] Update a value in a dictionary

- `update_counter(Key, Increment, Dict1) -> Dict2`  
  [page 90] Increment a value in a dictionary

### digraph

The following functions are exported:

- `add_edge(G, E, V1, V2, Label) -> edge() | {error, Reason}`  
  [page 92] Add an edge to a digraph.

- `add_edge(G, V1, V2, Label) -> edge() | {error, Reason}`  
  [page 92] Add an edge to a digraph.

- `add_edge(G, V1, V2) -> edge() | {error, Reason}`  
  [page 92] Add an edge to a digraph.

- `add_vertex(G, V, Label) -> vertex()`  
  [page 93] Add or modify a vertex of a digraph.
• add_vertex(G, V) -> vertex()
  [page 93] Add or modify a vertex of a digraph.
• add_vertex(G) -> vertex()
  [page 93] Add or modify a vertex of a digraph.
• del_edge(G, E) -> true
  [page 93] Delete an edge from a digraph.
• del_edges(G, Edges) -> true
  [page 93] Delete edges from a digraph.
• del_path(G, V1, V2) -> true
  [page 93] Delete paths from a digraph.
• del_vertex(G, V) -> true
  [page 94] Delete a vertex from a digraph.
• del_vertices(G, Vertices) -> true
  [page 94] Delete vertices from a digraph.
• delete(G) -> true
  [page 94] Delete a digraph.
• edge(G, E) -> {E, V1, V2, Label} | false
  [page 94] Return the vertices and the label of an edge of a digraph.
• edges(G) -> Edges
  [page 94] Return all edges of a digraph.
• edges(G, V) -> Edges
  [page 94] Return the edges emanating from or incident on a vertex of a digraph.
• get_cycle(G, V) -> Vertices | false
  [page 95] Find one cycle in a digraph.
• get_path(G, V1, V2) -> Vertices | false
  [page 95] Find one path in a digraph.
• get_short_cycle(G, V) -> Vertices | false
  [page 95] Find one short cycle in a digraph.
• get_short_path(G, V1, V2) -> Vertices | false
  [page 95] Find one short path in a digraph.
• in_degree(G, V) -> integer()
  [page 96] Return the in-degree of a vertex of a digraph.
• in_edges(G, V) -> Edges
  [page 96] Return all edges incident on a vertex of a digraph.
• in_neighbours(G, V) -> Vertices
  [page 96] Return all in-neighbours of a vertex of a digraph.
• info(G) -> InfoList
  [page 96] Return information about a digraph.
• new() -> digraph()
  [page 97] Return a protected empty digraph, where cycles are allowed.
• new(Type) -> digraph() | {error, Reason}
  [page 97] Create a new empty digraph.
• no_edges(G) -> integer() >= 0
  [page 97] Return the number of edges of the a digraph.
• no_vertices(G) -> integer() >= 0
  [page 97] Return the number of vertices of a digraph.
• **out_degree(G, V) -> integer()**  
  [page 97] Return the out-degree of a vertex of a digraph.

• **out_edges(G, V) -> Edges**  
  [page 97] Return all edges emanating from a vertex of a digraph.

• **out_neighbours(G, V) -> Vertices**  
  [page 97] Return all out-neighbours of a vertex of a digraph.

• **vertex(G, V) -> \{V, Label\} | false**  
  [page 98] Return the label of a vertex of a digraph.

• **vertices(G) -> Vertices**  
  [page 98] Return all vertices of a digraph.

### digraph_utils

The following functions are exported:

• **components(Digraph) -> [Component]**  
  [page 100] Return the components of a digraph.

• **condensation(Digraph) -> CondensedDigraph**  
  [page 100] Return a condensed graph of a digraph.

• **cyclic_strong_components(Digraph) -> [StrongComponent]**  
  [page 100] Return the cyclic strong components of a digraph.

• **is_acyclic(Digraph) -> bool()**  
  [page 100] Check if a digraph is acyclic.

• **loop_vertices(Digraph) -> Vertices**  
  [page 100] Return the vertices of a digraph included in some loop.

• **postorder(Digraph) -> Vertices**  
  [page 101] Return the vertices of a digraph in post-order.

• **preorder(Digraph) -> Vertices**  
  [page 101] Return the vertices of a digraph in pre-order.

• **reachable(Vertices, Digraph) -> Vertices**  
  [page 101] Return the vertices reachable from some vertices of a digraph.

• **reachable_neighbours(Vertices, Digraph) -> Vertices**  
  [page 101] Return the neighbours reachable from some vertices of a digraph.

• **reaching(Vertices, Digraph) -> Vertices**  
  [page 101] Return the vertices that reach some vertices of a digraph.

• **reaching_neighbours(Vertices, Digraph) -> Vertices**  
  [page 101] Return the neighbours that reach some vertices of a digraph.

• **strong_components(Digraph) -> [StrongComponent]**  
  [page 102] Return the strong components of a digraph.

• **subgraph(Digraph, Vertices [, Options]) -> Subgraph | \{error, Reason\**  
  [page 102] Return a subgraph of a digraph.

• **topsort(Digraph) -> Vertices | false**  
  [page 102] Return a topological sorting of the vertices of a digraph.
epp

The following functions are exported:

- \texttt{open(FileName, IncludePath)} \rightarrow \{ok, Epp\} \mid \{error, ErrorDescriptor\}
  
  \hspace{1cm}[\text{page 103}] Open a file for preprocessing

- \texttt{open(FileName, IncludePath, PredefMacros)} \rightarrow \{ok, Epp\} \mid \{error, ErrorDescriptor\}
  
  \hspace{1cm}[\text{page 103}] Open a file for preprocessing

- \texttt{close(Epp)} \rightarrow \texttt{ok}
  
  \hspace{1cm}[\text{page 103}] Close the preprocessing of the file associated with Epp

- \texttt{parse_erl_form(Epp)} \rightarrow \{ok, AbsForm\} \mid \{eof, Line\} \mid \{error, ErrorInfo\}
  
  \hspace{1cm}[\text{page 103}] Return the next Erlang form from the opened Erlang source file

- \texttt{parse_file(FileName, IncludePath, PredefMacro)} \rightarrow \{ok, \{Form\}\} \mid \{error, OpenError\}
  
  \hspace{1cm}[\text{page 103}] Preprocess and parse an Erlang source file

erl_eval

The following functions are exported:

- \texttt{exprs(Expressions, Bindings)} \rightarrow \{value, Value, NewBindings\}
  
  \hspace{1cm}[\text{page 105}] Evaluate expressions

- \texttt{exprs(Expressions, Bindings, LocalFunctionHandler)} \rightarrow \{value, Value, NewBindings\}
  
  \hspace{1cm}[\text{page 105}] Evaluate expressions

- \texttt{exprs(Expressions, Bindings, LocalFunctionHandler, NonlocalFunctionHandler)} \rightarrow \{value, Value, NewBindings\}
  
  \hspace{1cm}[\text{page 105}] Evaluate expressions

- \texttt{expr(Expression, Bindings)} \rightarrow \{value, Value, NewBindings\}
  
  \hspace{1cm}[\text{page 105}] Evaluate expression

- \texttt{expr(Expression, Bindings, LocalFunctionHandler)} \rightarrow \{value, Value, NewBindings\}
  
  \hspace{1cm}[\text{page 105}] Evaluate expression

- \texttt{expr(Expression, Bindings, LocalFunctionHandler, NonlocalFunctionHandler)} \rightarrow \{value, Value, NewBindings\}
  
  \hspace{1cm}[\text{page 105}] Evaluate expression

- \texttt{expr_list(ExpressionList, Bindings)} \rightarrow \{ValueList, NewBindings\}
  
  \hspace{1cm}[\text{page 106}] Evaluate a list of expressions

- \texttt{expr_list(ExpressionList, Bindings, LocalFunctionHandler)} \rightarrow \{ValueList, NewBindings\}
  
  \hspace{1cm}[\text{page 106}] Evaluate a list of expressions

- \texttt{expr_list(ExpressionList, Bindings, LocalFunctionHandler, NonlocalFunctionHandler)} \rightarrow \{ValueList, NewBindings\}
  
  \hspace{1cm}[\text{page 106}] Evaluate a list of expressions

- \texttt{new_bindings()} \rightarrow \texttt{BindingStruct}
  
  \hspace{1cm}[\text{page 106}] Return a bindings structure
- bindings(BindingStruct) -> Bindings [page 106] Return bindings
- binding(Name, BindingStruct) -> Binding [page 106] Return bindings
- add_binding(Name, Value, Bindings) -> BindingStruct [page 106] Add a binding
- del_binding(Name, Bindings) -> BindingStruct [page 106] Delete a binding

erl_expand_records

The following functions are exported:
- module(AbsForms, CompileOptions) -> AbsForms [page 108] Expand all records in a module

erl_id_trans

The following functions are exported:
- parse_transform(Forms, Options) -> Forms [page 109] Transform Erlang forms

erl_internal

The following functions are exported:
- bif(Name, Arity) -> bool() [page 110] Test for an Erlang BIF
- guard_bif(Name, Arity) -> bool() [page 110] Test for an Erlang BIF allowed in guards
- type_test(Name, Arity) -> bool() [page 110] Test for a valid type test
- arith_op(OpName, Arity) -> bool() [page 110] Test for an arithmetic operator
- bool_op(OpName, Arity) -> bool() [page 110] Test for a Boolean operator
- comp_op(OpName, Arity) -> bool() [page 111] Test for a comparison operator
- list_op(OpName, Arity) -> bool() [page 111] Test for a list operator
- send_op(OpName, Arity) -> bool() [page 111] Test for a send operator
- op_type(OpName, Arity) -> Type [page 111] Return operator type
erl_lint

The following functions are exported:

- `module(AbsForms) -> {ok,Warnings} | {error,Errors,Warnings}`
  [page 112] Check a module for errors
- `module(AbsForms, FileName) -> {ok,Warnings} | {error,Errors,Warnings}`
  [page 112] Check a module for errors
- `module(AbsForms, FileName, CompileOptions) -> {ok,Warnings} | {error,Errors,Warnings}`
  [page 112] Check a module for errors
- `is_guard_test(Expr) -> bool()`
  [page 113] Test for a guard test
- `format_error(ErrorDescriptor) -> Chars`
  [page 113] Format an error descriptor

erl_parse

The following functions are exported:

- `parse_form(Tokens) -> {ok, AbsForm} | {error, ErrorInfo}`
  [page 114] Parse an Erlang form
- `parse_exprs(Tokens) -> {ok, Expr_list} | {error, ErrorInfo}`
  [page 114] Parse Erlang expressions
- `parse_term(Tokens) -> {ok, Term} | {error, ErrorInfo}`
  [page 114] Parse an Erlang term
- `format_error(ErrorDescriptor) -> Chars`
  [page 115] Format an error descriptor
- `tokens(AbsTerm) -> Tokens`
  [page 115] Generate a list of tokens for an expression
- `tokens(AbsTerm, MoreTokens) -> Tokens`
  [page 115] Generate a list of tokens for an expression
- `normalise(AbsTerm) -> Data`
  [page 115] Convert abstract form to an Erlang term
- `abstract(Data) -> AbsTerm`
  [page 115] Convert an Erlang term into an abstract form

erl_pp

The following functions are exported:

- `form(Form) -> DeepCharList`
  [page 117] Pretty print a form
- `form(Form, HookFunction) -> DeepCharList`
  [page 117] Pretty print a form
- `attribute(Attribute) -> DeepCharList`
  [page 117] Pretty print an attribute
- attribute(Attribute, HookFunction) -> DeepCharList
  [page 117] Pretty print an attribute
- function(Function) -> DeepCharList
  [page 117] Pretty print a function
- function(Function, HookFunction) -> DeepCharList
  [page 117] Pretty print a function
- guard(Guard) -> DeepCharList
  [page 117] Pretty print a guard
- guard(Guard, HookFunction) -> DeepCharList
  [page 117] Pretty print a guard
- exprs(Expressions) -> DeepCharList
  [page 118] Pretty print Expressions
- exprs(Expressions, HookFunction) -> DeepCharList
  [page 118] Pretty print Expressions
- exprs(Expressions, Indent, HookFunction) -> DeepCharList
  [page 118] Pretty print Expressions
- expr(Expression) -> DeepCharList
  [page 118] Pretty print one Expression
- expr(Expression, HookFunction) -> DeepCharList
  [page 118] Pretty print one Expression
- expr(Expression, Indent, HookFunction) -> DeepCharList
  [page 118] Pretty print one Expression
- expr(Expression, Indent, Precedence, HookFunction) -> DeepCharList
  [page 118] Pretty print one Expression

erl_scan

The following functions are exported:

- string(CharList, StartLine) -> {ok, Tokens, EndLine} | Error
  [page 120] Scan a string and returns the Erlang tokens
- string(CharList) -> {ok, Tokens, EndLine} | Error
  [page 120] Scan a string and returns the Erlang tokens
- tokens(Continuation, CharList, StartLine) -> Return
  [page 120] Re-entrant scanner
- reserved_word(Atom) -> bool()
  [page 121] Test for a reserved word
- format_error(ErrorCode) -> string()
  [page 121] Format an error descriptor

erl_tar

The following functions are exported:

- add(TarDescriptor, Filename, Options) -> RetValue
  [page 123] Add a file to an open tar file
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(TarDescriptor, Filename, NameInArchive, Options) -&gt; RetValue</code></td>
<td>Add a file to an open tar file</td>
</tr>
<tr>
<td><code>close(TarDescriptor)</code></td>
<td>Close an open tar file</td>
</tr>
<tr>
<td><code>create(Name, FileList) -&gt; RetValue</code></td>
<td>Create a tar archive</td>
</tr>
<tr>
<td><code>create(Name, FileList, OptionList)</code></td>
<td>Create a tar archive with options</td>
</tr>
<tr>
<td><code>extract(Name) -&gt; RetValue</code></td>
<td>Extract all files from a tar file</td>
</tr>
<tr>
<td><code>extract(Name, OptionList)</code></td>
<td>Extract files from a tar file</td>
</tr>
<tr>
<td><code>format_error(Reason) -&gt; string()</code></td>
<td>Convert error term to a readable string</td>
</tr>
<tr>
<td><code>open(Name, OpenModeList) -&gt; RetValue</code></td>
<td>Open a tar file</td>
</tr>
<tr>
<td><code>table(Name)</code></td>
<td>Retrieve the name of all files in a tar file</td>
</tr>
<tr>
<td><code>table(Name, Options)</code></td>
<td>Retrieve name and information of all files in a tar file</td>
</tr>
<tr>
<td><code>t(Name)</code></td>
<td>Print the name of each file in a tar file</td>
</tr>
<tr>
<td><code>tt(Name)</code></td>
<td>Print name and information for each file in a tar file</td>
</tr>
</tbody>
</table>

**ets**

The following functions are exported:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>all() -&gt; [Tab]</code></td>
<td>Return a list of all ETS tables</td>
</tr>
<tr>
<td><code>delete(Tab) -&gt; true</code></td>
<td>Delete an entire ETS table</td>
</tr>
<tr>
<td><code>delete(Tab, Key) -&gt; true</code></td>
<td>Delete all objects with a given key from an ETS table</td>
</tr>
<tr>
<td><code>delete_all_objects(Tab) -&gt; true</code></td>
<td>Delete all objects in an ETS table</td>
</tr>
<tr>
<td><code>delete_object(Tab, Object) -&gt; true</code></td>
<td>Deletes a specific from an ETS table</td>
</tr>
<tr>
<td>`file2tab(Filename) -&gt; {ok,Tab}</td>
<td>{error,Reason}`</td>
</tr>
<tr>
<td>`first(Tab) -&gt; Key</td>
<td>'$end_of_table'`</td>
</tr>
<tr>
<td>`fixtable(Tab, true</td>
<td>false) -&gt; true</td>
</tr>
<tr>
<td><code>foldl(Function, Acc0, Tab) -&gt; Acc1</code></td>
<td>Fold a function over an ETS table</td>
</tr>
</tbody>
</table>
- foldr(Function, Acc0, Tab) -> Acc1
  [page 130] Fold a function over an ETS table
- from_dets(Tab, DetsTab) -> Tab
  [page 131] Fill an ETS table with objects from a Dets table.
- fun2ms(LiteralFun) -> MatchSpec
  [page 131] Pseudo function that transforms fun syntax to a match_spec.
- i() -> void()
  [page 132] Display information about all ETS tables on tty.
- i(Tab) -> void()
  [page 132] Browse an ETS table on tty.
- info(Tab) -> [[Item, Value]] | undefined
  [page 132] Return information about an ETS table.
- info(Tab, Item) -> Value | undefined
  [page 133] Return the information associated with given item for an ETS table.
- init_table(Name, InitFun) -> true
  [page 133] Replace all objects of an ETS table.
- insert(Tab, ObjectOrObjects) -> true
  [page 134] Insert an object into an ETS table.
- insert_new(Tab, ObjectOrObjects) -> bool()
  [page 134] Insert an object into an ETS table if the key is not already present.
- is_compiled_ms(Term) -> bool()
  [page 134] Checks if an Erlang term is the result of ets_match_spec_compile
- last(Tab) -> ['$end_of_table']
  [page 135] Return the last key in an ETS table of type ordered_set.
- lookup(Tab, Key) -> [Object]
  [page 135] Return all objects with a given key in an ETS table.
- lookup_element(Tab, Key, Pos) -> Elem
  [page 135] Return the Pos:th element of all objects with a given key in an ETS table.
- match(Tab, Pattern) -> [Match]
  [page 136] Match the objects in an ETS table against a pattern.
- match(Tab, Pattern, Limit) -> [{Match},Continuation] | '$end_of_table'
  [page 136] Match the objects in an ETS table against a pattern and returns part of the answers.
- match(Continuation) -> [{Match},Continuation] | '$end_of_table'
  [page 137] Continues matching objects in an ETS table.
- match_delete(Tab, Pattern) -> true
  [page 137] Delete all objects which match a given pattern from an ETS table.
- match_object(Tab, Pattern) -> [Object]
  [page 137] Match the objects in an ETS table against a pattern.
- match_object(Tab, Pattern, Limit) -> [{Match},Continuation] | '$end_of_table'
  [page 137] Match the objects in an ETS table against a pattern and returns part of the answers.
- match_object(Continuation) -> [{Match},Continuation] | '$end_of_table'
  [page 137] Continues matching objects in an ETS table.
- `match_spec_compile(MatchSpec) -> CompiledMatchSpec`  
  [page 138] Compiles a match specification into its internal representation

- `match_spec_run(List,CompiledMatchSpec) -> list()`  
  [page 138] Performs matching, using a compiled match_spec, on a list of tuples

- `member(Tab, Key) -> true | false`  
  [page 139] Tests for occurrence of a key in an ETS table

- `new(Name, Options) -> tid()`  
  [page 139] Create a new ETS table.

- `next(Tab, Key1) -> Key2 | '$end_of_table'`  
  [page 140] Return the next key in an ETS table.

- `prev(Tab, Key1) -> Key2 | '$end_of_table'`  
  [page 140] Return the previous key in an ETS table of type ordered_set.

- `rename(Tab, Name) -> Name`  
  [page 140] Rename a named ETS table.

- `repair_continuation(Continuation, MatchSpec) -> Continuation`  
  [page 140] Repair a continuation from ets:select/1 or ets:select/3 that has passed through external representation

- `safe_fixtable(Tab, true|false) -> true`  
  [page 141] Fix an ETS table for safe traversal.

- `select(Tab, MatchSpec) -> [Match]`  
  [page 142] Match the objects in an ETS table against a match_spec.

- `select(Tab, MatchSpec, Limit) -> [{Match},Continuation] | '$end_of_table'`  
  [page 144] Match the objects in an ETS table against a match_spec and returns part of the answers.

- `select(Continuation) -> [{Match},Continuation] | '$end_of_table'`  
  [page 144] Continue matching objects in an ETS table.

- `select_delete(Tab, MatchSpec) -> NumDeleted`  
  [page 144] 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 144] 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 145] Return all objects in a given slot of an ETS table.

- `tab2file(Tab, Filename) -> ok | {error,Reason}`  
  [page 145] Dump an ETS table to a file.

- `tab2list(Tab) -> [Object]`  
  [page 145] Return a list of all objects in an ETS table.

- `table(Tab [, Options]) -> QueryHandle`  
  [page 145] Return a QLC query handle.

- `test_ms(Tuple, MatchSpec) -> {ok, Result} | {error, Errors}`  
  [page 146] Test a match_spec for use in ets:select/2.

- `to_dets(Tab, DetsTab) -> Tab`  
  [page 147] Fill a Dets table with objects from an ETS table.

- `update_counter(Tab, Key, [Pos,Incr,Threshold,SetValue]) -> Result`  
  [page 147] Update a counter object in an ETS table.
• `update_counter(Tab, Key, {Pos,Incr})` -> Result
  [page 147] Update a counter object in an ETS table.
• `update_counter(Tab, Key, Incr)` -> Result
  [page 147] Update a counter object in an ETS table.

file_sorter

The following functions are exported:

• `sort(FileName)` -> Reply
  [page 151] Sort terms on files.
• `sort(Input, Output)` -> Reply
  [page 151] Sort terms on files.
• `sort(Input, Output, Options)` -> Reply
  [page 151] Sort terms on files.
• `keysort(KeyPos, FileName)` -> Reply
  [page 151] Sort terms on files by key.
• `keysort(KeyPos, Input, Output)` -> Reply
  [page 151] Sort terms on files by key.
• `keysort(KeyPos, Input, Output, Options)` -> Reply
  [page 151] Sort terms on files by key.
• `merge(FileNames, Output)` -> Reply
  [page 151] Merge terms on files.
• `merge(FileNames, Output, Options)` -> Reply
  [page 151] Merge terms on files.
• `keymerge(KeyPos, FileNames, Output)` -> Reply
  [page 152] Merge terms on files by key.
• `keymerge(KeyPos, FileNames, Output, Options)` -> Reply
  [page 152] Merge terms on files by key.
• `check(FileName)` -> Reply
  [page 152] Check whether terms on files are sorted.
• `check(FileNames, Options)` -> Reply
  [page 152] Check whether terms on files are sorted.
• `keycheck(KeyPos, FileName)` -> CheckReply
  [page 152] Check whether terms on files are sorted by key.
• `keycheck(KeyPos, FileNames, Options)` -> Reply
  [page 152] Check whether terms on files are sorted by key.

filelib

The following functions are exported:

• `ensure_dir(Name)` -> ok | {error, Reason}
  [page 153] Ensure that all parent directories for a file or directory exist.
• `file_size(Filename)` -> integer()
• `fold_files(Dir, RegExp, Recursive, Fun, AccIn)` -> AccOut
  [page 153] Fold over all files matching a regular expression.
The following functions are exported:

- `absname(Filename) -> string()`  
  [page 156] Convert a filename to an absolute name, relative the working directory
- `absname(Filename, Dir) -> string()`  
  [page 157] Convert a filename to an absolute name, relative a specified directory
- `absname(join(Dir, Filename) -> string()`  
  [page 157] Join an absolute directory with a relative filename
- `basename(Filename) -> string()`  
  [page 157] Return the last component of a filename
- `basename(Filename, Ext) -> string()`  
  [page 157] Return the last component of a filename, stripped of the specified extension
- `dirname(Filename) -> string()`  
  [page 158] Return the directory part of a path name
- `extension(Filename) -> string()`  
  [page 158] Return the file extension
- `flatten(Filename) -> string()`  
  [page 158] Convert a filename to a flat string
- `join(Components) -> string()`  
  [page 159] Join a list of filename components with directory separators
- `join(Name1, Name2) -> string()`  
  [page 159] Join two filename components with directory separators
- `nativename(Path) -> string()`  
  [page 159] Return the native form of a file path
- `pathtype(Path) -> absolute | relative | volumerelative`  
  [page 159] Return the type of a path
- `rootname(Filename) -> string()`  
  [page 160] Remove a filename extension
- `rootname(Filename, Ext) -> string()`  
  [page 160] Remove a filename extension
split(Filename) -> Components
[page 160] Split a filename into its path components

find(src(Beam)) -> {SourceFile, Options}
[page 160] Find the filename and compiler options for a module

find(src(Beam, Rules)) -> {SourceFile, Options}
[page 160] Find the filename and compiler options for a module

gb_sets

The following functions are exported:

- add(Element, Set1) -> Set2
  [page 163] Add a (possibly existing) element to a gb_set
- add_element(Element, Set1) -> Set2
  [page 163] Add a (possibly existing) element to a gb_set
- balance(Set1) -> Set2
  [page 163] Rebalance tree representation of a gb_set
- delete(Element, Set1) -> Set2
  [page 163] Remove an element from a gb_set
- delete_any(Element, Set1) -> Set2
  [page 163] Remove a (possibly non-existing) element from a gb_set
- del_element(Element, Set1) -> Set2
  [page 163] Remove a (possibly non-existing) element from a gb_set
- difference(Set1, Set2) -> Set3
  [page 163] Return the difference of two gb_sets
- subtract(Set1, Set2) -> Set3
  [page 164] Return the difference of two gb_sets
- empty() -> Set
  [page 164] Return an empty gb_set
- new() -> Set
  [page 164] Return an empty gb_set
- filter(Pred, Set1) -> Set2
  [page 164] Filter gb_set elements
- fold(Function, Acc0, Set) -> Acc1
  [page 164] Fold over gb_set elements
- from_list(List) -> Set
  [page 164] Convert a list into a gb_set
- from_ordset(List) -> Set
  [page 164] Make a gb_set from an ordset list
- insert(Element, Set1) -> Set2
  [page 164] Add a new element to a gb_set
- intersection(Set1, Set2) -> Set3
  [page 165] Return the intersection of two gb_sets
- intersection(SetList) -> Set
  [page 165] Return the intersection of a list of gb_sets
- is_empty(Set) -> bool()
  [page 165] Test for empty gb_set
• `is_member(Element, Set) -> bool()`  
  [page 165] Test for membership of a gb_set

• `is_element(Element, Set) -> bool()`  
  [page 165] Test for membership of a gb_set

• `is_set(Set) -> bool()`  
  [page 165] Test for a gb_set

• `is_subset(Set1, Set2) -> bool()`  
  [page 165] Test for subset

• `iterator(Set) -> Iter`  
  [page 165] Return an iterator for a gb_set

• `largest(Set) -> term()`  
  [page 166] Return largest element

• `next(Iter1) -> {Element, Iter2 | none}`  
  [page 166] Traverse a gb_set with an iterator

• `singleton(Element) -> gb_set()`  
  [page 166] Return a gb_set with one element

• `size(Set) -> int()`  
  [page 166] Return the number of elements in a gb_set

• `smallest(Set) -> term()`  
  [page 166] Return smallest element

• `take_largest(Set1) -> {Element, Set2}`  
  [page 166] Extract largest element

• `take_smallest(Set1) -> {Element, Set2}`  
  [page 166] Extract smallest element

• `to_list(Set) -> List`  
  [page 167] Convert a gb_set into a list

• `union(Set1, Set2) -> Set3`  
  [page 167] Return the union of two gb_sets

• `union(SetList) -> Set`  
  [page 167] Return the union of a list of gb_sets

---

**gb_trees**

The following functions are exported:

• `balance(Tree1) -> Tree2`  
  [page 168] Rebalance a tree

• `delete(Key, Tree1) -> Tree2`  
  [page 168] Remove a node from a tree

• `delete_any(Key, Tree1) -> Tree2`  
  [page 169] Remove a (possibly non-existing) node from a tree

• `empty() -> Tree`  
  [page 169] Return an empty tree

• `enter(Key, Val, Tree1) -> Tree2`  
  [page 169] Insert or update key with value in a tree

• `from_orddict(List) -> Tree`  
  [page 169] Make a tree from an orddict
STDLIB Reference Manual

- `get(Key, Tree) -> Val`
  [page 169] Look up a key in a tree, if present
- `lookup(Key, Tree) -> {value, Val} | none`
  [page 169] Look up a key in a tree
- `insert(Key, Val, Tree1) -> Tree2`
  [page 170] Insert a new key and value in a tree
- `is_defined(Key, Tree) -> bool()`  
  [page 170] Test for membership of a tree
- `is_empty(Tree) -> bool()`  
  [page 170] Test for empty tree
- `iterator(Tree) -> Iter`
  [page 170] Return an iterator for a tree
- `keys(Tree) -> [Key]`
  [page 170] Return a list of the keys in a tree
- `largest(Tree) -> {Key, Val}`
  [page 170] Return largest key and value
- `next(Iter) -> {Key, Val, Iter2}`
  [page 170] Traverse a tree with an iterator
- `size(Tree) -> int()`  
  [page 171] Return the number of nodes in a tree
- `smallest(Tree) -> {Key, Val}`
  [page 171] Return smallest key and value
- `take_largest(Tree1) -> {Key, Val, Tree2}`
  [page 171] Extract largest key and value
- `take_smallest(Tree1) -> {Key, Val, Tree2}`
  [page 171] Extract smallest key and value
- `to_list(Tree) -> [{Key, Val}]`
  [page 171] Convert a tree into a list
- `update(Key, Val, Tree1) -> Tree2`
  [page 171] Update a key to new value in a tree
- `values(Tree) -> [Val]`
  [page 172] Return a list of the values in a tree

**gen_event**

The following functions are exported:

- `start_link() -> Result`
  [page 174] Create a generic event manager process in a supervision tree.
- `start_link(EventMgrName) -> Result`
  [page 174] Create a generic event manager process in a supervision tree.
- `start() -> Result`
  [page 174] Create a stand-alone event manager process.
- `start(EventMgrName) -> Result`
  [page 174] Create a stand-alone event manager process.
- `add_handler(EventMgrRef, Handler, Args) -> Result`
  [page 174] Add an event handler to a generic event manager.
The following functions are exported:

- `gen_fsm:start_link(Module, Args, Options) -> Result`
  [page 184] Create a `gen_fsm` process in a supervision tree.

- `gen_fsm:start_link(FsmName, Module, Args, Options) -> Result`
  [page 184] Create a `gen_fsm` process in a supervision tree.

- `gen_fsm:start(Module, Args, Options) -> Result`
  [page 185] Create a stand-alone `gen_fsm` process.

- `gen_fsm:start(FsmName, Module, Args, Options) -> Result`
  [page 185] Create a stand-alone `gen_fsm` process.
- `send_event(FsmRef, Event) -> ok`
  [page 185] Send an event asynchronously to a generic FSM.
- `send_all_state_event(FsmRef, Event) -> ok`
  [page 186] Send an event asynchronously to a generic FSM.
- `sync_send_event(FsmRef, Event) -> Reply`
  [page 186] Send an event synchronously to a generic FSM.
- `sync_send_event(FsmRef, Event, Timeout) -> Reply`
  [page 186] Send an event synchronously to a generic FSM.
- `sync_send_all_state_event(FsmRef, Event) -> Reply`
  [page 186] Send an event synchronously to a generic FSM.
- `sync_send_all_state_event(FsmRef, Event, Timeout) -> Reply`
  [page 186] Send an event synchronously to a generic FSM.
- `reply(Caller, Reply) -> true`
  [page 187] Send a reply to a caller.
- `send_event_after(Time, Event) -> Ref`
  [page 187] Send a delayed event internally in a generic FSM.
- `start_timer(Time, Msg) -> Ref`
  [page 187] Send a timeout event internally in a generic FSM.
- `cancel_timer(Ref) -> RemainingTime | false`
  [page 188] Cancel an internal timer in a generic FSM.
- `enter_loop(Module, Options, StateName, StateData)`
  [page 188] Enter the gen_fsm receive loop
- `enter_loop(Module, Options, StateName, StateData, FsmName)`
  [page 188] Enter the gen_fsm receive loop
- `enter_loop(Module, Options, StateName, StateData, Timeout)`
  [page 188] Enter the gen_fsm receive loop
- `enter_loop(Module, Options, StateName, StateData, FsmName, Timeout)`
  [page 188] Enter the gen_fsm receive loop
- `Module:init(Args) -> Result`
  [page 189] Initialize process and internal state name and state data.
- `Module:StateName(Event, StateData) -> Result`
  [page 190] Handle an asynchronous event.
- `Module:handle_event(Event, StateName, StateData) -> Result`
  [page 190] Handle an asynchronous event.
- `Module:StateName(Event, From, StateData) -> Result`
  [page 191] Handle a synchronous event.
- `Module:handle_sync_event(Event, From, StateName, StateData) -> Result`
  [page 191] Handle a synchronous event.
- `Module:handle_info(Info, StateName, StateData) -> Result`
  [page 192] Handle an incoming message.
- `Module:terminate(Reason, StateName, StateData)`
  [page 192] Clean up before termination.
- `Module:code_change(OldVsn, StateName, StateData, Extra) -> {ok, NextStateName, NewStateData}`
  [page 193] Update the internal state data during upgrade/downgrade.
gen_server

The following functions are exported:

- `start_link(Module, Args, Options) -> Result`
  [page 194] Create a gen_server process in a supervision tree.
- `start_link(ServerName, Module, Args, Options) -> Result`
  [page 194] Create a gen_server process in a supervision tree.
- `start(Module, Args, Options) -> Result`
  [page 195] Create a stand-alone gen_server process.
- `start(ServerName, Module, Args, Options) -> Result`
  [page 195] Create a stand-alone gen_server process.
- `call(ServerRef, Request) -> Reply`
  [page 196] Make a synchronous call to a generic server.
- `call(ServerRef, Request, Timeout) -> Reply`
  [page 196] Make a synchronous call to a generic server.
- `multi_cast(Name, Request) -> Result`
  [page 197] Make a synchronous call to several generic servers
- `multi_cast(Nodes, Name, Request) -> Result`
  [page 197] Make a synchronous call to several generic servers
- `multi_cast(Nodes, Name, Request, Timeout) -> Result`
  [page 197] Make a synchronous call to several generic servers
- `cast(ServerRef, Request) -> ok`
  [page 198] Send an asynchronous request to a generic server.
- `abcast(Name, Request) -> abcast`
  [page 198] Send an asynchronous request to several generic servers.
- `abcast(Nodes, Name, Request) -> abcast`
  [page 198] Send an asynchronous request to several generic servers.
- `reply(Client, Reply) -> true`
  [page 198] Send a reply to a client.
- `enter_loop(Module, Options, State)`
  [page 199] Enter the gen_server receive loop
- `enter_loop(Module, Options, State, ServerName)`
  [page 199] Enter the gen_server receive loop
- `enter_loop(Module, Options, State, Timeout)`
  [page 199] Enter the gen_server receive loop
- `enter_loop(Module, Options, State, ServerName, Timeout)`
  [page 199] Enter the gen_server receive loop
- `Module:init(Args) -> Result`
  [page 200] Initialize process and internal state.
- `Module:handle_call(Request, From, State) -> Result`
  [page 200] Handle a synchronous request.
- `Module:handle_cast(Request, State) -> Result`
  [page 201] Handle an asynchronous request.
- `Module:handle_info(Info, State) -> Result`
  [page 201] Handle an incoming message.
- Module: `terminate(Reason, State)`  
  [page 201] Clean up before termination.
- Module: `code_change(OldVsn, State, Extra) -> {ok, NewState}`  
  [page 202] Update the internal state during upgrade/downgrade.

**io**

The following functions are exported:

- `put_chars([IoDevice,] IoData) -> ok`  
  [page 203] Write a list of characters
- `nl([IoDevice]) -> ok`  
  [page 203] Write a newline
- `get_chars([IoDevice,] Prompt, Count) -> string() | eof`  
  [page 203] Read a specified number of characters
- `get_line([IoDevice,] Prompt) -> string() | eof`  
  [page 204] Read a line
- `setopts([IoDevice,] Opts) -> ok | {error, Reason}`  
  [page 204] Set options
- `write([IoDevice,] Term) -> ok`  
  [page 204] Write a term
- `read([IoDevice,] Prompt) -> Result`  
  [page 205] Read a term
- `read([IoDevice,] Prompt, StartLine) -> Result`  
  [page 205] Read a term
- `fwrite(Format) -> Result`  
  [page 205] Write formatted output
- `fwrite([IoDevice,] Format, Data) -> ok`  
  [page 205] Write formatted output
- `format(Format) -> Result`  
  [page 205] Write formatted output
- `format([IoDevice,] Format, Data) -> ok`  
  [page 205] Write formatted output
- `fread([IoDevice,] Prompt, Format) -> Result`  
  [page 209] Read formatted input
- `scan_erl_exprs(Prompt) -> Result`  
  [page 211] Read and tokenize Erlang expressions
- `scan_erl_exprs([IoDevice,] Prompt, StartLine) -> Result`  
  [page 211] Read and tokenize Erlang expressions
- `scan_erl_form(Prompt) -> Result`  
  [page 211] Read and tokenize an Erlang form
- `scan_erl_form([IoDevice,] Prompt, StartLine) -> Result`  
  [page 211] Read and tokenize an Erlang form
- `parse_erl_exprs(Prompt) -> Result`  
  [page 212] Read, tokenize and parse Erlang expressions
- `parse_erl_exprs([IoDevice,] Prompt, StartLine) -> Result`  
  [page 212] Read, tokenize and parse Erlang expressions
io_lib

The following functions are exported:

- `nl()` -> `chars()`
  [page 214] Write a newline
- `write(Term)` ->
  [page 214] Write a term
- `write(Term, Depth)` -> `chars()`
  [page 214] Write a term
- `print(Term)` ->
  [page 214] Pretty print a term
- `print(Term, Column, LineLength, Depth)` -> `chars()`
  [page 214] Pretty print a term
- `fwrite(Format, Data)` ->
  [page 215] Write formatted output
- `format(Format, Data)` -> `chars()`
  [page 215] Write formatted output
- `fread(Format, String)` -> `Result`
  [page 215] Read formatted input
- `fread(Continuation, String, Format)` -> `Result`
  [page 215] Re-entrant formatted reader
- `write_atom(Atom)` -> `chars()`
  [page 216] Write an atom
- `write_string(String)` -> `chars()`
  [page 216] Write a string
- `write_char(Integer)` -> `chars()`
  [page 216] Write a character
- `indentation(String, StartIndent)` -> `int()`
  [page 216] Indentation after printing string
- `char_list(Term)` -> `bool()`
  [page 217] Test for a list of characters
- `deep_char_list(Term)` -> `bool()`
  [page 217] Test for a deep list of characters
- `printable_list(Term)` -> `bool()`
  [page 217] Test for a list of printable characters
lib

The following functions are exported:

- `flush_receive() -> void()` [page 218] Flush messages
- `error_message(Format, Args) -> ok` [page 218] Print error message
- `proname() -> atom()` [page 218] Return name of Erlang start script
- `nonl(String1) -> String2` [page 218] Remove last newline
- `send(To, Msg)` [page 218] Send a message
- `sendw(To, Msg)` [page 219] Send a message and wait for an answer

lists

The following functions are exported:

- `append(ListOfLists) -> List1` [page 220] Append a list of lists
- `append(List1, List2) -> List3` [page 220] Append two lists
- `concat(Things) -> string()` [page 220] Concatenate a list of atoms
- `delete(Elem, List1) -> List2` [page 221] Delete an element from a list
- `duplicate(N, Elem) -> List` [page 221] Make N copies of element
- `flatlength(DeepList) -> int()` [page 221] Length of flattened deep list
- `flatten(DeepList) -> List` [page 221] Flatten a deep list
- `flatten(DeepList, Tail) -> List` [page 221] Flatten a deep list
- `keydelete(Key, N, TupleList1) -> TupleList2` [page 221] Delete an element from a list of tuples
- `keymember(Key, N, TupleList) -> bool()` [page 222] Test for membership of a list of tuples
- `keymerge(N, TupleList1, TupleList2) -> TupleList3` [page 222] Merge two key-sorted lists of tuples
- `keyreplace(Key, N, TupleList1, NewTuple) -> TupleList2` [page 222] Replace an element in a list of tuples
- `keysearch(Key, N, TupleList) -> {value, Tuple} | false` [page 222] Search for an element in a list of tuples
- `keysort(N, TupleList1) -> TupleList2`
  [page 222] Sort a list of tuples

- `last(List) -> Last`
  [page 223] Return last element in a list

- `max(List) -> Max`
  [page 223] Return maximum element of list

- `member(Elem, List) -> bool()`
  [page 223] Test for membership of a list

- `merge(ListOfLists) -> List1`
  [page 223] Merge a list of sorted lists

- `merge(List1, List2) -> List3`
  [page 223] Merge two sorted lists

- `merge3(List1, List2, List3) -> List4`
  [page 224] Merge three sorted lists

- `min(List) -> Min`
  [page 224] Return minimum element of list

- `nth(N, List) -> Elem`
  [page 224] Return the Nth element of a list

- `nthtail(N, List1) -> Tail`
  [page 224] Return the Nth tail of a list

- `prefix(List1, List2) -> bool()`
  [page 224] Test for list prefix

- `reverse(List1) -> List2`
  [page 225] Reverse a list

- `reverse(List1, Tail) -> List2`
  [page 225] Reverse a list appending a tail

- `seq(From, To) -> Seq`
  [page 225] Generate a sequence of integers

- `seq(From, To, Incr) -> Seq`
  [page 225] Generate a sequence of integers

- `sort(List1) -> List2`
  [page 225] Sort a list

- `sort(Fun, List1) -> List2`
  [page 225] Sort a list

- `split(N, List1) -> {List2, List3}`
  [page 226] Split a list into two lists

- `sublist(List1, Len) -> List2`
  [page 226] Return a sub-list of a certain length, starting at the first position

- `sublist(List1, Start, Len) -> List2`
  [page 226] Return a sub-list starting at a given position and with a given number of elements

- `subtract(List1, List2) -> List3`
  [page 226] Subtract the element in one list from another list

- `suffix(List1, List2) -> bool()`
  [page 227] Test for list suffix
- `sum(List) -> number()` [page 227] Return sum of elements in a list
- `ukeymerge(N, TupleList1, TupleList2) -> TupleList3` [page 227] Merge two key-sorted lists of tuples, removing duplicates
- `ukeysort(N, TupleList1) -> TupleList2` [page 227] Sort a list of tuples, removing duplicates
- `umerge(List0fLists) -> List1` [page 227] Merge a list of sorted lists, removing duplicates
- `umerge(List1, List2) -> List3` [page 227] Merge two sorted lists, removing duplicates
- `umerge(Fun, List1, List2) -> List3` [page 228] Merge two sorted lists, removing duplicates
- `umerge3(List1, List2, List3) -> List4` [page 228] Merge three sorted lists, removing duplicates
- `unzip(List1) -> {List2, List3}` [page 228] Unzip a list of two-tuples into two lists
- `unzip3(List1) -> {List2, List3, List4}` [page 228] Unzip a list of three-tuples into three lists
- `usort(List1) -> List2` [page 228] Sort a list, removing duplicates
- `usort(Fun, List1) -> List2` [page 229] Sort a list, removing duplicates
- `zip(List1, List2) -> List3` [page 229] Zip two lists into a list of two-tuples
- `zip3(List1, List2, List3) -> List4` [page 229] Zip three lists into a list of three-tuples
- `zipwith(Combine, List1, List2) -> List3` [page 229] Zip two lists into one list according to a fun
- `zipwith3(Combine, List1, List2, List3) -> List4` [page 230] Zip three lists into one list according to a fun
- `all(Pred, List) -> bool()` [page 230] Return true if all elements in the list satisfy Pred
- `any(Pred, List) -> bool()` [page 230] Return true if any of the elements in the list satisfies Pred
- `dropwhile(Pred, List1) -> List2` [page 231] Drop elements from a list while a predicate is true
- `filter(Pred, List1) -> List2` [page 231] Choose elements which satisfy a predicate
- `flatMap(Fun, List1) -> List2` [page 231] Map and flatten in one pass
- `foldl(Fun, Acc0, List) -> Acc1` [page 231] Fold a function over a list
- `foldr(Fun, Acc0, List) -> Acc1` [page 232] Fold a function over a list
- `foreach(Fun, List) -> void()` [page 232] Apply a function to each element of a list
• keymap(Fun, N, TupleList1) -> TupleList2
  [page 232] Map a function over a list of tuples

• map(Fun, List1) -> List2
  [page 233] Map a function over a list

• mapfold1(Fun, Acc0, List1) -> {List2, Acc1}
  [page 233] Map and fold in one pass

• mapfoldr(Fun, Acc0, List1) -> {List2, Acc1}
  [page 233] Map and fold in one pass

• partition(Pred, List) -> {Satisfying, NonSatisfying}
  [page 233] Partition a list into two lists based on a predicate

• splitwith(Pred, List) -> {List1, List2}
  [page 234] Split a list into two lists based on a predicate

• takewhile(Pred, List1) -> List2
  [page 234] Take elements from a list while a predicate is true

log_mf_h

The following functions are exported:

• init(Dir, MaxBytes, MaxFiles)
  [page 235] Initiate the event handler

• init(Dir, MaxBytes, MaxFiles, Pred) -> Args
  [page 235] Initiate the event handler

math

The following functions are exported:

• pi() -> float()
  [page 236] A useful number

• sin(X)
  [page 236] Diverse math functions

• cos(X)
  [page 236] Diverse math functions

• tan(X)
  [page 236] Diverse math functions

• asin(X)
  [page 236] Diverse math functions

• acos(X)
  [page 236] Diverse math functions

• atan(X)
  [page 236] Diverse math functions

• atan2(Y, X)
  [page 236] Diverse math functions

• sinh(X)
  [page 236] Diverse math functions

• cosh(X)
  [page 236] Diverse math functions
• \( \tanh(X) \)  
  [page 236] Diverse math functions
• \( \text{asinh}(X) \)  
  [page 236] Diverse math functions
• \( \text{acosh}(X) \)  
  [page 236] Diverse math functions
• \( \text{atanh}(X) \)  
  [page 236] Diverse math functions
• \( \exp(X) \)  
  [page 236] Diverse math functions
• \( \log(X) \)  
  [page 236] Diverse math functions
• \( \log_{10}(X) \)  
  [page 236] Diverse math functions
• \( \text{pow}(X, Y) \)  
  [page 236] Diverse math functions
• \( \sqrt{X} \)  
  [page 236] Diverse math functions
• \( \text{erf}(X) \rightarrow \text{float}() \)  
  [page 237] Error function.
• \( \text{erfc}(X) \rightarrow \text{float}() \)  
  [page 237] Another error function

ms_transform

The following functions are exported:

• \( \text{parse_transform}(\text{Forms, Options}) \rightarrow \text{Forms} \)  
  [page 247] Transforms Erlang abstract format containing calls to ets/dbg:fun2ms into literal match specifications.
• \( \text{transform_from_shell}(\text{Dialect, Clauses, BoundEnvironment}) \rightarrow \text{term()} \)  
  [page 247] Used when transforming fun’s created in the shell into match specifications.
• \( \text{format_error}(\text{Errcode}) \rightarrow \text{ErrMessage} \)  
  [page 248] 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 251] Create an empty group
- `create(PgName, Node) -> ok | {error, Reason}`
  [page 251] Create an empty group on another node
- `join(PgName, Pid) -> Members`
  [page 251] Join a pid to a process group
- `send(PgName, Msg) -> void()`
  [page 252] Send a message to all members of a process group
- `esend(PgName, Msg) -> void()`
  [page 252] Send a message to all members of a process group, except ourselves
- `members(PgName) -> Members`
  [page 252] Return a list of all members of a process group

pool

The following functions are exported:

- `start(Name) ->`
  [page 253] Start a new pool
- `start(Name, Args) -> Nodes`
  [page 253] Start a new pool
- `attach(Node) -> allready_attached | attached`
  [page 253] Ensure that a pool master is running
- `stop() -> stopped`
  [page 254] Stop the pool and kill all the slave nodes
- `get_nodes() -> Nodes`
  [page 254] Return a list of the current member nodes of the pool
- `pspawn(Mod, Fun, Args) -> pid()`
  [page 254] Spawn a process on the pool node with expected lowest future load
- `pspawn_link(Mod, Fun, Args) -> pid()`
  [page 254] Spawn and link to a process on the pool node with expected lowest future load
- `get_node() -> node()`
  [page 254] Return the node with the expected lowest future load

proc_lib

The following functions are exported:

- `spawn(Fun) -> pid()`
  [page 255] Spawn a new process
- `spawn(Node, Fun) -> pid()`
  [page 255] Spawn a new process
- `spawn(Module, Function, Args) -> pid()`
  [page 255] Spawn a new process
- `spawn(Node, Module, Function, Args) -> pid()`  
  [page 255] Spawn a new process.
- `spawn_link(Fun) -> pid()`  
  [page 255] Spawn and link to a new process.
- `spawn_link(Node, Fun) -> pid()`  
  [page 255] Spawn and link to a new process.
- `spawn_link(Module, Function, Args) -> pid()`  
  [page 255] Spawn and link to a new process.
- `spawn_link(Node, Module, Function, Args) -> pid()`  
  [page 256] Spawn and link to a new process.
- `spawn_opt(Fun, SpawnOpts) -> pid()`  
  [page 256] Spawn a new process with given options.
- `spawn_opt(Node, Fun, SpawnOpts) -> pid()`  
  [page 256] Spawn a new process with given options.
- `spawn_opt(Module, Function, Args, SpawnOpts) -> pid()`  
  [page 256] Spawn a new process with given options.
- `spawn_opt(Node, Module, Func, Args, SpawnOpts) -> pid()`  
  [page 256] Spawn a new process with given options.
- `start(Module, Function, Args) -> Ret`  
  [page 256] Start a new process synchronously.
- `start(Module, Function, Args, Time) -> Ret`  
  [page 256] Start a new process synchronously.
- `start(Module, Function, Args, Time, SpawnOpts) -> Ret`  
  [page 256] Start a new process synchronously.
- `start_link(Module, Function, Args) -> Ret`  
  [page 256] Start a new process synchronously.
- `start_link(Module, Function, Args, Time) -> Ret`  
  [page 256] Start a new process synchronously.
- `start_link(Module, Function, Args, Time, SpawnOpts) -> Ret`  
  [page 256] Start a new process synchronously.
- `init_ack(Parent, Ret) -> void()`  
  [page 257] Used by a process when it has started.
- `init_ack(Ret) -> void()`  
  [page 257] Used by a process when it has started.
- `format(CrashReport) -> string()`  
  [page 258] Format a crash report.
- `initial_call(Process) -> {Module,Function,Args} | Fun | false`  
  [page 258] Extract the initial call of a proc_lib spawned process.
- `translate_initial_call(Process) -> {Module,Function,Arity} | Fun`  
  [page 258] Extract and translate the initial call of a proc_lib spawned process.
- `hibernate(Module, Function, Args)`  
  [page 259] Hibernate a process until a message is sent to it.
proplists

The following functions are exported:

- `append_values(Key, List) -> List` [page 260]
- `compact(List) -> List` [page 260]
- `delete(Key, List) -> List` [page 260]
- `expand(Expansions, List) -> List` [page 260]
- `get_all_values(Key, List) -> [term()]` [page 261]
- `get_bool(Key, List) -> bool()` [page 261]
- `get_keys(List) -> [term()]` [page 261]
- `get_value(Key, List) -> term()` [page 262]
- `get_value(Key, List, Default) -> term()` [page 262]
- `is_defined(Key, List) -> bool()` [page 262]
- `lookup(Key, List) -> none | tuple()` [page 262]
- `lookup_all(Key, List) -> [tuple()]` [page 262]
- `normalize(List, Stages) -> List` [page 262]
- `property(Property) -> Property` [page 263]
- `property(Key, Value) -> Property` [page 263]
- `split(List, Keys) -> {Lists, Rest}` [page 263]
- `substitute_aliases(Aliases, List) -> List` [page 264]
- `substitute_negations(Negations, List) -> List` [page 264]
- `unfold(List) -> List` [page 264]
qlc

The following functions are exported:

- `append(QHL) => QH` [page 272] Return a query handle.
- `append(QH1, QH2) => QH3` [page 272] Return a query handle.
- `cursor(QueryHandleOrList [, Options]) => QueryCursor` [page 272] Create a query cursor.
- `delete_cursor(QueryCursor) => ok` [page 273] Delete a query cursor.
- `eval(QueryHandleOrList [, Options]) => Answers | Error` [page 273] Return all answers to a query.
- `e(QueryHandleOrList [, Options]) => Answers` [page 273] Return all answers to a query.
- `fold(Function, Acc0, QueryHandleOrList [, Options]) => Acc1 | Error` [page 273] Fold a function over the answers to a query.
- `info(QueryHandleOrList [, Options]) => Info` [page 274] Return code describing a query handle.
- `keysort(KeyPos, QH1 [, SortOptions]) => QH2` [page 275] Return a query handle.
- `next_answers(QueryCursor [, NumberOfAnswers]) => Answers | Error` [page 275] Return some or all answers to a query.
- `q(QueryListComprehension [, Options]) => QueryHandle` [page 275] Return a handle for a query list comprehension.
- `sort(QH1 [, SortOptions]) => QH2` [page 278] Return a query handle.
- `string_to_handle(QueryString [, Options [, Bindings]]]) => QueryHandle | Error` [page 278] Return a handle for a query list comprehension.
- `table(TraverseFun, Options) => QueryHandle` [page 279] Return a query handle for a table.

queue

The following functions are exported:

- `cons(Item, Q1) => Q2` [page 282] Insert an item at the head of a queue
- `daeh(Q) => Item` [page 282] Return the last item of a queue
- `from_list(L) => queue()` [page 282] Convert a list to a queue
- `head(Q) => Item` [page 282] Return the item at the head of a queue
The following functions are exported:

- `in(Item, Q1) -> Q2`
  [page 282] Insert an item at the tail of a queue
- `in_r(Item, Q1) -> Q2`
  [page 283] Insert an item at the head of a queue
- `init(Q1) -> Q2`
  [page 283] Remove the last item from a queue
- `is_empty(Q) -> true | false`
  [page 283] Test if a queue is empty
- `join(Q1, Q2) -> Q3`
  [page 283] Join two queues
- `lait(Q1) -> Q2`
  [page 283] Remove the last item from a queue
- `last(Q) -> Item`
  [page 283] Return the last item of a queue
- `len(Q) -> N`
  [page 283] Get the length of a queue
- `new() -> Q`
  [page 284] Create a new empty FIFO queue
- `out(Q1) -> Result`
  [page 284] Remove the head item from a queue
- `out_r(Q1) -> Result`
  [page 284] Remove the last item from a queue
- `reverse(Q1) -> Q2`
  [page 284] Reverse a queue
- `snoc(Q1, Item) -> Q2`
  [page 284] Insert an item at the end of a queue
- `split(N, Q1) -> {Q2,Q3}`
  [page 284] Split a queue in two
- `tail(Q1) -> Q2`
  [page 284] Remove the head item from a queue
- `to_list(Q) -> list()`
  [page 285] Convert a queue to a list

### random

The following functions are exported:

- `seed() -> ran()`
  [page 286] Seeds random number generation with default values
- `seed(A1, A2, A3) -> ran()`
  [page 286] Seeds random number generator
- `seed0() -> ran()`
  [page 286] Return default state for random number generation
- `uniform() -> float()`
  [page 286] Return a random float
- `uniform(N) -> int()`
  [page 286] Return a random integer
• **uniform(State0) -> {float(), State1}**  
  [page 287] Return a random float

• **uniform(N, State0) -> {int(), State1}**  
  [page 287] Return a random integer

### regexp

The following functions are exported:

• **match(String, RegExp) -> MatchRes**  
  [page 288] Match a regular expression

• **first_match(String, RegExp) -> MatchRes**  
  [page 288] Match a regular expression

• **matches(String, RegExp) -> MatchRes**  
  [page 288] Match a regular expression

• **sub(String, RegExp, New) -> SubRes**  
  [page 289] Substitute the first occurrence of a regular expression

• **gsub(String, RegExp, New) -> SubRes**  
  [page 289] Substitute all occurrences of a regular expression

• **split(String, RegExp) -> SplitRes**  
  [page 289] Split a string into fields

• **sh_to_awk(ShRegExp) -> AwkRegExp**  
  [page 290] Convert an sh regular expression into an AWK one

• **parse(RegExp) -> ParseRes**  
  [page 290] Parse a regular expression

• **format_error(ErrorDescriptor) -> Chars**  
  [page 290] Format an error descriptor

### sets

The following functions are exported:

• **new() -> Set**  
  [page 293] Return an empty set

• **is_set(Set) -> bool()**  
  [page 293] Test for an Set

• **size(Set) -> int()**  
  [page 293] Return the number of elements in a set

• **to_list(Set) -> List**  
  [page 293] Convert an Set into a list

• **from_list(List) -> Set**  
  [page 293] Convert a list into an Set

• **is_element(Element, Set) -> bool()**  
  [page 293] Test for membership of an Set

• **add_element(Element, Set1) -> Set2**  
  [page 294] Add an element to an Set

• **del_element(Element, Set1) -> Set2**  
  [page 294] Remove an element from an Set
union(Set1, Set2) -> Set3
[page 294] Return the union of two Sets
union(SetList) -> Set
[page 294] Return the union of a list of Sets
intersection(Set1, Set2) -> Set3
[page 294] Return the intersection of two Sets
intersection(SetList) -> Set
[page 294] Return the intersection of a list of Sets
subtract(Set1, Set2) -> Set3
[page 294] Return the difference of two Sets
is_subset(Set1, Set2) -> bool()
[page 295] Test for subset
fold(Function, Acc0, Set) -> Acc1
[page 295] Fold over set elements
filter(Pred, Set1) -> Set2
[page 295] Filter set elements

shell
The following functions are exported:

- history(N) -> integer()
  [page 304] Sets the number of previous commands to keep
- results(N) -> integer()
  [page 304] Sets the number of previous commands to keep
- start_restricted(Module) -> ok
  [page 304] Exits a normal shell and starts a restricted shell.
- stop_restricted() -> ok
  [page 304] Exits a restricted shell and starts a normal shell.

shell_default
No functions are exported.

slave
The following functions are exported:

- start(Host) ->
  [page 306] Start a slave node on a host
- start(Host, Name) ->
  [page 306] Start a slave node on a host
- start(Host, Name, Args) -> {ok, Node} | {error, Reason}
  [page 306] Start a slave node on a host
- start_link(Host) ->
  [page 307] Start and link to a slave node on a host
• `start_link(Host, Name) ->`  
  [page 307] Start and link to a slave node on a host

• `start_link(Host, Name, Args) -> ok, Node`  
  [error, Reason]  
  [page 307] Start and link to a slave node on a host

• `stop(Node) -> ok`  
  [page 308] Stop (kill) a node

• `pseudo([Master | ServerList]) -> ok`  
  [page 308] Start a number of pseudo servers

• `pseudo(Master, ServerList) -> ok`  
  [page 308] Start a number of pseudo servers

• `relay(Pid)`  
  [page 308] Run a pseudo server

**sofs**

The following functions are exported:

• `a_function(Tuples [, Type]) -> Function`  
  [page 313] Create a function.

• `canonical_relation(SetOfSets) -> BinRel`  
  [page 313] Return the canonical map.

• `composite(Function1, Function2) -> Function3`  
  [page 313] Return the composite of two functions.

• `constant_function(Set, AnySet) -> Function`  
  [page 313] Create the function that maps each element of a set onto another set.

• `converse(BinRel1) -> BinRel2`  
  [page 314] Return the converse of a binary relation.

• `difference(Set1, Set2) -> Set3`  
  [page 314] Return the difference of two sets.

• `digraph_to_family(Graph [, Type]) -> Family`  
  [page 314] Create a family from a directed graph.

• `domain(BinRel) -> Set`  
  [page 314] Return the domain of a binary relation.

• `drestriction(BinRel1, Set) -> BinRel2`  
  [page 314] Return a restriction of a binary relation.

• `drestriction(SetFun, Set1, Set2) -> Set3`  
  [page 315] Return a restriction of a relation.

• `empty_set() -> Set`  
  [page 315] Return the untyped empty set.

• `extension(BinRel1, Set, AnySet) -> BinRel2`  
  [page 315] Extend the domain of a binary relation.

• `family(Tuples [, Type]) -> Family`  
  [page 316] Create a family of subsets.

• `family_difference(Family1, Family2) -> Family3`  
  [page 316] Return the difference of two families.

• `family_domain(Family1) -> Family2`  
  [page 316] Return a family of domains.
- `family_field(Family1) -> Family2` (page 316) Return a family of fields.
- `family_intersection(Family1) -> Family2` (page 317) Return the intersection of a family of sets of sets.
- `family_intersection(Family1, Family2) -> Family3` (page 317) Return the intersection of two families.
- `family_projection(SetFun, Family1) -> Family2` (page 317) Return a family of modified subsets.
- `family_range(Family1) -> Family2` (page 317) Return a family of ranges.
- `family_specification(Fun, Family1) -> Family2` (page 318) Select a subset of a family using a predicate.
- `family_to_digraph(Family [], GraphType) -> Graph` (page 318) Create a directed graph from a family.
- `family_to_relation(Family) -> BinRel` (page 318) Create a binary relation from a family.
- `family_union(Family1) -> Family2` (page 319) Return the union of a family of sets of sets.
- `family_union(Family1, Family2) -> Family3` (page 319) Return the union of two families.
- `field(BinRel) -> Set` (page 319) Return the field of a binary relation.
- `from_external(ExternalSet, Type) -> AnySet` (page 319) Create a set.
- `from_sets(ListOfSets) -> Set` (page 319) Create a set out of a list of sets.
- `from_sets(TupleOfSets) -> Ordset` (page 319) Create an ordered set out of a tuple of sets.
- `from_term(Term [], Type) -> AnySet` (page 320) Create a set.
- `image(BinRel, Set1) -> Set2` (page 321) Return the image of a set under a binary relation.
- `intersection(SetOfSets) -> Set` (page 321) Return the intersection of a set of sets.
- `intersection(Set1, Set2) -> Set3` (page 321) Return the intersection of two sets.
- `intersection_of_family(Family) -> Set` (page 321) Return the intersection of a family.
- `inverse(Function1) -> Function2` (page 321) Return the inverse of a function.
- `inverse_image(BinRel, Set1) -> Set2` (page 322) Return the inverse image of a set under a binary relation.
- `is_a_function(BinRel) -> Bool` (page 322) Test for a function.
- `is_disjoint(Set1, Set2) -> Bool` (page 322) Test for disjoint sets.
• `is_empty_set(AnySet) -> Bool`  
  [page 322] Test for an empty set.
• `is_equal(AnySet1, AnySet2) -> Bool`  
  [page 322] Test two sets for equality.
• `is_set(AnySet) -> Bool`  
  [page 322] Test for an unordered set.
• `is_sofs_set(Term) -> Bool`  
  [page 323] Test for an unordered set.
• `is_subset(Set1, Set2) -> Bool`  
  [page 323] Test two sets for subset.
• `is_type(Term) -> Bool`  
  [page 323] Test for a type.
• `join(Relation1, I, Relation2, J) -> Relation3`  
  [page 323] Return the join of two relations.
• `multiple_relative_product(TupleOfBinRels, BinRel1) -> BinRel2`  
  [page 323] Return the multiple relative product of a tuple of binary relations and a relation.
• `no_elements(ASet) -> NoElements`  
  [page 324] Return the number of elements of a set.
• `partition(SetOfSets) -> Partition`  
  [page 324] Return the coarsest partition given a set of sets.
• `partition(SetFun, Set) -> Partition`  
  [page 324] Return a partition of a set.
• `partition(SetFun, Set1, Set2) -> {Set3, Set4}`  
  [page 324] Return a partition of a set.
• `partition_family(SetFun, Set) -> Family`  
  [page 325] Return a family indexing a partition.
• `product(TupleOfSets) -> Relation`  
  [page 325] Return the Cartesian product of a tuple of sets.
• `product(Set1, Set2) -> BinRel`  
  [page 325] Return the Cartesian product of two sets.
• `projection(SetFun, Set1) -> Set2`  
  [page 326] Return a set of substituted elements.
• `range(BinRel) -> Set`  
  [page 326] Return the range of a binary relation.
• `relation(Tuples [, Type]) -> Relation`  
  [page 326] Create a relation.
• `relation_to_family(BinRel) -> Family`  
  [page 326] Create a family from a binary relation.
• `relative_product(TupleOfBinRels [, BinRel1]) -> BinRel2`  
  [page 327] Return the relative product of a tuple of binary relations and a binary relation.
• `relative_product(BinRel1, BinRel2) -> BinRel3`  
  [page 327] Return the relative product of two binary relations.
- \texttt{restriction(BinRel1, Set)} \rightarrow \texttt{BinRel2}  \\
  \texttt{[page 327]} Return a restriction of a binary relation.

- \texttt{restriction(SetFun, Set1, Set2)} \rightarrow \texttt{Set3}  \\
  \texttt{[page 328]} Return a restriction of a set.

- \texttt{set(Terms [\_], Type)} \rightarrow \texttt{Set}  \\
  \texttt{[page 328]} Create a set of atoms or any type of sets.

- \texttt{specification(Fun, Set1)} \rightarrow \texttt{Set2}  \\
  \texttt{[page 328]} Select a subset using a predicate.

- \texttt{strict_relation(BinRel1)} \rightarrow \texttt{BinRel2}  \\
  \texttt{[page 328]} Return the strict relation corresponding to a given relation.

- \texttt{substitution(SetFun, Set1)} \rightarrow \texttt{Set2}  \\
  \texttt{[page 329]} Return a function with a given set as domain.

- \texttt{symdiff(Set1, Set2)} \rightarrow \texttt{Set3}  \\
  \texttt{[page 329]} Return the symmetric difference of two sets.

- \texttt{symmetric_partition(Set1, Set2)} \rightarrow \{\texttt{Set3, Set4, Set5}\}  \\
  \texttt{[page 330]} Return a partition of two sets.

- \texttt{to_external(AnySet)} \rightarrow \texttt{ExternalSet}  \\
  \texttt{[page 330]} Return the elements of a set.

- \texttt{to_sets(ASet)} \rightarrow \texttt{Sets}  \\
  \texttt{[page 330]} Return a list or a tuple of the elements of set.

- \texttt{type(AnySet)} \rightarrow \texttt{Type}  \\
  \texttt{[page 330]} Return the type of a set.

- \texttt{union(SetOfSets)} \rightarrow \texttt{Set}  \\
  \texttt{[page 330]} Return the union of a set of sets.

- \texttt{union(Set1, Set2)} \rightarrow \texttt{Set3}  \\
  \texttt{[page 330]} Return the union of two sets.

- \texttt{union_of_family(Family)} \rightarrow \texttt{Set}  \\
  \texttt{[page 331]} Return the union of a family.

- \texttt{weak_relation(BinRel1)} \rightarrow \texttt{BinRel2}  \\
  \texttt{[page 331]} Return the weak relation corresponding to a given relation.

\subsection*{string}

The following functions are exported:

- \texttt{len(String)} \rightarrow \texttt{Length}  \\
  \texttt{[page 332]} Return the length of a string

- \texttt{equal(String1, String2)} \rightarrow \texttt{bool()}  \\
  \texttt{[page 332]} Test string equality

- \texttt{concat(String1, String2)} \rightarrow \texttt{String3}  \\
  \texttt{[page 332]} Concatenate two strings

- \texttt{chr(String, Character)} \rightarrow \texttt{Index}  \\
  \texttt{[page 332]} Return the index of the first/last occurrence of Character in String

- \texttt{rchr(String, Character)} \rightarrow \texttt{Index}  \\
  \texttt{[page 332]} Return the index of the first/last occurrence of Character in String

- \texttt{str(String, SubString)} \rightarrow \texttt{Index}  \\
  \texttt{[page 332]} Find the index of a substring
- `rstr(String, SubString) -> Index` [page 332] Find the index of a substring
- `span(String, Chars) -> Length` [page 333] Span characters at start of string
- `cspan(String, Chars) -> Length` [page 333] Span characters at start of string
- `substr(String, Start) -> SubString` [page 333] Return a substring of String
- `substr(String, Start, Length) -> Substring` [page 333] Return a substring of String
- `tokens(String, SeparatorList) -> Tokens` [page 333] Split string into tokens
- `chars(Character, Number) -> String` [page 333] Returns a string consisting of numbers of characters
- `chars(Character, Number, Tail) -> String` [page 333] Returns a string consisting of numbers of characters
- `copies(String, Number) -> Copies` [page 334] Copy a string
- `words(String) -> Count` [page 334] Count blank separated words
- `words(String, Character) -> Count` [page 334] Count blank separated words
- `sub_string(String, Start) -> SubString` [page 335] Extract a substring
- `left(String, Number) -> Left` [page 335] Adjust left end of string
- `left(String, Number, Character) -> Left` [page 335] Adjust left end of string
- `right(String, Number) -> Right` [page 335] Adjust right end of string
- `right(String, Number, Character) -> Right` [page 335] Adjust right end of string
- `centre(String, Number) -> Centered` [page 335] Center a string
- `centre(String, Number, Character) -> Centered` [page 335] Center a string
- `sub_string(String, Start) -> SubString` [page 335] Extract a substring
• `sub_string(String, Start, Stop) -> SubString`  
  [page 336] Extract a substring

• `to_float(String) -> {Float,Rest} | {error,Reason}`  
  [page 336] Returns a float whose text representation is the integers (ASCII values) in String.

• `to_integer(String) -> {Int,Rest} | {error,Reason}`  
  [page 336] Returns an integer whose text representation is the integers (ASCII values) in String.

• `to_lower(String) -> Result`  
  [page 337] Convert case of string (ISO/IEC 8859-1)

• `to_lower(Char) -> CharResult`  
  [page 337] Convert case of string (ISO/IEC 8859-1)

• `to_upper(String) -> Result`  
  [page 337] Convert case of string (ISO/IEC 8859-1)

• `to_upper(Char) -> CharResult`  
  [page 337] Convert case of string (ISO/IEC 8859-1)

**supervisor**

The following functions are exported:

• `start_link(Module, Args) -> Result`  
  [page 340] Create a supervisor process.

• `start_link(SupName, Module, Args) -> Result`  
  [page 340] Create a supervisor process.

• `start_child(SupRef, ChildSpec) -> Result`  
  [page 341] Dynamically add a child process to a supervisor.

• `terminate_child(SupRef, Id) -> Result`  
  [page 341] Terminate a child process belonging to a supervisor.

• `delete_child(SupRef, Id) -> Result`  
  [page 342] Delete a child specification from a supervisor.

• `restart_child(SupRef, Id) -> Result`  
  [page 342] Restart a terminated child process belonging to a supervisor.

• `which_children(SupRef) -> [{Id,Child,Type,Modules}]`  
  [page 343] Return information about all children specifications and child processes belonging to a supervisor.

• `check_childspecs([ChildSpec]) -> Result`  
  [page 343] Check if child specifications are syntactically correct.

• `Module:init(Arg) -> Result`  
  [page 344] Return a supervisor specification.

**supervisor_bridge**

The following functions are exported:

• `start_link(Module, Args) -> Result`  
  [page 345] Create a supervisor bridge process.
• `start_link(SupBridgeName, Module, Args) -> Result`
  [page 345] Create a supervisor bridge process

• `Module:init(Args) -> Result`
  [page 346] Initialize process and start subsystem.

• `Module:terminate(Reason, State)`
  [page 346] Clean up and stop subsystem.

sys

The following functions are exported:

• `log(Name, Flag)`
  [page 349] Log system events in memory

• `log(Name, Flag, Timeout) -> ok | {ok, [system_event()]}`
  [page 349] Log system events in memory

• `log_to_file(Name, Flag)`
  [page 349] Log system events to the specified file

• `log_to_file(Name, Flag, Timeout) -> ok | {error, open_file}`
  [page 349] Log system events to the specified file

• `statistics(Name, Flag)`
  [page 349] Enable or disable the collections of statistics

• `statistics(Name, Flag, Timeout) -> ok | {ok, Statistics}`
  [page 349] Enable or disable the collections of statistics

• `trace(Name, Flag)`
  [page 350] Print all system events on standard io

• `trace(Name, Flag, Timeout) -> void()`
  [page 350] Print all system events on standard io

• `no_debug(Name)`
  [page 350] Turn off debugging

• `no_debug(Name, Timeout) -> void()`
  [page 350] Turn off debugging

• `suspend(Name)`
  [page 350] Suspend the process

• `suspend(Name, Timeout) -> void()`
  [page 350] Suspend the process

• `resume(Name)`
  [page 350] Resume a suspended process

• `resume(Name, Timeout) -> void()`
  [page 350] Resume a suspended process

• `change_code(Name, Module, OldVsn, Extra)`
  [page 350] Send the code change system message to the process

• `change_code(Name, Module, OldVsn, Extra, Timeout) -> ok | {error, Reason}`
  [page 350] Send the code change system message to the process

• `get_status(Name)`
  [page 350] Get the status of the process
• get_status(Name, Timeout) -> {status, Pid, [module, Mod], [PDict, SysState, Parent, Dbg, Misc]}
  [page 350] Get the status of the process
• install(Name, [Func, FuncState])
  [page 351] Install a debug function in the process
• install(Name, [Func, FuncState], Timeout)
  [page 351] Install a debug function in the process
• remove(Name, Func)
  [page 351] Remove a debug function from the process
• remove(Name, Func, Timeout) -> void()
  [page 351] Remove a debug function from the process
• debug_options(Options) -> [dbg_opt()]
  [page 352] Convert a list of options to a debug structure
• get_debug(Item, Debug, Default) -> term()
  [page 352] Get the data associated with a debug option
• handle_debug([dbg_opt()], FormFunc, Extra, Event) -> [dbg_opt()]
  [page 352] Generate a system event
• handle_system_msg(Msg, From, Parent, Module, Debug, Misc)
  [page 352] Take care of system messages
• print_log(Debug) -> void()
  [page 353] Print the logged events in the debug structure
• Mod:system_continue(Parent, Debug, Misc)
  [page 353] Called when the process should continue its execution
• Mod:system_terminate(Reason, Parent, Debug, Misc)
  [page 353] Called when the process should terminate
• Mod:system_code_change(Misc, Module, OldVsn, Extra) -> {ok, NMisc}
  [page 353] Called when the process should perform a code change

timer

The following functions are exported:

• start() -> ok
  [page 355] 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 355] Send Message to Pid after a specified Time.
• send_after(Time, Message) -> {ok, TRef} | {error, Reason}
  [page 355] Send Message to Pid after a specified Time.
• exit_after(Time, Pid, Reason1) -> {ok, TRef} | {error, Reason2}
  [page 356] Send an exit signal with Reason after a specified Time.
• exit_after(Time, Reason1) -> {ok, TRef} | {error, Reason2}
  [page 356] Send an exit signal with Reason after a specified Time.
• kill_after(Time, Pid) -> {ok, TRef} | {error, Reason2}
  [page 356] Send an exit signal with Reason after a specified Time.
- `kill_after(Time)` -> `{ok, TRef} | {error, Reason2}
  [page 356] 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 356] Send Message repeatedly at intervals of Time.
- `send_interval(Time, Message)` -> `{ok, TRef} | {error, Reason}
  [page 356] Send Message repeatedly at intervals of Time.
- `cancel(TRef)` -> `{ok, cancel} | {error, Reason}
  [page 356] Cancel a previously requested timeout identified by TRef.
- `sleep(Time)` -> ok
  [page 356] Suspend the calling process for Time amount of milliseconds.
- `tc(Module, Function, Arguments)` -> `{Time, Value}
  [page 357] Measure the real time it takes to evaluate apply(Module, Function, Arguments)
- `now_diff(T2, T1)` -> Tdiff
  [page 357] Calculate time difference between now/0 timestamps
- `seconds(Seconds)` -> Milliseconds
  [page 357] Convert Seconds to Milliseconds.
- `minutes(Minutes)` -> Milliseconds
  [page 357] Converts Minutes to Milliseconds.
- `hours(Hours)` -> Milliseconds
  [page 357] Convert Hours to Milliseconds.
- `hms(Hours, Minutes, Seconds)` -> Milliseconds
  [page 357] Convert Hours+Minutes+Seconds to Milliseconds.

### win32reg

The following functions are exported:

- `change_key(RegHandle, Key)` -> ReturnValue
  [page 360] Move to a key in the registry
- `change_key_create(RegHandle, Key)` -> ReturnValue
  [page 360] Move to a key, create it if it is not there
- `close(RegHandle)` -> ReturnValue
  [page 360] Close the registry.
- `current_key(RegHandle)` -> ReturnValue
  [page 360] Return the path to the current key.
- `delete_key(RegHandle)` -> ReturnValue
  [page 360] Delete the current key
- `delete_value(RegHandle, Name)` -> ReturnValue
  [page 361] Delete the named value on the current key.
- `expand(String)` -> ExpandedString
  [page 361] Expand a string with environment variables
• `format_error(ErrorId) -> ErrorString`
  [page 361] Convert an POSIX errorcode to a string

• `open(OpenModeList) -> ReturnValue`
  [page 361] Open the registry for reading or writing

• `set_value(RegHandle, Name, Value) -> ReturnValue`
  [page 361] Set value at the current registry key with specified name.

• `sub_keys(RegHandle) -> ReturnValue`
  [page 362] Get subkeys to the current key.

• `value(RegHandle, Name) -> ReturnValue`
  [page 362] Get the named value on the current key.

• `values(RegHandle) -> ReturnValue`
  [page 362] Get all values on the current key.

```
zip
```

The following functions are exported:

• `zip(Name, FileList) -> RetValue`
  [page 364] Create a zip archive with options

• `zip(Name, FileList, Options) -> RetValue`
  [page 364] Create a zip archive with options

• `create(Name, FileList) -> RetValue`
  [page 364] Create a zip archive with options

• `create(Name, FileList, Options) -> RetValue`
  [page 364] Create a zip archive with options

• `unzip(Archive) -> RetValue`
  [page 365] Extract files from a zip archive

• `unzip(Archive, Options) -> RetValue`
  [page 365] Extract files from a zip archive

• `extract(Archive) -> RetValue`
  [page 365] Extract files from a zip archive

• `extract(Archive, Options) -> RetValue`
  [page 365] Extract files from a zip archive

• `list_dir(Archive) -> RetValue`
  [page 366] Retrieve the name of all files in a zip archive

• `list_dir(Archive, Options)`
  [page 366] Retrieve the name of all files in a zip archive

• `table(Archive) -> RetValue`
  [page 366] Retrieve the name of all files in a zip archive

• `table(Archive, Options)`
  [page 366] Retrieve the name of all files in a zip archive

• `t(Archive)`
  [page 366] Print the name of each file in a zip archive

• `tt(Archive)`
  [page 366] Print name and information for each file in a zip archive

• `zip_open(Archive) -> {ok, ZipHandle} | {error, Reason}`
  [page 366] Open an archive and return a handle to it

```
zip_open(Archive, Options) -> {ok, ZipHandle} | {error, Reason}
[page 367] Open an archive and return a handle to it

zip_list_dir(ZipHandle) -> Result | {error, Reason}
[page 367] Return a table of files in open zip archive

zip_get(ZipHandle) -> {ok, Result} | {error, Reason}
[page 367] Extract files from an open archive

zip_get(FileName, ZipHandle) -> {ok, Result} | {error, Reason}
[page 367] Extract files from an open archive

zip_close(ZipHandle) -> ok | {error, EINVAL}
[page 367] Close an open archive
STDLIB

Application

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

Configuration

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

shell_esc = icl | abort This parameter can be used to alter the behaviour of the Erlang shell when ^G is pressed.
restricted_shell = module() This parameter can be used to run the Erlang shell in restricted mode.
shell_history_length = integer() >= 0 This parameter can be used to determine how many commands are saved by the Erlang shell.
shell_saved_results = integer() >= 0 This parameter can be used to determine how many results are saved by the Erlang shell.

See Also

[app(4)], [application(3)], shell(3) [page 296],
base64

Erlang Module

Implements base 64 encode and decode, see RFC 2045.

Exports

code(Data) -> Base64
code_to_string(Data) -> Base64String

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

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

decode(Base64) -> Data
decode_to_string(Base64) -> DataString
mime_decode(Base64) -> Data
mime_decode_string(Base64) -> DataString

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

Decodes a base64 encoded string to plain ASCII. The string should only consist of characters in the base64 set, see RFC 4648, mime_decode/1 and mime_decode_to_string/1 strips away illegal characters, while decode/1 and decode_to_string/1 fails if an illegal character is found.
beam_lib

Erlang Module

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

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

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

Debug Information/Abstract Code

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

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

The debug information can also be removed from BEAM files using strip/1 [page 59], strip_files/1 [page 59] and/or strip_release/1 [page 60].

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

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

Encrypted debug information

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

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

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

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

There are two ways to provide the key:

1. Use the compiler option {debug_info,Key}, see [compile(3)], and the function crypto_keyfun/1 [page 60] to register a fun which returns the key whenever beam_lib needs to decrypt the debug information.
   If no such fun is registered, beam_lib will instead search for a .erlang.crypt file, see below.
2. Store the key in a text file named .erlang.crypt.
   In this case, the compiler option encrypt_debug_info can be used, see [compile(3)].

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

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

   {debug_info, Mode, Module, Key}

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

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

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

   [{debug_info, des3_cbc, [], "%>7}pc/DM6Goa*68$Mw|l#&Gejr]G~"}].

And here is a slightly more complicated example of an .erlang.crypt which provides one key for the module t, and another key for all other modules:
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
chunkref() = chunkname() | chunkid()

Exports

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

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

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

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

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

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

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

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

Types:
- Beam = beam()
- Module = atom()
- Version = term()
- Reason - see chunks/2

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

1> beam_lib:version(a).
   \{ok,\{a,\[1\]\}\}
2> beam_lib:version(b).
   \{ok,\{b,\[1\]\}\}
3> beam_lib:version(c).
   \{ok,\{c,\[1,2\]\}\}
4> beam_lib:version(d).
   \{ok,\{d,\[275613208176997377698094100858909383631\]\}\}

\[md5(\text{Beam}) \rightarrow \{\text{ok, \{Module, MD5\}} | \text{error, beam_lib, Reason}\}\]

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

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

\[\text{info}(\text{Beam}) \rightarrow [\{\text{Item, Info}\}] | \{\text{error, beam_lib, Reason1}\}\]

Types:
- Beam = beam()
- Item, Info - see below
- Reason1 = \{\text{chunk\_too\_big}, Filename, ChunkId, ChunkSize, FileSize\}
- | \{\text{invalid\_beam\_file}, Filename, Pos\}
- | \{\text{invalid\_chunk}, Filename, ChunkId\}
- | \{\text{missing\_chunk}, Filename, ChunkId\}
- | \{\text{not\_a\_beam\_file}, Filename\}
- | \{\text{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.
For each chunk, the identifier (string) and the position and size of the chunk data, in bytes.

\[
\text{cmp}(\text{Beam}_1, \text{Beam}_2) \rightarrow \text{ok} | \{\text{error, beam_lib, Reason}\}
\]

**Types:**
- \(\text{Beam}_1 = \text{Beam}_2 = \text{beam()}\)
- \(\text{Reason} = \{\text{modules different, Module}_1, \text{Module}_2\}\)
- \(\{\text{chunks different, ChunkId}\}\)
- \(\{\text{Reason} - \text{see info/1}\}\)
- \(\text{Module}_1 = \text{Module}_2 = \text{atom()}\)
- \(\text{ChunkId} = \text{chunkid()}\)

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, \(\text{ok}\) is returned. Otherwise an error message is returned.

\[
\text{cmp_dirs}(\text{Dir}_1, \text{Dir}_2) \rightarrow \{\text{Only}_1, \text{Only}_2, \text{Different}\} | \{\text{error, beam_lib, Reason}_1\}
\]

**Types:**
- \(\text{Dir}_1 = \text{Dir}_2 = \text{string()} | \text{atom()}\)
- \(\text{Different} = \{\text{Filename}_1, \text{Filename}_2\}\)
- \(\text{Only}_1 = \text{Only}_2 = \{\text{Filename}\}\)
- \(\text{Filename} = \text{Filename}_1 = \text{Filename}_2 = \text{string()}\)
- \(\text{Reason}_1 - \text{see info/1}\)

The \(\text{cmp_dirs/2}\) function compares the BEAM files in two directories. Only files with extension ".beam" are compared. BEAM files that exist in directory \(\text{Dir}_1\) (\(\text{Dir}_2\)) only are returned in \(\text{Only}_1\) (\(\text{Only}_2\)). BEAM files that exist on both directories but are considered different by \(\text{cmp/2}\) are returned as pairs \{\text{Filename}_1, \text{Filename}_2\} where \(\text{Filename}_1\) (\(\text{Filename}_2\)) exists in directory \(\text{Dir}_1\) (\(\text{Dir}_2\)).

\[
\text{diff_dirs}(\text{Dir}_1, \text{Dir}_2) \rightarrow \text{ok} | \{\text{error, beam_lib, Reason}_1\}
\]

**Types:**
- \(\text{Dir}_1 = \text{Dir}_2 = \text{string()} | \text{atom()}\)
- \(\text{Reason}_1 - \text{see info/1}\)

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

\[
\text{strip}(\text{Beam}_1) \rightarrow \{\text{ok, Module, Beam}_2\} | \{\text{error, beam_lib, Reason}_1\}
\]

**Types:**
- \(\text{Beam}_1 = \text{Beam}_2 = \text{beam()}\)
- \(\text{Module} = \text{atom()}\)
- \(\text{Reason}_1 - \text{see info/1}\)

The \(\text{strip/1}\) function removes all chunks from a BEAM file except those needed by the loader. In particular, the debug information (abstract code chunk) is removed.

\[
\text{strip_files}(\text{Files}) \rightarrow \{\text{ok, [[Module, Beam}_2]]\} | \{\text{error, beam_lib, Reason}_1\}
\]
The `strip_files/1` function removes all chunks except those needed by the loader from BEAM files. In particular, the debug information (abstract code chunk) is removed. The returned list contains one element for each given file name, in the same order as in `Files`.

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

Types:
- `Dir` = `string()` \| `atom()`
- `Module` = `atom()`
- `Filename` = `string()`
- `Reason1` - see `info/1`

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

```
format_error(Reason) -> Chars
```

Types:
- `Reason` - see other functions
- `Chars` = `[char() \| Chars]`

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

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

Types:
- `CryptoKeyFun` = `fun()` - see below
- `Reason` = `badfun` \| `exists` \| `term()`

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

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

The fun must handle the following arguments:

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

\[\text{CryptoKeyFun}(\{\text{debug_info, Mode, Module, Filename}\}) \rightarrow \text{Key}\]

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

\[\text{CryptoKeyFun}(\text{clear}) \rightarrow \text{term()}\]

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

\[\text{clear_crypto_key_fun()} \rightarrow \{\text{ok, Result}\}\]

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.
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`
  - **Types:**
    - File = name() – see filename(3)
  - `c/1,2` compiles and then purges and loads the code for a file. `Options` defaults to `[]`. Compilation is equivalent to:
    ```erlang```
    ```erlang```compile:file(File, Options ++ [report_errors, report_warnings])
    ```erlang```
  - 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.

- `c(File, Options) -> {ok, Module} | error`
  - **Types:**
    - Options = [Opt] – see compile:file/2

- `cd(Dir) -> void()`
  - **Types:**
    - Dir = name() – see filename(3)
  - Changes working directory to `Dir`, which may be a relative name, and then prints the name of the new working directory.
2> cd("../erlang").
/home/ron/erlang

flush() -> void()
Flushes any messages sent to the shell.

help() -> void()
Displays help information: all valid shell internal commands, and commands in this module.

i() -> void()
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.

ni() -> void()
i/0 displays information about the system, listing information about all processes. ni/0 does the same, but for all nodes the network.

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.

c(Files) -> ok
Types:
• Files = [File]
• File = name() – see filename(3)
Compiles a list of files by calling compile:file(File, [report_errors, report_warnings]) for each File in Files.

ls() -> void()
Lists files in the current directory.

ls(Dir) -> void()
Types:
• Dir = name() – see filename(3)
Lists files in directory Dir.

\texttt{m() \rightarrow \text{void()}}

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

\texttt{m(Module) \rightarrow \text{void()}}

Types:
- Module = \text{atom()}
Displays information about Module.

\texttt{memory() \rightarrow \{\text{[Type, Size]}\}}

Types:
- Type, Size - see \texttt{erlang:memory/0}
Memory allocation information. Equivalent to \texttt{erlang:memory/0}.

\texttt{memory(Type) \rightarrow \text{Size}}

\texttt{memory([Type]) \rightarrow \{\text{[Type, Size]}\}}

Types:
- Type, Size - see \texttt{erlang:memory/0}
Memory allocation information. Equivalent to \texttt{erlang:memory/1}.

\texttt{nc(File) \rightarrow \{\text{ok, Module}\} | \text{error}}

\texttt{nc(File, Options) \rightarrow \{\text{ok, Module}\} | \text{error}}

Types:
- File = \text{name()} - see \texttt{filename(3)}
- Options = [Opt] - see \texttt{compile:file/2}
Compiles and then loads the code for a file on all nodes. Options defaults to \texttt{[]}.
Compilation is equivalent to:
\texttt{compile:file(File, Opts ++ [report_errors, report_warnings])}

\texttt{nl(Module) \rightarrow \text{void()}}

Types:
- Module = \text{atom()}
Loads Module on all nodes.

\texttt{pid(X, Y, Z) \rightarrow \text{pid()}}

Types:
- X = Y = Z = \text{int()}
Converts X, Y, Z to the \text{pid} \langle X.Y.Z\rangle. This function should only be used when debugging.
pwd() -> void()

Prints the name of the working directory.

q() -> void()

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

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

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

xm(ModSpec) -> void()

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

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

y(File) -> YeccRet

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

Generates an LALR-1 parser. Equivalent to:
yecc:file(File)

y(File, Options) -> YeccRet

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

Generates an LALR-1 parser. Equivalent to:
yecc:file(File, Options)

See Also

[compile(3)], filename(3) [page 156], [erlang(3)], [yecc(3)], [xref(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 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.

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

Types:
- `Date = date()`
- `Time = time()`

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

```erlang
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.

```erlang
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.
- `[DateTimeUTC, 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.

```erlang
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.
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.

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

Types:
- `Name = name()`

Closes a table. Only processes that have opened a table are allowed to close it. All open tables must be closed before the system is stopped. If an attempt is made to open a table which has not been properly closed, Dets automatically tries to repair the table.

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

Types:
- `Name = name()`
- `Key` = any

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

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

Types:
- `Name = name()`

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

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

Types:
- `Name = name()`
- `Object = object()`

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

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

Types:
• Key = term()
• Name = name()

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

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

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

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

foldl(Function, Acc0, Name) -> Acc1 | {error, Reason}
Types:
• Function = fun(Object, AccIn) -> AccOut
• Acc0 = Acc1 = AccIn = AccOut = term()
• Name = name()
• Object = object()

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

foldr(Function, Acc0, Name) -> Acc1 | {error, Reason}
Types:
• Function = fun(Object, AccIn) -> AccOut
• Acc0 = Acc1 = AccIn = AccOut = term()
• Name = name()
• Object = object()

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

from_ets(Name, EtsTab) -> ok | {error, Reason}
Types:
• Name = name()
• EtsTab = -see ets(3)-

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

info(Name) -> InfoList | undefined
Types:
- Name = name()
- InfoList = [{Item, Value}]

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

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

info(Name, Item) -> Value | undefined

Types:
- Name = name()

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

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

init_table(Name, InitFun [, Options]) -> ok | {error, Reason}
Types:
- Name = atom()
- InitFun = fun(Args) -> Res
- Args = read | close
- Res = end_of_input | [{object()}, InitFun] | {Data, InitFun} | term()
- Data = binary() | tuple()

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

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

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

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

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

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

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

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

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

```
insert_new(Name, Objects) -> Bool
```
Types:
- Name = name()
- Objects = object() ! [object()]
- Bool = bool()

Inserts one or more objects into the table Name. If there already exists 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.

```
is_compatible_bchunk_format(Name, BchunkFormat) -> Bool
Types:
  - Name = name()
  - BchunkFormat = binary()
  - Bool = bool()

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

```
is_dets_file(FileName) -> Bool | [error, Reason]
Types:
  - FileName = file()
  - Bool = bool()

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

```
lookup(Name, Key) -> [Object] | {error, Reason}
Types:
  - Key = term()
  - Name = name()
  - Object = object()

Returns a list of all objects with the key Key stored in the table Name. For example:
```
2> dets:open_file(abc, [{type, bag}]).
   {ok,abc}
3> dets:insert(abc, [1,2,3]).
   ok
4> dets:insert(abc, [1,3,4]).
   ok
5> dets:lookup(abc, 1).[[1,2,3],[1,3,4]]
```

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

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

```
match(Continuation) -> [[Match], Continuation2] | '$end_of_table' | {error, Reason}
Types:
```
M et Ch o o n 2 = bindings_cont() 

M ath = [term()] 

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

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

\[
\text{match}(\text{Name}, \text{Pattern}) \rightarrow \text{[Match]} \mid \{\text{error, Reason}\}
\]

Types:
- \(\text{Name} = \text{name}()\)
- \(\text{Pattern} = \text{tuple}()\)
- \(\text{Match} = [\text{term}()]\)

Returns for each object of the table \text{Name} that matches \text{Pattern} a list of bindings in 
some unspecified order. See ets(3) [page 128] 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.

\[
\text{match}(\text{Name}, \text{Pattern}, N) \rightarrow \{\text{[Match], Continuation}\} \mid '$end_of_table' \mid \{\text{error, Reason}\}
\]

Types:
- \(\text{Name} = \text{name}()\)
- \(\text{Pattern} = \text{tuple}()\)
- \(N = \text{default} \mid \text{int}()\)
- \(\text{Match} = [\text{term}()]\)
- \(\text{Continuation} = \text{bindings_cont}()\)

Matches some or all objects of the table \text{Name} and returns a non-empty list of the 
bindings that match \text{Pattern} in some unspecified order. See ets(3) [page 128] for a 
description of patterns.

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

If the keypos'th element of Pattern is bound, all objects of the table are matched. If 
the keypos'th element is unbound, all objects of the table are matched, \(N\) objects at a 
time, until at least one object matches or the end of the table has been reached. The 
default, indicated by giving \(N\) the value default, is to let the number of objects vary 
depending on the sizes of the objects. If \text{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.

\[
\text{match_delete}(\text{Name}, \text{Pattern}) \rightarrow N \mid \{\text{error, Reason}\}
\]

Types:
- \(\text{Name} = \text{name}()\)
- \(N = \text{int}()\)
Pattern = tuple()

Deletes all objects that match Pattern from the table Name, and returns the number of deleted objects. See ets(3) [page 128] for a description of patterns.

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

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

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

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

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

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

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

Returns a list of all objects of the table Name that match Pattern in some unspecified order. See ets(3) [page 128] for a description of patterns.

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

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

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

Types:
- Name = name()
- Pattern = tuple()
- N = default | int()
- Object = object()
- Continuation = object_cont()
Matches some or all objects stored in the table `Name` and returns a non-empty list of the objects that match `Pattern` in some unspecified order. See ets(3) [page 128] for a description of patterns.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Types:
- Name = atom()
Opens a table. An empty Dets table is created if no file exists.

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

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

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

- \( \{ \text{access}, \text{access()} \} \). 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.

- \( \{ \text{auto} \_\text{save}, \text{auto} \_\text{save()} \} \), 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).

- \( \{ \text{estimated} \_\text{no} \_\text{objects}, \text{int()} \} \). Equivalent to the min_no_slots option.

- \( \{ \text{file}, \text{file()} \} \), the name of the file to be opened. The default value is the name of the table.

- \( \{ \text{max} \_\text{no} \_\text{slots}, \text{no} \_\text{slots()} \} \), 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.

- \( \{ \text{min} \_\text{no} \_\text{slots}, \text{no} \_\text{slots()} \} \). 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.

- \( \{ \text{keypos}, \text{keypos()} \} \), 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.

- \( \{ \text{ram} \_\text{file}, \text{bool()} \} \), 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.

- \( \{ \text{repair}, \text{Value} \} \). 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 \( \{ \text{error}, \text{need} \_\text{repair} \} \) 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}.

```erlang
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.

```erlang
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.

```erlang
safe_fixtable(Name, Fix)
```

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

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

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

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

select(Continuation) -> \{Selection, Continuation2\} | 'end of table' | \{error, Reason\}

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

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

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

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

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

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

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

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

select(Name, MatchSpec, N) -> \{Selection, Continuation\} | 'end of table' | \{error, Reason\}

Types:
- Name = name()
- MatchSpec = match_spec()
- N = default | int()
- Selection = [term()]
- Continuation = select_cont()
Returns the results of applying the match specification \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 \texttt{'$end_of_table'} is returned. The continuation is to be used when matching further objects by calling \texttt{select/1}.

If the 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 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, until at least one object matches or the end of the table has been reached. 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}

Types:
\begin{itemize}
  \item \texttt{Name} = name()
  \item \texttt{MatchSpec} = match_spec()
  \item \texttt{N} = int()
\end{itemize}

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

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

Types:
\begin{itemize}
  \item \texttt{Name} = name()
  \item \texttt{I} = int()
  \item \texttt{Object} = object()
\end{itemize}

The objects of a table are distributed among slots, starting with slot \texttt{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} \texttt{'$end_of_table'} is returned.
\end{verbatim}

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

Types:
\begin{itemize}
  \item \texttt{Name} = name()
\end{itemize}

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

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

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

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

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

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

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

An example with implicit match specification:

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

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

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

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

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

Types:
- \texttt{Name = name()}
- \texttt{EtsTab = \texttt{-see ets(3)-}}

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

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

Types:
- \texttt{Fun = fun(Object) -> FunReturn}
- \texttt{FunReturn = continue | \{continue, Val\} | \{done, Value\}}
- \texttt{Val = Value = term()}
- \texttt{Name = name()}
- \texttt{Object = object()}
- \texttt{Return = \{term()\}}

Applies \texttt{Fun} to each object stored in the table \texttt{Name} in some unspecified order. Different actions are taken depending on the return value of \texttt{Fun}. The following \texttt{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:

  \texttt{fun(X) -> io:format("-p-n", [X]), continue end.}

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

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

- **\{done, Value\}** Terminate the traversal and return \texttt{[Value | Acc]}.

Any other value returned by \texttt{Fun} terminates the traversal and is immediately returned.

\texttt{update_counter(Name, Key, Increment) -> Result}

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

Updates the object with key \texttt{Key} stored in the table \texttt{Name} of type \texttt{set} by adding \texttt{Incr} to the element at the \texttt{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.

\textbf{See Also}

ets(3) [page 128], mnesia(3), qlc(3) [page 265]
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.

appendList(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.

```erlang
fetch(Dict, Key) ->
    Value = fetch(Dict, Key, term()),
    {ok, Value}.
```

### fetch_keys

```erlang
fetch_keys(Dict) ->
    Keys = fetch_keys(Dict, [], term()),
    [{Key, Value} || {Key, Value} <- Keys].
```

### map

```erlang
map(Fun, Dict) ->
    Map = map(Fun, Dict, term()),
    [{Key, Value} || {Key, Value} <- Map].
```

### keys

```erlang
keys(Dict) ->
    Keys = fetch_keys(Dict, [], term()),
    [{Key, Value} || {Key, Value} <- Keys].
```

### fold

```erlang
fold(Fun, Acc0, Dict) ->
    Fold = fold(Fun, Acc0, Dict, term()),
    [{Key, Value} || {Key, Value} <- Fold].
```

### from_list

```erlang
from_list(List) ->
    FromList = from_list(List, term()),
    [{Key, Value} || {Key, Value} <- FromList].
```

### is_key

```erlang
is_key(Key, Dict) ->
    IsKey = is_key(Key, Dict, term()),
    IsKey.
```

---

**fetch_keys**

```erlang
fetch_keys(Dict) ->
    Keys = fetch_keys(Dict, [], term()),
    [{Key, Value} || {Key, Value} <- Keys].
```

**filter**

```erlang
filter(Pred, Dict) ->
    Filter = filter(Pred, Dict, term()),
    [{Key, Value} || {Key, Value} <- Filter].
```

**find**

```erlang
find(Key, Dict) ->
    Find = find(Key, Dict, term()),
    Find.
```

**fold**

```erlang
fold(Fun, Acc0, Dict) ->
    Fold = fold(Fun, Acc0, Dict, term()),
    Fold.
```

**from_list**

```erlang
from_list(List) ->
    FromList = from_list(List, term()),
    FromList.
```

**is_key**

```erlang
is_key(Key, Dict) ->
    IsKey = is_key(Key, Dict, term()),
    IsKey.
```
Types:
- Key = term()
- Dict = dictionary()

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

\[
\text{map}(\text{Fun}, \text{Dict1}) \rightarrow \text{Dict2}
\]

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

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

\[
\text{merge}(\text{Fun}, \text{Dict1}, \text{Dict2}) \rightarrow \text{Dict3}
\]

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:

\[
\text{merge}(\text{Fun}, \text{D1}, \text{D2}) \rightarrow \\
\text{fold}(\text{fun} \ (\text{K}, \text{V1}, \text{D}) \rightarrow \\
\quad \text{update}(\text{K}, \text{fun} \ (\text{V2}) \rightarrow \text{Fun}(\text{K}, \text{V1}, \text{V2}) \text{ end}, \text{V1}, \text{D}) \\
\quad \text{end}, \text{D2}, \text{D1}).
\]

but is faster.

\[
\text{new()} \rightarrow \text{dictionary()}
\]

This function creates a new dictionary.

\[
\text{store}(\text{Key}, \text{Value}, \text{Dict1}) \rightarrow \text{Dict2}
\]

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.

\[
\text{toList}(\text{Dict}) \rightarrow \text{List}
\]

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:

```erlang
> 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 168], `orddict(3)` [page 249]
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 is said to be an out-neighbour of \(v\), and \(v\) is said to be an in-neighbour of \(w\). A path \(P\) from \(v[1]\) to \(v[k]\) in a digraph \((V,E)\) is a non-empty sequence \(v[1],v[2],...,v[k]\) of vertices in \(V\) such that there is an edge \((v[i],v[i+1])\) in \(E\) for \(1 \leq i \leq k-1\). 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

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

Types:
- \(G = \text{digraph}()\)
- \(E = \text{edge}()\)
- \(V1 = V2 = \text{vertex}()\)
- \(Label = \text{label}()\)
- \(Reason = \{\text{bad_edge}, \text{Path}\} | \{\text{bad_vertex}, V\}\)
- \(Path = [\text{vertex}()]\)
add_edge/5 creates (or modifies) the edge E of the digraph G, using Label as the (new) label [page 92] of the edge. The edge is emanating [page 92] from V1 and incident [page 92] on V2. Returns E.

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

add_edge(G, V1, V2) is equivalent to add_edge(G, V1, V2, []). If the edge would create a cycle in an acyclic digraph [page 92], then {error, {bad_edge, Path}} is returned. If either of V1 or V2 is not a vertex of the digraph G, then {error, {bad_vertex, V}} is returned, V = V1 or V = V2.

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

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

add_vertex/3 creates (or modifies) the vertex V of the digraph G, using Label as the (new) label [page 92] of the vertex. Returns V.

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

del_edge(G, E) -> true

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

Deletes the edge E from the digraph G.

del_edges(G, Edges) -> true

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

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

del_path(G, V1, V2) -> true

Types:
- G = digraph()
- V1 = V2 = vertex()
Deletes edges from the digraph \( G \) until there are no paths \([\text{page 92}]\) from the vertex \( V_1 \) to the vertex \( V_2 \).

A sketch of the procedure employed: Find an arbitrary simple path \([\text{page 92}]\) \( v[1], v[2], \ldots, v[k] \) from \( V_1 \) to \( V_2 \) in \( G \). Remove all edges of \( G \) emanating \([\text{page 92}]\) from \( v[i] \) and incident \([\text{page 92}]\) to \( v[i+1] \) for \( 1 \leq i < k \) (including multiple edges). Repeat until there is no path between \( V_1 \) and \( V_2 \).

del_vertex(G, V) -> true

Types:
- \( G \) = digraph()
- \( V \) = vertex()

Deletes the vertex \( V \) from the digraph \( G \). Any edges emanating \([\text{page 92}]\) from \( V \) or incident \([\text{page 92}]\) on \( V \) are also deleted.

del_vertices(G, Vertices) -> true

Types:
- \( G \) = digraph()
- \( \text{Vertices} = [\text{vertex()}] \)

Deletes the vertices in the list \( \text{Vertices} \) from the digraph \( G \).

delete(G) -> true

Types:
- \( G \) = digraph()

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

del_vertex(G, V) -> true

Types:
- \( G \) = digraph()

Deletes the vertex \( V \) from the digraph \( G \). Any edges emanating \([\text{page 92}]\) from \( V \) or incident \([\text{page 92}]\) on \( V \) are also deleted.

del_vertices(G, Vertices) -> true

Types:
- \( G \) = digraph()
- \( \text{Vertices} = [\text{vertex()}] \)

Deletes the vertices in the list \( \text{Vertices} \) from the digraph \( G \).

delete(G) -> true

Types:
- \( G \) = digraph()

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

del_vertex(G, V) -> true

Types:
- \( G \) = digraph()

Deletes the vertex \( V \) from the digraph \( G \). Any edges emanating \([\text{page 92}]\) from \( V \) or incident \([\text{page 92}]\) on \( V \) are also deleted.

del_vertices(G, Vertices) -> true

Types:
- \( G \) = digraph()
- \( \text{Vertices} = [\text{vertex()}] \)

Deletes the vertices in the list \( \text{Vertices} \) from the digraph \( G \).

delete(G) -> true

Types:
- \( G \) = digraph()

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

del_vertex(G, V) -> true

Types:
- \( G \) = digraph()

Deletes the vertex \( V \) from the digraph \( G \). Any edges emanating \([\text{page 92}]\) from \( V \) or incident \([\text{page 92}]\) on \( V \) are also deleted.

del_vertices(G, Vertices) -> true

Types:
- \( G \) = digraph()
- \( \text{Vertices} = [\text{vertex()}] \)

Deletes the vertices in the list \( \text{Vertices} \) from the digraph \( G \).

delete(G) -> true

Types:
- \( G \) = digraph()

Deletes the digraph \( G \). This call is important because digraphs are implemented with \( \text{Ets} \). There is no garbage collection of \( \text{Ets} \) tables. The digraph will, however, be deleted if the process that created the digraph terminates.
Types:
- \( G = \text{digraph()} \)
- \( V = \text{vertex()} \)
- \( \text{Edges} = [\text{edge()}] \)

Returns a list of all edges emanating [page 92] from or incident [page 92] on \( V \) of the digraph \( G \), in some unspecified order.

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

Types:
- \( G = \text{digraph()} \)
- \( V1 = V2 = \text{vertex()} \)
- \( \text{Vertices} = [\text{vertex()}] \)

If there is a simple cycle [page 92] of length two or more through the vertex \( V \), then the cycle is returned as a list \([V, \ldots, V]\) of vertices, otherwise if there is a loop [page 92] through \( V \), then the loop is returned as a list \([V]\). If there are no cycles through \( V \), then false is returned.

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

Types:
- \( G = \text{digraph()} \)
- \( V1 = V2 = \text{vertex()} \)
- \( \text{Vertices} = [\text{vertex()}] \)

Tries to find a simple path [page 92] from the vertex \( V1 \) to the vertex \( V2 \) of the digraph \( G \). Returns the path as a list \([V1, \ldots, V2]\) 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:
- \( G = \text{digraph()} \)
- \( V1 = V2 = \text{vertex()} \)
- \( \text{Vertices} = [\text{vertex()}] \)

Tries to find an as short as possible simple cycle [page 92] through the vertex \( V \) of the digraph \( G \). Returns the cycle as a list \([V, \ldots, V]\) of vertices, or false if no simple cycle through \( V \) exists. Note that a loop [page 92] through \( V \) is returned as the list \([V, V]\).

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

Types:
- \( G = \text{digraph()} \)
- \( V1 = V2 = \text{vertex()} \)
- \( \text{Vertices} = [\text{vertex()}] \)
Tries to find an as short as possible simple path [page 92] 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.

\[ \text{in\_degree}(G, V) \rightarrow \text{integer() } \]

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

Returns the in-degree [page 92] of the vertex $V$ of the digraph $G$.

\[ \text{in\_edges}(G, V) \rightarrow \text{Edges } \]

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

Returns a list of all edges incident [page 92] on $V$ of the digraph $G$, in some unspecified order.

\[ \text{in\_neighbours}(G, V) \rightarrow \text{Vertices } \]

Types:
- $G = \text{digraph()}$
- $V = \text{vertex()}$
- $\text{Vertices} = [\text{vertex()}]$  

Returns a list of all in-neighbours [page 92] of $V$ of the digraph $G$, in some unspecified order.

\[ \text{info}(G) \rightarrow \text{InfoList } \]

Types:
- $G = \text{digraph()}$
- $\text{InfoList} = [\{\text{cyclicity, Cyclicity}\}, \{\text{memory, NoWords}\}, \{\text{protection, Protection}\}]$
- $\text{Cyclicity} = \text{cyclic} \mid \text{acyclic}$
- $\text{Protection} = \text{protected} \mid \text{private}$
- $\text{NoWords} = \text{integer()} > 0$

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

- $\{\text{cyclicity, Cyclicity}\}$, where $\text{Cyclicity}$ is cyclic or acyclic, according to the options given to $\text{new}$.
- $\{\text{memory, NoWords}\}$, where $\text{NoWords}$ is the number of words allocated to the ets tables.
- $\{\text{protection, Protection}\}$, where $\text{Protection}$ is protected or private, according to the options given to $\text{new}$.
new() -> digraph()
   Equivalent to new([]).

new(Type) -> digraph() | {error, Reason}
   Types:
   • Type = [cyclic | acyclic | private | protected]
   • Reason = {unknown_type, term()}
   Returns an empty digraph [page 92] with properties according to the options in Type:
   
   cyclic  Allow cycles [page 92] in the digraph (default).
   acyclic The digraph is to be kept acyclic [page 92].
   protected Other processes can read the digraph (default).
   private  The digraph can be read and modified by the creating process only.

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

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

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

out_degree(G, V) -> integer()
   Types:
   • G = digraph()
   • V = vertex()
   Returns the out-degree [page 92] of the vertex V of the digraph G.

out_edges(G, V) -> Edges
   Types:
   • G = digraph()
   • V = vertex()
   • Edges = [edge()]
   Returns a list of all edges emanating [page 92] from V of the digraph G, in some unspecified order.

out_neighbours(G, V) -> Vertices
   Types:
   • G = digraph()
digraph

- \( V = \text{vertex()} \)
- \( \text{Vertices} = [\text{vertex()}] \)

Returns a list of all out-neighbours [page 92] 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 92] of the vertex \( V \) of the digraph \( G \), or 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
utils(3) [page 99], ets(3) [page 128]
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 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 \(1\leq i\leq k\). 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 99]. 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 99] of Digraph as returned by strong_components/. 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 99] from x and incident [page 99] 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 99] []

Each and every cycle [page 99] is included in some strongly connected component, which implies that there always exists a topological ordering [page 99] of the created digraph.

cyclic_strong_components(Digraph) -> [StrongComponent]

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

Returns a list of strongly connected components [page 99]. 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 99] 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 99].

loop_vertices(Digraph) -> Vertices

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

Returns a list of all vertices of Digraph that are included in some loop [page 99].
postorder(Digraph) \rightarrow Vertices

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

Returns all vertices of the digraph Digraph. The order is given by a depth-first traversal [page 99] 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) \rightarrow Vertices

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

Returns all vertices of the digraph Digraph. The order is given by a depth-first traversal [page 99] of the digraph, collecting visited vertices in pre-order.

reachable(Vertices, Digraph) \rightarrow 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 99] 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) \rightarrow 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 99] 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 99] are returned.

reaching(Vertices, Digraph) \rightarrow 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 99] 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) \rightarrow 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 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 are returned.

**strong_components(Digraph) -> [StrongComponent]**

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

Returns a list of strongly connected components. 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 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 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 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 that occur earlier in the list.

**See Also**

digraph(3) [page 92]
The Erlang code preprocessor includes functions which are used by compile to preprocess macros and include files before the actual parsing takes place.

Exports

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

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

Opens a file for preprocessing.

close(Epp) -> ok

Types:
- Epp = pid() - handle to the epp server

Closes the preprocessing of a file.

parse_erl_form(Epp) -> {ok, AbsForm} | {eof, Line} | {error, ErrorInfo}

Types:
- Epp = pid()
- AbsForm = term()
- Line = integer()
- ErrorInfo = see separate description below.

Returns the next Erlang form from the opened Erlang source file. The tuple {eof, Line} is returned at end-of-file. The first form corresponds to an implicit attribute -file(File,1), where File is the name of the file.

parse_file(FileName, IncludePath, PredefMacro) -> {ok, [Form]} | {error, OpenError}

Types:
- FileName = atom() | string()
Preprocesses and parses an Erlang source file. Note that the tuple `{eof, Line}` returned at end-of-file is included as a “form”.

**Error Information**

The `ErrorInfo` mentioned above is the standard `ErrorInfo` structure which is returned from all IO modules. It has the following format:

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

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

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

**See Also**

`erl_parse(3)` [page 114]
erl_eval

Erlang Module

This module provides an interpreter for Erlang expressions. The expressions are in the abstract syntax as returned by erl_parse, the Erlang parser, or a call to io:parse_erl_exprs/2.

Exports

exprs(Expressions, Bindings) -> {value, Value, NewBindings}
exprs(Expressions, Bindings, LocalFunctionHandler) -> {value, Value, NewBindings}
exprs(Expressions, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) ->
    {value, Value, NewBindings}

Types:
- Expressions = as returned by erl_parse or io:parse_erl_exprs/2
- Bindings = as returned by bindings/1
- LocalFunctionHandler = {value, Func} | {eval, Func} | none
- NonlocalFunctionHandler = {value, Func} | none

Evaluates Expressions with the set of bindings Bindings, where Expressions is a sequence of expressions (in abstract syntax) of a type which may be returned by io:parse_erl_exprs/2. See below for an explanation of how and when to use the arguments LocalFunctionHandler and NonlocalFunctionHandler.

Returns {value, Value, NewBindings}.

expr(Expression, Bindings) -> {value, Value, NewBindings}
expr(Expression, Bindings, LocalFunctionHandler) -> {value, Value, NewBindings}
expr(Expression, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) ->
    {value, Value, NewBindings}

Types:
- Expression = as returned by io:parse_erl_form/2, for example
- Bindings = as returned by bindings/1
- LocalFunctionHandler = {value, Func} | {eval, Func} | none
- NonlocalFunctionHandler = {value, Func} | none

Evaluates Expression with the set of bindings Bindings. Expression is an expression (in abstract syntax) of a type which may be returned by io:parse_erl_form/2. See below for an explanation of how and when to use the arguments LocalFunctionHandler and NonlocalFunctionHandler.

Returns {value, Value, NewBindings}.
expr_list(ExpressionList, Bindings) -> \{ValueList, NewBindings\}
expr_list(ExpressionList, Bindings, LocalFunctionHandler) -> \{ValueList, NewBindings\}
expr_list(ExpressionList, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> \{ValueList, NewBindings\}

Evaluates a list of expressions in parallel, using the same initial bindings for each expression. Attempts are made to merge the bindings returned from each evaluation. This function is useful in the LocalFunctionHandler. See below.
Returns \{ValueList, NewBindings\}.

new_bindings() -> BindingStruct
Returns an empty binding structure.

bindings(BindingStruct) -> Bindings
Returns the list of bindings contained in the binding structure.

binding(Name, BindingStruct) -> Binding
Returns the binding of Name in BindingStruct.

add_binding(Name, Value, Bindings) -> BindingStruct
Adds the binding Name = Value to Bindings. Returns an updated binding structure.

del_binding(Name, Bindings) -> BindingStruct
Removes the binding of Name in Bindings. Returns an updated binding structure.

Local Function Handler

During evaluation of a function, no calls can be made to local functions. An undefined function error would be generated. However, the optional argument LocalFunctionHandler may be used to define a function which is called when there is a call to a local function. The argument can have the following formats:

\{value,Func\} This defines a local function handler which is called with:

\texttt{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 \texttt{exit/1} with a suitable exit value.

\{eval,Func\} This defines a local function handler which is called with:

\texttt{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.

**Note:**

For calls such as erlang:apply(Fun, Args) or erlang:apply(Module, Function, Args) the call of the non-local function handler corresponding to the call to erlang:apply/2,3 itself- Func\{erlang, apply\}, [Fun, Args] or Func\{erlang, apply\}, [Module, Function, Args]\- will never take place. The non-local function handler will however be called with the evaluated arguments of the call to erlang:apply/2,3: Func(Fun, Args) or Func\{Module, Function\}, Args\ (assuming that \{Module, Function\} is not \{erlang, apply\}).

The nonlocal function handler argument is probably not used as frequently as the local function handler argument. A possible use is to call exit/1 on calls to functions that for some reason are not allowed to be called.

**Bugs**

The evaluator is not complete receive cannot be handled properly. Any undocumented functions in erl_eval should not be used.
erl expand records

Erlang Module

Exports

module(AbsForms, CompileOptions) -> AbsForms

Types:
  • AbsForms = [term()]
  • CompileOptions = [term()]

Expands all records in a module. The returned module has no references to records, neither attributes nor code.

See Also

The [abstract format] documentation in ERTS User's Guide
erl_id_trans

Erlang Module

This module performs an identity parse transformation of Erlang code. It is included as an example for users who may wish to write their own parse transformers. If the option \{parse_transform,Module\} is passed to the compiler, a user written function parse_transform/2 is called by the compiler before the code is checked for errors.

Exports

\texttt{parse_transform(Forms, Options) \rightarrow Forms}

Types:
- \texttt{Forms = [erlang_form()]}  
- \texttt{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 114], compile(3).
This module defines Erlang BIFs, guard tests and operators. This module is only of interest to programmers who manipulate Erlang code.

Exports

bif(Name, Arity) -> bool()
Types:
• Name = atom()
• Arity = integer()
Returns true if Name/Arity is an Erlang BIF which is automatically recognized by the compiler, otherwise false.

guard_bif(Name, Arity) -> bool()
Types:
• Name = atom()
• Arity = integer()
Returns true if Name/Arity is an Erlang BIF which is allowed in guards, otherwise false.

type_test(Name, Arity) -> bool()
Types:
• Name = atom()
• Arity = integer()
Returns true if Name/Arity is a valid Erlang type test, otherwise false.

arith_op(OpName, Arity) -> bool()
Types:
• OpName = atom()
• Arity = integer()
Returns true if OpName/Arity is an arithmetic operator, otherwise false.

bool_op(OpName, Arity) -> bool()
Types:
• OpName = atom()
Arity = integer()

Returns true if OpName/Arity is a Boolean operator, otherwise false.

comp_op(OpName, Arity) -> bool()

Types:
- OpName = atom()
- Arity = integer()

Returns true if OpName/Arity is a comparison operator, otherwise false.

list_op(OpName, Arity) -> bool()

Types:
- OpName = atom()
- Arity = integer()

Returns true if OpName/Arity is a list operator, otherwise false.

send_op(OpName, Arity) -> bool()

Types:
- OpName = atom()
- Arity = integer()

Returns true if OpName/Arity is a send operator, otherwise false.

op_type(OpName, Arity) -> Type

Types:
- OpName = atom()
- Arity = integer()
- Type = arith | bool | comp | list | send

Returns the Type of operator that OpName/Arity belongs to, or generates a function_clause error if it is not an operator at all.
erl_lint

Erlang Module

This module is used to check Erlang code for illegal syntax and other bugs. It also warns against coding practices which are not recommended.

The errors detected include:

- redefined and undefined functions
- unbound and unsafe variables
- illegal record usage.

Warnings include:

- unused functions and imports
- unused variables
- variables imported into matches
- variables exported from if/case/receive
- variables shadowed in lambdas and list comprehensions.

Some of the warnings are optional, and can be turned on by giving the appropriate option, described below.

The functions in this module are invoked automatically by the Erlang compiler and there is no reason to invoke these functions separately unless you have written your own Erlang compiler.

Exports

module(AbsForms) -> {ok,Warnings} | {error,Errors,Warnings}
module(AbsForms, FileName) -> {ok,Warnings} | {error,Errors,Warnings}
module(AbsForms, FileName, CompileOptions) -> {ok,Warnings} | {error,Errors,Warnings}

Types:

- AbsForms = [term()]
- FileName = FileName2 = atom() | string()
- Warnings = Errors = [{Filename2,[ErrorInfo]}]
- ErrorInfo = see separate description below.
- CompileOptions = [term()]

This function checks all the forms in a module for errors. It returns:

{ok,Warnings} There were no errors in the module.
{error,Errors,Warnings} There were errors in the module.
Since this module is of interest only to the maintainers of the compiler, and to avoid having the same description in two places to avoid the usual maintainence nightmare, the elements of Options that control the warnings are only described in [compile(3)].

The AbsForms of a module which comes from a file that is read through epp, the Erlang pre-processor, can come from many files. This means that any references to errors must include the file name (see epp(3) [page 103], or parser erl_parse(3) [page 114]. The warnings and errors returned have the following format:

```
{{FileName2,[ErrorMessage]}}
```

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(ErrorMessage) -> Chars

Types:
• ErrorMessage = errordesc()
• Chars = [char() | Chars]

Takes an ErrorMessage and returns a string which describes the error or warning. This function is usually called implicitly when processing an ErrorMessage structure (see below).
```

## Error Information

The ErrorMessage mentioned above is the standard ErrorMessage structure which is returned from all IO modules. It has the following format:

```
{ErrorLine, Module, ErrorMessage}
```

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

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

## See Also

erl_parse(3) [page 114], epp(3) [page 103]


**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_parse).

Exports

```erlang
parse_form(Tokens) -> {ok, AbsForm} | {error, ErrorInfo}

Types:
- Tokens = [Token]
- Token = {Tag,Line} | {Tag,Line,term()}
- Tag = atom()
- AbsForm = term()
- ErrorInfo = see section Error Information below.

This function parses Tokens as if it were a form. It returns:

- {ok, AbsForm} The parsing was successful. AbsForm is the abstract form of the parsed form.
- {error, ErrorInfo} An error occurred.

parse_exprs(Tokens) -> {ok, Expr_list} | {error, ErrorInfo}

Types:
- Tokens = [Token]
- Token = {Tag,Line} | {Tag,Line,term()}
- Tag = atom()
- Expr_list = [AbsExpr]
- AbsExpr = term()
- ErrorInfo = see section Error Information below.

This function parses Tokens as if it were a list of expressions. It returns:

- {ok, Expr_list} The parsing was successful. Expr_list is a list of the abstract forms of the parsed expressions.
- {error, ErrorInfo} An error occurred.

parse_term(Tokens) -> {ok, Term} | {error, ErrorInfo}
```
Types:
- Tokens = [Token]
- Token = {Tag,Line} | {Tag,Line,term()}
- Tag = atom()
- Term = term()
- ErrorInfo = see section Error Information below.

This function parses Tokens as if it were a term. It returns:

{ok, Term} The parsing was successful. Term is the Erlang term corresponding to the token list.

{error, ErrorInfo} An error occurred.

format_error(ErrorCode) -> Chars

Types:
- ErrorCode = erordesc()
- Chars = [Char() | Chars]

Uses an ErrorCode and returns a string which describes the error. This function is usually called implicitly when an ErrorInfo structure is processed (see below).

tokens(AbsTerm) -> Tokens
tokens(AbsTerm, MoreTokens) -> Tokens

Types:
- Tokens = MoreTokens = [Token]
- Token = {Tag,Line} | {Tag,Line,term()}
- Tag = atom()
- AbsTerm = term()
- ErrorInfo = see section Error Information below.

This function generates a list of tokens representing the abstract form AbsTerm of an expression. Optionally, it appends Moretokens.

normalise(AbsTerm) -> Data

Types:
- AbsTerm = Data = term()

Converts the abstract form AbsTerm of a term into a conventional Erlang data structure (i.e., the term itself). This is the inverse of abstract/1.

abstract(Data) -> AbsTerm

Types:
- Data = AbsTerm = term()

Converts the Erlang data structure Data into an abstract form of type AbsTerm. This is the inverse of normalise/1.
Error Information

The ErrorInfo mentioned above is the standard ErrorInfo structure which is returned from all IO modules. It has the format:

{ErrorLine, Module, ErrorDescriptor}

A string which describes the error is obtained with the following call:
apply(Module, format_error, ErrorDescriptor)

See Also

io(3) [page 203], erl_scan(3) [page 120], ERTS User’s Guide
erl_pp

Erlang Module

The functions in this module are used to generate aesthetically attractive representations of abstract forms, which are suitable for printing. All functions return (possibly deep) lists of characters and generate an error if the form is wrong. All functions can have an optional argument which specifies a hook that is called if an attempt is made to print an unknown form.

Exports

form(Form) -> DeepCharList
form(Form, HookFunction) -> DeepCharList

Types:
- Form = term()
- HookFunction = see separate description below.
- DeepCharList = [char() | DeepCharList]
Pretty prints a Form which is an abstract form of a type which is returned by erl_parse:parse_form.

attribute(Attribute) -> DeepCharList
attribute(Attribute, HookFunction) -> DeepCharList

Types:
- Attribute = term()
- HookFunction = see separate description below.
- DeepCharList = [char() | DeepCharList]
The same as form, but only for the attribute Attribute.

function(Function) -> DeepCharList
function(Function, HookFunction) -> DeepCharList

Types:
- Function = term()
- HookFunction = see separate description below.
- DeepCharList = [char() | DeepCharList]
The same as form, but only for the function Function.

guard(Guard) -> DeepCharList
guard(Guard, HookFunction) -> DeepCharList
Types:
- Form = term()
- HookFunction = see separate description below.
- DeepCharList = [char()|DeepCharList]

The same as form, but only for the guard test Guard.

exprs(Expressions) -> DeepCharList
exprs(Expressions, HookFunction) -> DeepCharList
exprs(Expressions, Indent, HookFunction) -> DeepCharList

Types:
- Expressions = term()
- HookFunction = see separate description below.
- Indent = integer()
- DeepCharList = [char()|DeepCharList]

The same as form, but only for the sequence of expressions in Expressions.

expr(Expression) -> DeepCharList
expr(Expression, HookFunction) -> DeepCharList
expr(Expression, Indent, HookFunction) -> DeepCharList
expr(Expression, Indent, Precedence, HookFunction) -> DeepCharList

Types:
- Expression = term()
- HookFunction = see separate description below.
- Indent = integer()
- Precedence =
- DeepCharList = [char()|DeepCharList]

This function prints one expression. It is useful for implementing hooks (see below).

Unknown Expression Hooks

The optional argument HookFunction, shown in the functions described above, defines a function which is called when an unknown form occurs where there should be a valid expression. It can have the following formats:

Function The hook function is called by:

    Function(Expr,
              CurrentIndentation,
              CurrentPrecedence,
              HookFunction)

none There is no hook function

The called hook function should return a (possibly deep) list of characters. expr/4 is useful in a hook.

If CurrentIndentation is negative, there will be no line breaks and only a space is used as a separator.
Bugs

It should be possible to have hook functions for unknown forms at places other than expressions.

See Also

io(3) [page 203], erl_parse(3) [page 114], erl_eval(3) [page 105]
erl_scan

Erlang Module

This module contains functions for tokenizing characters into Erlang tokens.

Exports

string(CharList, StartLine) -> {ok, Tokens, EndLine} | Error
string(CharList) -> {ok, Tokens, EndLine} | Error

Types:
- CharList = string()
- StartLine = EndLine = Line = integer()
- Tokens = [{atom(), Line} | {atom(), Line, term()}]
- Error = {error, ErrorInfo, EndLine}

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

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

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

tokens(Continuation, CharList, StartLine) -> Return

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

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

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

{ok, Tokens, EndLine} The scanning was successful. Tokens is the list of tokens including dot.
{eof, EndLine} End of file was encountered before any more tokens.
{error, ErrorInfo, EndLine} An error occurred.
{more, Continuation} More data is required for building a term. Continuation must be passed in a new call to tokens/3 when more data is available.

reserved_word(Atom) -> bool()
    Returns true if Atom is an Erlang reserved word, otherwise false.

format_error(ErrorCode) -> string()
    Types:
    • ErrorCode = errordesc()
    Takes an ErrorCode 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, ErrorCode}

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

apply(Module, format_error, ErrorCode)

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 203], erl_parse(3) [page 114]
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/IEC 9945-1. 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 124] or `create/3` [page 124] function.

Alternatively, for more control, the `open` [page 126], `add/3,4` [page 123], and `close/1` [page 123] functions can be used.

To extract all files from a tar file, use the `extract/1` [page 124] function. To extract only some files or to be able to specify some more options, use the `extract/2` [page 125] function.

To return a list of the files in a tar file, use either the `table/1` [page 126] or `table/2` [page 126] function. To print a list of files to the Erlang shell, use either the `t/1` [page 126] or `tt/1` [page 127] function.

To convert an error term returned from one of the functions above to a readable message, use the `format_error/1` [page 125] function.

**LIMITATIONS**

For maximum compatibility, it is safe to archive files with names up to 100 characters in length. Such tar files can generally be extracted by any **tar** program.

If filenames exceed 100 characters in length, the resulting tar file can only be correctly extracted by a POSIX-compatible **tar** program (such as Solaris **tar**), not by GNU **tar**.

File have longer names than 256 bytes cannot be stored at all.

The filename of the file a symbolic link points is always limited to 100 characters.
Exports

add(TarDescriptor, Filename, Options) -> RetValue

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

The add/3 function adds a file to a tar file that has been opened for writing by open/1 [page 126].

dereference By default, symbolic links will be stored as symbolic links in the tar file. Use the dereference option to override the default and store the file that the symbolic link points to into the tar file.
verbose Print an informational message about the file being added.

add(TarDescriptor, Filename, NameInArchive, Options) -> RetValue

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

The add/4 function adds a file to a tar file that has been opened for writing by open/1 [page 126]. It accepts the same options as add/3 [page 123].
NameInArchive is the name under which the file will be stored in the tar file. That is the name that the file will get when it will be extracted from the tar file.

close(TarDescriptor)

Types:
- TarDescriptor = term()

The close/1 function closes a tar file opened by open/1 [page 126].

create(Name, FileList) -> RetValue

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

```
create(Name, FileList, OptionList)
```

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

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.

```
extract(Name) -> RetValue
```

**Types:**
- `Name = filename()`
- `RetVal = ok | {error, {Name, Reason}}`
- `Reason = term()`

The `extract/1` function extracts all files from a tar archive. If the `Name` argument is given as `{binary, Binary}`, the contents of the binary is assumed to be a tar archive.

If the `Name` argument is given as `{file, Fd}`, `Fd` is assumed to be a file descriptor returned from the `file:open/2` function.

Otherwise, `Name` should be a filename.

```
extract(Name, OptionList)
```

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

### `format_error/1` conversion

```eryrl
define_error(Reason) -> string()
```

Types:
- `Reason = term()`

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

### `open/2` function

```eryrl
define(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` functions to add one file at the time into an opened tar file. When you are finished adding files, use the `close` 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) -&gt; 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 \texttt{tt/1} function prints names and information about all files in the tar file \texttt{Name} to the Erlang shell. (Similar to \texttt{"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(ver1)). 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 142]. 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 \texttt{lists:foldr/3}. The order in which the elements of the table are traversed is unspecified, except for tables of type \texttt{ordered_set}, for which they are traversed last to first.

\begin{verbatim}
from_dets(Tab, DetsTab) -> Tab
Types:
  \begin{itemize}
  \item Tab = tid() \mid atom()
  \item DetsTab = atom()
  \end{itemize}
Fills an already created ETS table with the objects in the already opened Dets table named DetsTab. The existing objects of the ETS table are kept unless overwritten.
\end{verbatim}

\begin{verbatim}
fun2ms(LiteralFun) -> MatchSpec
Types:
  \begin{itemize}
  \item LiteralFun \textendash{} see below
  \item MatchSpec = match_spec()
  \end{itemize}
Pseudo function that by means of a \texttt{parse_transform} translates \texttt{LiteralFun} typed as parameter in the function call to a \texttt{match_spec} [page 128]. With \enquote{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 \texttt{parse_transform} is implemented in the module \texttt{ms_transform} and the source must include the file \texttt{ms_transform.hrl} in \texttt{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 \texttt{-include_lib(\textquoterightstdlib/include/ms_transform.hrl\textquoteright)}. 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 \texttt{is XXX} guard tests. Language constructs that have no representation in a \texttt{match_spec} (like \texttt{if}, \texttt{case}, \texttt{receive} etc) are not allowed.

The return value is the resulting \texttt{match_spec}.

Example:

\begin{verbatim}
1> ets:fun2ms(fun({M,N}) when N > 3 -> M end).
[\[\{'$1'$,\'$2'$\},[\{'>'\,'$2'$,3\}],\[\'$1'$\]]]
\end{verbatim}

Variables from the environment can be imported, so that this works:

\begin{verbatim}
2> X=3.
3
3> ets:fun2ms(fun({M,N}) when N > X -> M end).
[\[\{'$1'$,\'$2'$\},[\{'>'\,'$2'$,\{\texttt{const},3\}\}],\[\'$1'$\]]]
\end{verbatim}

The imported variables will be replaced by \texttt{match_spec} \texttt{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 \texttt{match_spec} functions of course is allowed:
4> ets:fun2ms(fun([M,N]) when N > X, is_atom(M) -> M end).
Error: fun containing local Erlang function calls
('is_atom' called in guard) cannot be translated into match_spec
{error,transform_error}
5> ets:fun2ms(fun([M,N]) when N > X, is_atom(M) -> M end).
[{{'$$1',$$2'}},{{'>',$$2',{const,3}},{is_atom,'$$1'}},{'$$1'}]

As can be seen by the example, the function can be called from the shell too. The fun
needs to be literally in the call when used from the shell as well. Other means than the
parse_transform are used in the shell case, but more or less the same restrictions apply
(the exception being records, as they are not handled by the shell).

**Warning:**
If the parse_transform is not applied to a module which calls this pseudo function,
the call will fail in runtime (with a badarg). The module ets actually exports a
function with this name, but it should never really be called except for when using
the function in the shell. If the parse_transform is properly applied by including the
ms_transform.hrl header file, compiled code will never call the function, but the
function call is replaced by a literal match_spec.

For more information, see ms_transform(3) [page 238].

i() -> void()

Displays information about all ETS tables on tty.

i(Tab) -> void()

Types:
- Tab = tid() | atom()
  Browses the table Tab on tty.

info(Tab) -> [{Item, Value}] | undefined

Types:
- Tab = tid() | atom()
- Item = atom(), see below
- Value = term(), see below

Returns information about the table Tab as a list of {Item, Value} tuples.

**Warning:**
In Erlang/OTP R10B and earlier releases, this function erroneously returned a tuple.
This has now been corrected.

- Item=memory, Value=int()
  The number of words allocated to the table.
- Item=owner, Value=pid()
The pid of the owner of the table.
- Item=name, Value=atom()
The name of the table.
- Item=size, Value=int()
The number of objects inserted in the table.
- Item=node, Value=atom()
The node where the table is stored. This field is no longer meaningful as tables cannot be accessed from other nodes.
- Item=named_table, Value=true|false
  Indicates if the table is named or not.
- Item=type, Value=set|ordered_set|bag|duplicate_bag
  The table type.
- Item=keypos, Value=int()
The key position.
- Item=protection, Value=public|protected|private
  The table access rights.

info(Tab, Item) -> Value | undefined

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

Returns the information associated with Item for the table Tab. In addition to the [Item,Value] pairs defined for info/1, the following items are allowed:

- Item=fixed, Value=true|false
  Indicates if the table is fixed by any process or not.
- Item=safe_fixed, Value={FirstFixed,Info}|false
  If the table has been fixed using safe_fixtable/2, the call returns a tuple where FirstFixed is the time when the table was first fixed by a process, which may or may not be one of the processes it is fixed by right now. Info is a possibly empty lists of tuples {Pid,RefCount}, one tuple for every process the table is fixed by right now. RefCount is the value of the reference counter, keeping track of how many times the table has been fixed by the process. If the table never has been fixed, the call returns false.

init_table(Name, InitFun) -> true

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

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

If the type of the table is set and there is more than one object with a given key, one of the objects is chosen. This is not necessarily the last object with the given key in the sequence of objects returned by the input functions. This holds also for duplicated objects stored in tables of type duplicate_bag.

### Insert

**insert(Tab, ObjectOrObjects) -> true**

Types:
- Tab = tid() | atom()
- ObjectOrObjects = tuple() | [tuple()]

Inserts the object or all of the objects in the list ObjectOrObjects into the table Tab. If there already exists an object with the same key as one of the objects, and the table is a set or ordered_set table, the old object will be replaced. If the list contains more than one object with the same key and the table is a set/ordered_set, one will be inserted, which one is not defined.

### Insert New

**insert_new(Tab, ObjectOrObjects) -> bool()**

Types:
- Tab = tid() | atom()
- ObjectOrObjects = tuple() | [tuple()]

This function works exactly like insert/2, with the exception that instead of overwriting objects with the same key (in the case of set or ordered_set) or adding more objects with keys already existing in the table (in the case of bag and duplicate_bag), it simply returns false. If ObjectOrObjects is a list, the function checks every key prior to inserting anything. Nothing will be inserted if not all keys present in the list are absent from the table.

### Is Compiled MS

**is_compiled_ms(Term) -> bool()**

Types:
- Term = term()

This function is used to check if a term is a valid compiled match_spec [page 128]. The compiled match_spec is an opaque datatype which can not be sent between Erlang nodes nor be stored on disk. Any attempt to create an external representation of a compiled match_spec will result in an empty binary (\<<\>>). As an example, the following expression:

```erlang```
```
ets:is_compiled_ms(ets:match_spec_compile([{'_',[],[true]}])).
```

will yield true, while the following expressions:
MS = ets:match_spec_compile([{'_',[[],[true]]]),
Broken = binary_to_term(term_to_binary(MS)),
ets:is_compiled_ms(Broken).

will yield false, as the variable Broken will contain a compiled match_spec that has
passed through external representation.

**Note:**
The fact that compiled match_specs has no external representation is for
performance reasons. It may be subject to change in future releases, while this
interface will still remain for backward compatibility reasons.

```
last(Tab) -> Key | '$end_of_table'
  Types:
  • Tab = tid() | atom()
  • Key = term()
  Returns the last key Key according to Erlang term order in the table Tab of the
    ordered_set type. If the table is of any other type, the function is synonymous to
    first/2. If the table is empty, '$end_of_table' is returned.
    Use prev/2 to find preceding keys in the table.
```

```
lookup(Tab, Key) -> [Object]
  Types:
  • Tab = tid() | atom()
  • Key = term()
  • Object = tuple()
  Returns a list of all objects with the key Key in the table Tab.
  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.

match(Tab, Pattern) -> [Match]
Types:
- Tab = tid() | atom()
- Pattern = tuple()
- Match = [term()]

Matches the objects in the table Tab against the pattern Pattern.

A pattern is a term that may contain:
- bound parts (Erlang terms),
- '_' which matches any Erlang term, and
- pattern variables: '$_N'$ where N=0,1,...

The function returns a list with one element for each matching object, where each element is an ordered list of pattern variable bindings. An example:

6> ets:match(T, '$_1'). % Matches every object in the table
[[[rufsen,dog,7]],[[brunte,horse,5]],[[ludde,dog,5]]]

7> ets:match(T, {'_',dog,'$_1'}).
[[7],[5]]

8> ets:match(T, {'_','_','cow','$_1'}).
[]

If the key is specified in the pattern, the match is very efficient. If the key is not specified, i.e. if it is a variable or an underscore, the entire table must be searched. The search time can be substantial if the table is very large.

On tables of the ordered_set type, the result is in the same order as in a first/next traversal.

match(Tab, Pattern, Limit) -> { [Match],Continuation} | '$end_of_table'

Types:
- Tab = tid() | atom()
- Pattern = tuple()
- Match = [term()]
- Continuation = term()

Works like ets:match/2 but only returns a limited (Limit) number of matching objects. The Continuation term can then be used in subsequent calls to ets:match/1 to get the next chunk of matching objects. This is a space efficient way to work on objects in a table which is still faster than traversing the table object by object using ets:first/1 and ets:next/1.

'$end_of_table' is returned if the table is empty.
match(Continuation) -> { [Match], Continuation } \| '$end_of_table'

Types:
  - Match = [term()]
  - Continuation = term()

Continues a match started with ets:match/3. The next chunk of the size given in the initial ets:match/3 call is returned together with a new Continuation that can be used in subsequent calls to this function.

'$end_of_table' is returned when there are no more objects in the table.

match_delete(Tab, Pattern) -> true

Types:
  - Tab = tid() \| atom()
  - Pattern = tuple()

Deletes all objects which match the pattern Pattern from the table Tab. See match/2 for a description of patterns.

match_object(Tab, Pattern) -> [Object]

Types:
  - Tab = tid() \| atom()
  - Pattern = Object = tuple()

Matches the objects in the table Tab against the pattern Pattern. See match/2 for a description of patterns. The function returns a list of all objects which match the pattern.

If the key is specified in the pattern, the match is very efficient. If the key is not specified, i.e. if it is a variable or an underscore, the entire table must be searched. The search time can be substantial if the table is very large.

On tables of the ordered_set type, the result is in the same order as in a first/next traversal.

match_object(Tab, Pattern, Limit) -> { [Match], Continuation } \| '$end_of_table'

Types:
  - Tab = tid() \| atom()
  - Pattern = tuple()
  - Match = [term()]
  - Continuation = term()

Works like ets:match_object/2 but only returns a limited (Limit) number of matching objects. The Continuation term can then be used in subsequent calls to ets:match_object/1 to get the next chunk of matching objects. This is a space efficient way to work on objects in a table which is still faster than traversing the table object by object using ets:first/1 and ets:next/1.

'$end_of_table' is returned if the table is empty.

match_object(Continuation) -> { [Match], Continuation } \| '$end_of_table'

Types:
match_spec compile(MatchSpec) -> CompiledMatchSpec

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

This function transforms a match spec [page 128] 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:isCompiledMs/1.

If the term MatchSpec can not be compiled (does not represent a valid match_spec), a badarg fault is thrown.

Note:
This function has limited use in normal code, it is used by Dets to perform the dets:select operations.

match_spec_run(List,CompiledMatchSpec) -> list()

Types:
- List = [ tuple() ]
- CompiledMatchSpec = comp_match_spec()

This function executes the matching specified in a compiled match_spec [page 128] 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 length of the parameter List. The two calls in the following example will give the same result (but certainly not the same execution time.):

Table = ets:new...  
MatchSpec = ....  
% The following call...  
ets:match_spec_run(ets:tab2list(Table),  
ets:match_spec_compile(MatchSpec)),  
% ...will give the same result as the more common (and more efficient)  
ets:select(Table,MatchSpec),
Note: This function has limited use in normal code, it is used by Dets to perform the dets:select operations and by Mnesia during transactions.

member(Tab, Key) -> true | false

Types:
- Tab = tid() | atom()
- Key = term()

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

new(Name, Options) -> tid()

Types:
- Name = atom()
- Options = [Option]
  - Option = Type | Access | named_table | {keypos,Pos}
  - Type = set | ordered_set | bag | duplicate_bag
  - Access = public | protected | private
  - Pos = int()

Creates a new table and returns a table identifier which can be used in subsequent operations. The table identifier can be sent to other processes so that a table can be shared between different processes within a node.

The parameter Options is a list of atoms which specifies table type, access rights, key position and if the table is named or not. If one or more options are left out, the default values are used. This means that not specifying any options ([]) is the same as specifying [set,protected,{keypos,1}].

- set The table is a set table - one key, one object, no order among objects. This is the default table type.
- ordered_set The table is a ordered_set table - one key, one object, ordered in Erlang term order, which is the order implied by the < and > operators. Tables of this type have a somewhat different behavior in some situations than tables of the other types.
- 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=ets:new(x,[]),
...
{_,C} = ets:select(T,ets:fun2ms(fun({N,_,}=A)
when (N rem 10) =:= 0 ->
A
end),10),
Broken = binary_to_term(term_to_binary(C)),
ets:select(Broken).

...while the following sequence will work:

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

...as the call to ets:repair_continuation/2 will reestablish the (deliberately) invalidated continuation Broken.

Note:
This function is very rarely needed in application code. It is used by Mnesia to implement distributed select/3 and select/1 sequences. A normal application would either use Mnesia or keep the continuation from being converted to external format.

The reason for not having an external representation of a compiled match spec is performance. It may be subject to change in future releases, while this interface will remain for backward compatibility.

safe_fixtable(Tab, true|false) -> true

Types:
  • Tab = tid() | atom()
Fixes a table of the set, bag or duplicate_bag table type for safe traversal.

A process fixes a table by calling `safe_fixtable(Tab, true)`. The table remains fixed until the process releases it by calling `safe_fixtable(Tab, false)`, or until the process terminates.

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

When a table is fixed, a sequence of `first/1` and `next/2` calls are guaranteed to succeed even if objects are removed during the traversal. An example:

```erlang
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.

```erlang
select(Tab, MatchSpec) -> [Match]

Types:
- `Tab` = `tid()` | `atom()`
- `Match` = `term()`
- `MatchSpec` = `match_spec()`

Matches the objects in the table `Tab` using a match spec [page 128]. This is a more general call than the `ets:match/2` and `ets:match_object/2` calls. In its simplest forms the `match_specs` look like this:

- `MatchSpec = [MatchFunction]`
- `MatchFunction = {MatchHead, [Guard], [Result]}`
- `MatchHead = "Pattern as in ets:match"`
- `Guard = {"Guardtest name", ...}`
- Result = "Term construct"

This means that the match_spec is always a list of one or more tuples (of arity 3). The tuples first element should be a pattern as described in the documentation of ets:match/2. The second element of the tuple should be a list of 0 or more guard tests (described below). The third element of the tuple should be a list containing a description of the value to actually return. In almost all normal cases the list contains exactly one term which fully describes the value to return for each object.

The return value is constructed using the "match variables" bound in the MatchHead or using the special match variables '$.' (the whole matching object) and '$$' (all match variables in a list), so that the following ets:match/2 expression:

```erlang
ets:match(Tab,{'$1','$2','$3'})
```

is exactly equivalent to:

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

- and the following ets:match_object/2 call:

```erlang
ets:match_object(Tab,{'$1','$2','$1'})
```

is exactly equivalent to

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

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

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

gives the same output as:

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

i.e all the bound variables in the match head as a list. If tuples are to be constructed, one has to write a tuple of arity 1 with the single element in the tuple being the tuple one wants to construct (as an ordinary tuple could be mistaken for a Guard). Therefore the following call:

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

gives the same output as:

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

- this syntax is equivalent to the syntax used in the trace patterns (see [dbg(3)]).

The Guards are constructed as tuples where the first element is the name of the test and the rest of the elements are the parameters of the test. To check for a specific type (say a list) of the element bound to the match variable '$1', one would write the test as `{is_list, '$1'}`. If the test fails, the object in the table will not match and the next MatchFunction (if any) will be tried. Most guard tests present in Erlang can be used, but only the new versions prefixed is_ are allowed (like is_float, is_atom etc).

The Guard section can also contain logic and arithmetic operations, which are written with the same syntax as the guard tests (prefix notation), so that a guard test written in Erlang looking like this:

```erlang
is_integer(X), is_integer(Y), X + Y < 4711
```

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

```erlang
[{is_integer, '$1'}, {is_integer, '$2'}, {'<', {'+', '$1', '$2'}, 4711}]
```
select(Tab, MatchSpec, Limit) -> [[Match],Continuation] | '$end_of_table'

Types:
- Tab = tid() | atom()
- Match = term()
- MatchSpec = match_spec()
- Continuation = term()

Works like ets:select/2 but only returns a limited (Limit) number of matching objects. The Continuation term can then be used in subsequent calls to ets:select/1 to get the next chunk of matching objects. This is a space efficient way to work on objects in a table which is still faster than traversing the table object by object using ets:first/1 and ets:next/1.

'$end_of_table' is returned if the table is empty.

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

Types:
- Match = term()
- Continuation = term()

Continues a match started with ets:select/3. The next chunk of the size given in the initial ets:select/3 call is returned together with a new Continuation that can be used in subsequent calls to this function.

'$end_of_table' is returned when there are no more objects in the table.

select_delete(Tab, MatchSpec) -> NumDeleted

Types:
- Tab = tid() | atom()
- Object = tuple()
- MatchSpec = match_spec()
- NumDeleted = integer()

Matches the objects in the table Tab using a match_spec [page 128]. 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 128]. 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.

\[\text{slot(Tab, I)} \rightarrow \text{[Object] | 'end_of_table'}\]

Types:
- \(\text{Tab} = \text{tid()} \mid \text{atom()}\)
- \(\text{I} = \text{int()}\)
- \(\text{Object} = \text{tuple()}\)

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 \(\text{I}=0\) 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.

\[\text{tab2file(Tab, Filename)} \rightarrow \text{ok} \mid \{\text{error,Reason}\}\]

Types:
- \(\text{Tab} = \text{tid()} \mid \text{atom()}\)
- \(\text{Filename} = \text{string()} \mid \text{atom()}\)
- \(\text{Reason} = \text{term()}\)

Dumps the table Tab to the file Filename. The implementation of this function is not efficient.

\[\text{tab2list(Tab)} \rightarrow \text{[Object]}\]

Types:
- \(\text{Tab} = \text{tid()} \mid \text{atom()}\)
- \(\text{Object} = \text{tuple()}\)

Returns a list of all objects in the table Tab.

\[\text{table(Tab [, Options])} \rightarrow \text{QueryHandle}\]

Types:
- \(\text{Tab} = \text{tid()} \mid \text{atom()}\)
- \(\text{QueryHandle} = -\text{a query handle, see qlc(3)}-\)
- \(\text{Options} = \{\text{Option} \mid \text{Option}\}\)
- \(\text{Option} = \{\text{n_objects, NObjects} \mid \{\text{traverse, TraverseMethod}\}\}
- \(\text{NObjects} = \text{default} \mid \text{integer()} > 0\)
returns a QLC (Query List Comprehension) query handle. The module qlc implements a query language aimed mainly at Mnesia but ETS tables, Dets tables, and lists are also recognized by QLC as sources of data. Calling ets:table/1,2 is the means to make the ETS table Tab usable to QLC.

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

- first, next. The table is traversed one key at a time by calling ets:first/1 and ets:next/2.
- last, prev. The table is traversed one key at a time by calling ets:last/1 and ets:prev/2.
- select. The table is traversed by calling ets:select/3 and ets:select/1. The option n_objects determines the number of objects returned (the third argument of select/3); the default is to return 100 objects at a time. The match_spec [page 128] (the second argument of select/3) is assembled by QLC: simple filters are translated into equivalent match_specs while more complicated filters have to be applied to all objects returned by select/3 given a match_spec that matches all objects.
- {select, MatchSpec}. As for select the table is traversed by calling ets:select/3 and ets:select/1. The difference is that the match_spec is explicitly given. This is how to state match_specs that cannot easily be expressed within the syntax provided by QLC.

The following example uses an explicit match_spec to traverse the table:

```erlang
g9> ets:insert(Tab = ets:new(t, []), [[1,a],[2,b],[3,c],[4,d]]),
   MS = ets:fun2ms(fun(X,Y) when (X = 1) or (X = 5) -> Y end),
   QH1 = ets:table(Tab, [{traverse, [select, MS]}]).
```

An example with implicit match_spec:

```erlang
g10> QH2 = qlc:q([Y || X,Y <- ets:table(Tab), (X > 1) or (X < 5)])).
```

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

```erlang
g11> qlc:info(QH1) =:= qlc:info(QH2).
true
```

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

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

Types:
- Tuple = tuple()
- MatchSpec = match_spec()
- Result = term()
• Errors = [{warning|error, string()}]

This function is a utility to test a match_spec [page 128] 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 (keypos+1) is updated.

If a Threshold is specified, the counter will be reset to the value SetValue if the following conditions occur:

• The Incr is not negative (>= 0) and the result would be greater than (>) Threshold
• The Incr is negative (< 0) and the result would be less than (<) Threshold

The function will fail with reason badarg if:

• the table is not of type set or ordered_set,
• no object with the right key exists,
• the object has the wrong arity,
• the element to update is not an integer,
• the element to update is also the key, or,
• any of Pos, Incr, Threshold or SetValue is not an integer
The functions of this module sort terms on files, merge already sorted files, and check files for sortedness. Chunks containing binary terms are read from a sequence of files, sorted internally in memory and written on temporary files, which are merged producing one sorted file as output. Merging is provided as an optimization; it is faster when the files are already sorted, but it always works to sort instead of merge.

On a file, a term is represented by a header and a binary. Two options define the format of terms on files:

- `{header, HeaderLength}`. HeaderLength determines the number of bytes preceding each binary and containing the length of the binary in bytes. Default is 4. The order of the header bytes is defined as follows: if `B` is a binary containing a header only, the size `Size` of the binary is calculated as `<<Size:HeaderLength/unit:8>> = B`.

- `{format, Format}`. The format determines the function that is applied to binaries in order to create the terms that will be sorted. The default value is `binary_term`, which is equivalent to `fun binary_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`. 
sorter

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:getcwd() is used instead. The names of temporary files are derived from the Erlang nodename (node()), the process identifier of the current Erlang emulator (os:getpid()), and a timestamp (erlang:now()); a typical name would be fs_mynode@myhost_1763_1043_337000_266005.17, where 17 is a sequence number. Existing files will be overwritten. Temporary files are deleted unless some uncaught EXIT signal occurs.

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.

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 sort or keysort. Each output function is called exactly once. When some output function has been applied to all of the results or an error occurs, the last function is called with the argument close, and the reply is
returned as value of the current call to the sort or merge function. If a function is given as input and the last input function returns \{end_of_input, Value\}, the function given as output will be called with the argument \{value, Value\}. This makes it easy to initiate the sequence of output functions with a value calculated by the input functions.

As an example, consider sorting the terms on a disk log file. A function that reads chunks from the disk log and returns a list of binaries is used as input. The results are collected in a list of terms.

```
sort(Log) ->
    \{ok, _\} = disk_log:open([[name,Log], [mode,read_only]]),
    Input = input(Log, start),
    Output = output([[]]),
    Reply = file_sorter:sort(Input, Output, \{format,term\}),
    ok = disk_log:close(Log),
    Reply.
```

```
input(Log, Cont) ->
    fun(close) ->
        ok;
    (read) ->
        case disk_log:chunk(Log, Cont) of
        \{error, Reason\} ->
            \{error, Reason\};
        \{Cont2, Terms\} ->
            \{Terms, input(Log, Cont2)\};
        \{Cont2, Terms, _Badbytes\} ->
            \{Terms, input(Log, Cont2)\};
        eof ->
            end_of_input
        end.
end.
```

```
output(L) ->
    fun(close) ->
        lists:append(lists:reverse(L));
    (Terms) ->
        output([Terms | L])
    end.
```

Further examples of functions as input and output can be found at the end of the file_sorter module; the term format is implemented with functions.

The possible values of Reason returned when an error occurs are:

- \{bad_object, \{bad_object, FileName\}\}: Applying the format function failed for some binary, or the key(s) could not be extracted from some term.
- \{bad_term, FileName\}: \io:read/2 failed to read some term.
- \{file_error, FileName, Reason2\}: See file(3) for an explanation of Reason2.
- \{premature_eof, FileName\}: End-of-file was encountered inside some binary term.

Types
Binary = binary()
FileName = file_name()
FileNames = [FileName]
ICommand = read | close
IReply = end_of_input | {end_of_input, Value} | {[Object], Infun} | InputReply
Infun = fun(ICommand) -> IReply
Input = FileNames | Infun
InputReply = Term
KeyPos = int() > 0 | [int()>0]
OCommand = {value, Value} | [Object] | close
OReply = Outfun | OutputReply
Object = Term | Binary
Outfun = fun(OCommand) -> OReply
Output = FileName | Outfun
OutputReply = Term
Term = term()
Value = Term

Exports

sort(FileName) -> Reply
sort(Input, Output) -> Reply
sort(Input, Output, Options) -> Reply

Types:
- Reply = ok | {error, Reason} | InputReply | OutputReply

Sorts terms on files.

sort(FileName) is equivalent to sort([FileName], FileName).
sort(Input, Output) is equivalent to sort(Input, Output, []).

keysort(KeyPos, FileName) -> Reply
keysort(KeyPos, Input, Output) -> Reply
keysort(KeyPos, Input, Output, Options) -> Reply

Types:
- Reply = ok | {error, Reason} | InputReply | OutputReply

Sorts tuples on files. The sort is performed on the element(s) mentioned in KeyPos. If two tuples compare equal on one element, next element according to KeyPos is compared. The sort is stable.

keysort(N, FileName) is equivalent to keysort(N, [FileName], FileName).
keysort(N, Input, Output) is equivalent to keysort(N, Input, Output, []).

merge(FileNames, Output) -> Reply
merge(FileNames, Output, Options) -> Reply

Types:
- Reply = ok | {error, Reason} | OutputReply
Merges terms on files. Each input file is assumed to be sorted.

merge(FileNames, Output) is equivalent to merge(FileNames, Output, []).

keymerge(KeyPos, FileNames, Output) -> Reply
keymerge(KeyPos, FileNames, Output, Options) -> Reply
Types:
  • Reply = ok | {error, Reason} | OutputReply
Merges tuples on files. Each input file is assumed to be sorted on key(s).

keymerge(KeyPos, FileNames, Output) is equivalent to keymerge(KeyPos, FileNames, Output, []).

check(FileName) -> Reply
check(FileNames, Options) -> Reply
Types:
  • Reply = ok, [Result] | error, Reason
  • Result = {FileName, TermPosition, Term}
  • TermPosition = int() > 1
Checks files for sortedness. If a file is not sorted, the first out-of-order element is returned. The first term on a file has position 1.

check(FileName) is equivalent to check([FileName], []).

keycheck(KeyPos, FileName) -> CheckReply
keycheck(KeyPos, FileNames, Options) -> Reply
Types:
  • Reply = ok, [Result] | error, Reason
  • Result = {FileName, TermPosition, Term}
  • TermPosition = int() > 1
Checks files for sortedness. If a file is not sorted, the first out-of-order element is returned. The first term on a file has position 1.

keycheck(KeyPos, FileName) is equivalent to keycheck(KeyPos, [FileName], []).
This module contains utilities on a higher level than the `file` module.

**Exports**

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

Types:
- `Name = filename() | dirname()`
- `Reason = posix() – see file(3)`

The `ensure_dir/1` function ensures that all parent directories for the given file or directory name `Name` exist, trying to create them if necessary. Returns `ok` if all parent directories already exist or could be created, or `{error, Reason}` if some parent directory does not exist and could not be created for some reason.

```erlang
file_size(Filename) -> integer()
```

The `file_size` function returns the size of the given file.

```erlang
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.

```erlang
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.

```erlang
is_file(Name) -> true | false
```
Types:

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

The \text{is\_file/1} function returns \text{true} if \text{Name} refers to a file or a directory, and \text{false} otherwise.

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

Types:

- \( \text{Name} = \text{filename()} \)

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

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

Types:

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

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

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

Types:

- \( \text{Wildcard} = \text{filename()} \mid \text{dirname()} \)

The \text{wildcard/1} function returns a list of all files that match Unix-style wildcard-string \text{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.
- \( \{\text{Item},\ldots\} \) 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.
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:

```
```prolog
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) \rightarrow \text{string()}

Types:
- Filename = name()

Returns the directory part of Filename.

13> filename:dirname("/usr/src/kalle.erl").
   "/usr/src"
14> filename:dirname("kalle.erl").
   "."
15> filename:dirname("\usr\src\kalle.erl"). \% Windows
    "/usr/src"

\[extension\](Filename) \rightarrow \text{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) \rightarrow \text{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.

```plaintext
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.

```plaintext
19> filename:nativename("/usr/local/bin/"). % Unix
="/usr/local/bin"
7> filename:nativename("/usr/local/bin/"). % Windows
="/usr\local\bin"
```

pathtype(Path) -> absolute | relative | volumerelative

Returns the type of path, one of absolute, relative, or volumerelative.

- absolute: The path name refers to a specific file on a specific volume.
  - Unix example: /usr/local/bin
  - Windows example: D:/usr/local/bin
relative The path name is relative to the current working directory on the current volume.
Example: foo/bar, ../src
volumerelative The path name is relative to the current working directory on a specified volume, or it is a specific file on the current working volume.
Windows example: D:bar.erl, /bar/foo.erl

rootname(Filename) -> string()
rootname(Filename, Ext) -> string()

Types:
- Filename = Ext = name()

Remove a filename extension. rootname/2 works as rootname/1, except that the extension is removed only if it is Ext.

20> filename:rootname("/beam.src/kalle").
/beam.src/kalle"
21> filename:rootname("/beam.src/foo.erl").
"/beam.src/foo"
22> filename:rootname("/beam.src/foo.erl", ".erl").
"/beam.src/foo"
23> filename:rootname("/beam.src/foo.beam", ".erl").
"/beam.src/foo.beam"

split(Filename) -> Components

Types:
- Filename = name()
- Components = [string()]

Returns a list whose elements are the path components of Filename.

24> filename:split("/usr/local/bin").
["/", "usr", "local", "bin"]
25> filename:split("foo/bar").
["foo", "bar"]
26> filename:split("a:\msdev\include").
["a:\", "msdev", "include"]

find_src(Beam) -> {SourceFile, Options}
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

Erlang Module

An implementation of ordered sets using Prof. Arne Andersson’s General Balanced Trees. This can be much more efficient than using ordered lists, for larger sets, but depends on the application.

Complexity note

The complexity on set operations is bounded by either $O(|S|)$ or $O(|T| \times \log(|S|))$, 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

\texttt{gb\_set()} = a GB set

Exports

\texttt{add(\text{Element}, \text{Set1}) -> Set2}
\texttt{add\_element(\text{Element}, \text{Set1}) -> Set2}

Types:
• \text{Element} = \text{term()}
• \text{Set1} = \text{Set2} = \text{gb\_set()}

Returns a new \text{gb\_set} formed from \text{Set1} with \text{Element} inserted. If \text{Element} is already an element in \text{Set1}, nothing is changed.

\texttt{balance(\text{Set1}) -> Set2}

Types:
• \text{Set1} = \text{Set2} = \text{gb\_set()}

Rebalances the tree representation of \text{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.

\texttt{delete(\text{Element}, \text{Set1}) -> Set2}

Types:
• \text{Element} = \text{term()}
• \text{Set1} = \text{Set2} = \text{gb\_set()}

Returns a new \text{gb\_set} formed from \text{Set1} with \text{Element} removed. Assumes that \text{Element} is present in \text{Set1}.

\texttt{delete\_any(\text{Element}, \text{Set1}) -> Set2}
\texttt{del\_element(\text{Element}, \text{Set1}) -> Set2}

Types:
• \text{Element} = \text{term()}
• \text{Set1} = \text{Set2} = \text{gb\_set()}

Returns a new \text{gb\_set} formed from \text{Set1} with \text{Element} removed. If \text{Element} is not an element in \text{Set1}, nothing is changed.

\texttt{difference(\text{Set1}, \text{Set2}) -> Set3}
subtract(Set1, Set2) -> Set3

Types:
- Set1 = Set2 = Set3 = gb_set()

Returns only the elements of Set1 which are not also elements of Set2.

empty() -> Set
new() -> Set

Types:
- Set = gb_set()

Returns a new empty gb_set.

filter(Pred, Set1) -> Set2

Types:
- Pred = fun (E) -> bool()
- E = term()
- Set1 = Set2 = gb_set()

Filters elements in Set1 using predicate function Pred.

fold(Function, Acc0, Set) -> Acc1

Types:
- Function = fun (E, AccIn) -> AccOut
- Acc0 = Acc1 = AccIn = AccOut = term()
- E = term()
- Set = gb_set()

Folds Function over every element in Set returning the final value of the accumulator.

from_list(List) -> Set

Types:
- List = [term()]
- Set = gb_set()

Returns a gb_set of the elements in List, where List may be unordered and contain duplicates.

from_ordset(List) -> Set

Types:
- List = [term()]
- Set = gb_set()

Turns an ordered-set list List into a gb_set. The list must not contain duplicates.

insert(Element, Set1) -> Set2

Types:
- Element = term()
Set1 = Set2 = gb_set()
Returns a new gb_set formed from Set1 with Element inserted. Assumes that Element is not present in Set1.

intersection(Set1, Set2) -> Set3
Types:
- Set1 = Set2 = Set3 = gb_set()
Returns the intersection of Set1 and Set2.

intersection(SetList) -> Set
Types:
- SetList = [gb_set()]
- Set = gb_set()
Returns the intersection of the non-empty list of gb_sets.

is_empty(Set) -> bool()
Types:
- Set = gb_set()
Returns true if Set is an empty set, and false otherwise.

is_member(Element, Set) -> bool()

is_element(Element, Set) -> bool()
Types:
- Element = term()
- Set = gb_set()
Returns true if Element is an element of Set, otherwise false.

is_set(Set) -> bool()
Types:
- Set = gb_set()
Returns true if Set appears to be a gb_set, otherwise false.

is_subset(Set1, Set2) -> bool()
Types:
- Set1 = Set2 = gb_set()
Returns true when every element of Set1 is also a member of Set2, otherwise false.

iterator(Set) -> Iter
Types:
- Set = gb_set()
- Iter = term()
Returns an iterator that can be used for traversing the entries of Set; see \texttt{next/1}. The implementation of this is very efficient; traversing the whole set 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.

\begin{verbatim}
largest(Set) -> term()
Types:
  - Set = gb_set()
Returns the largest element in Set. Assumes that Set is nonempty.

next(Iter1) -> \{Element, Iter2 | none\}
Types:
  - Iter1 = Iter2 = Element = term()
Returns \{Element, Iter2\} where Element is the smallest element referred to by the iterator Iter1, and Iter2 is the new iterator to be used for traversing the remaining elements, or the atom none if no elements remain.

singleton(Element) -> gb_set()
Types:
  - Element = term()
Returns a gb_set containing only the element Element.

size(Set) -> int()
Types:
  - Set = gb_set()
Returns the number of elements in Set.

smallest(Set) -> term()
Types:
  - Set = gb_set()
Returns the smallest element in Set. Assumes that Set is nonempty.

take_largest(Set1) -> \{Element, Set2\}
Types:
  - Set1 = Set2 = gb_set()
  - Element = term()
Returns \{Element, Set2\}, where Element is the largest element in Set1, and Set2 is this set with Element deleted. Assumes that Set1 is nonempty.

take_smallest(Set1) -> \{Element, Set2\}
Types:
  - Set1 = Set2 = gb_set()
Element = term()

Returns \{Element, Set2\}, where Element is the smallest element in Set1, and Set2 is this set with Element deleted. Assumes that Set1 is nonempty.

to_list(Set) -> List

Types:
- Set = gb_set()
- List = [term()]

Returns the elements of Set as a list.

union(Set1, Set2) -> Set3

Types:
- Set1 = Set2 = Set3 = gb_set()

Returns the merged (union) gb_set of Set1 and Set2.

union(SetList) -> Set

Types:
- SetList = [gb_set()]
- Set = gb_set()

Returns the merged (union) gb_set of the list of gb_sets.

SEE ALSO

gb_trees(3) [page 168], ordsets(3) [page 250], sets(3) [page 293]
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:
    - \text{nil}.

There is no attempt to balance trees after deletions. Since deletions do not increase the height of a tree, this should be OK.

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

Performance is comparable to the AVL trees in the Erlang book (and faster in general due to less overhead); the difference is that deletion works for these trees, but not for the book’s trees. Behaviour is logarithmic (as it should be).

DATA TYPES

\texttt{gb_tree() = a GB tree}

Exports

\texttt{balance(Tree1) -> Tree2}

Types:
- \( \text{Tree1 = Tree2 = gb_tree()} \)

Rebalances \texttt{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.

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

Types:
- Key = term()
- Tree1 = Tree2 = gb_tree()

Removes the node with key Key from Tree1; returns new tree. Assumes that the key is present in the tree, crashes otherwise.

delete_any(Key, Tree1) -> Tree2

Types:
- Key = term()
- Tree1 = Tree2 = gb_tree()

Removes the node with key Key from Tree1 if the key is present in the tree, otherwise does nothing; returns new tree.

empty() -> Tree

Types:
- Tree = gb_tree()

Returns a new empty tree.

enter(Key, Val, Tree1) -> Tree2

Types:
- Key = Val = term()
- Tree1 = Tree2 = gb_tree()

Inserts Key with value Val into Tree1 if the key is not present in the tree, otherwise updates Key to value Val in Tree1. Returns the new tree.

from_orddict(List) -> Tree

Types:
- List = [{Key, Val}]
- Key = Val = term()
- Tree = gb_tree()

Turns an ordered list List of key-value tuples into a tree. The list must not contain duplicate keys.

get(Key, Tree) -> Val

Types:
- Key = Val = term()
- Tree = gb_tree()

Retrieves the value stored with Key in Tree. Assumes that the key is present in the tree, crashes otherwise.

lookup(Key, Tree) -> {value, Val} | none

Types:
- Key = Val = term()
- Tree = gb_tree()
Looks up Key in Tree; returns \{value, Val\}, or none if Key is not present.

**insert(Key, Val, Tree1) \rightarrow Tree2**

Types:
- Key = Val = term()
- Tree1 = Tree2 = gb_tree()

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

**is_defined(Key, Tree) \rightarrow bool()**

Types:
- Tree = gb_tree()

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

**is_empty(Tree) \rightarrow bool()**

Types:
- Tree = gb_tree()

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

**iterator(Tree) \rightarrow Iter**

Types:
- Tree = gb_tree()
- Iter = term()

Returns an iterator that can be used for traversing the entries of Tree; see next/1. The implementation of this is very efficient; traversing the whole tree using next/1 is only slightly slower than getting the list of all elements using to_list/1 and traversing that. The main advantage of the iterator approach is that it does not require the complete list of all elements to be built in memory at one time.

**keys(Tree) \rightarrow [Key]**

Types:
- Tree = gb_tree()
- Key = term()

Returns the keys in Tree as an ordered list.

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

Types:
- Tree = gb_tree()
- Key = Val = term()

Returns \{Key, Val\}, where Key is the largest key in Tree, and Val is the value associated with this key. Assumes that the tree is nonempty.

**next(Iter1) \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.

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 162], dict(3) [page 87]
gen_event

Erlang Module

A behaviour module for implementing event handling functionality. The OTP event handling model consists of a generic event manager process with an arbitrary number of event handlers which are added and deleted dynamically.

An event manager implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into an OTP supervision tree. Refer to OTP Design Principles for more information.

Each event handler is implemented as a callback module exporting a pre-defined set of functions. The relationship between the behaviour functions and the callback functions can be illustrated as follows:

<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>Module:handle_event/2</td>
</tr>
<tr>
<td>gen_event:notify</td>
<td>Module:handle_call/2</td>
</tr>
<tr>
<td>gen_event:sync_notify</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>Module:code_change/3</td>
</tr>
<tr>
<td>gen_event:stop</td>
<td>Module:terminate/2</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**

```
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` is `{local, Name}`, the event manager is registered locally as `Name` using `register/2`. If `EventMgrName` is `{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.

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

Types:
- EventMgrName = {local, Name} | {global, Name}
- 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.

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

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

Types:

- EventMgr = Name | \{Name, Node\} | \{global, Name\} | pid()
- Name = Node = atom()
- Handler = Module | \{Module, Id\}
- Module = atom()
- Id = term()
- Args = term()
- Result = ok | \{‘EXIT’, Reason\} | term()
- Reason = term()

Adds a new event handler in the same way as add_handler/3 but will also supervise the connection between the event handler and the calling process.

- If the calling process later terminates with Reason, the event manager will delete the event handler by calling Module:terminate/2 with \{stop, Reason\} as argument.
- If the event handler later is deleted, the event manager sends a message gen_event(EXIT, Handler, Reason) to the calling process. Reason is one of the following:
  - normal, if the event handler has been removed due to a call to delete_handler/3, or remove_handler has been returned by a callback function (see below).
  - shutdown, if the event handler has been removed because the event manager is terminating.
- {swapped, NewHandler, Pid}, if the process Pid has replaced the event handler with another event handler NewHandler using a call to swap_handler/3 or swap_sup_handler/3.
- a term, if the event handler is removed due to an error. Which term depends on the error.

See add_handler/3 for a description of the arguments and return values.

notify(EventMgrRef, Event) -> ok
sync_notify(EventMgrRef, Event) -> ok

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

Sends an event notification to the event manager EventMgrRef. The event manager will call Module:handle_event/2 for each installed event handler to handle the event.

notify is asynchronous and will return immediately after the event notification has been sent. sync_notify is synchronous in the sense that it will return ok after the event has been handled by all event handlers.

See add_handler/3 for a description of EventMgrRef.

Event is an arbitrary term which is passed as one of the arguments to Module:handle_event/2.

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

call(EventMgrRef, Handler, Request) -> Result
call(EventMgrRef, Handler, Request, Timeout) -> Result

Types:
- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler = Module | {Module,Id}
- Module = atom()
- Id = term()
- Request = term()
- Timeout = int() > 0 | infinity
- Result = Reply | {error,Error}
- Reply = term()
- Error = bad_module | {'EXIT',Reason} | term()
- Reason = term()
Makes a synchronous call to the event handler \texttt{Handler} installed in the event manager \texttt{EventMgrRef} by sending a request and waiting until a reply arrives or a timeout occurs. The event manager will call \texttt{Module:handle_call}/2 to handle the request.

See \texttt{add_handler}/3 for a description of \texttt{EventMgrRef} and \texttt{Handler}.

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

\texttt{Timeout} is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom \texttt{infinity} to wait indefinitely. Default value is 5000. If no reply is received within the specified time, the function call fails.

The return value \texttt{Reply} is defined in the return value of \texttt{Module:handle_call}/2. If the specified event handler is not installed, the function returns \{\texttt{error,\texttt{bad_module}}\}. If the callback function fails with \texttt{Reason} or returns an unexpected value \texttt{Term}, this function returns \{\texttt{error,\{\texttt{EXIT},\texttt{Reason}\}}\} or \{\texttt{error,\texttt{Term}}\}, respectively.

\texttt{delete_handler(EventMgrRef, Handler, Args) -&gt; Result}

\textbf{Types:}
- \texttt{EventMgrRef} = \texttt{Name} | \{\texttt{Name,Node} \} | \{\texttt{global,Name} \} | \texttt{pid()}
- \texttt{Name} = \texttt{Node} = \texttt{atom()}
- \texttt{Handler} = \texttt{Module} | \{\texttt{Module,Id} \}
- \texttt{Module} = \texttt{atom()}
- \texttt{Id} = \texttt{term()}
- \texttt{Args} = \texttt{term()}
- \texttt{Result} = \texttt{term() | \{error,module_not_found\} | \{\texttt{EXIT},\texttt{Reason}\}}
- \texttt{Reason} = \texttt{term()}

Deletes an event handler from the event manager \texttt{EventMgrRef}. The event manager will call \texttt{Module:terminate/2} to terminate the event handler.

See \texttt{add_handler}/3 for a description of \texttt{EventMgrRef} and \texttt{Handler}.

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

The return value is the return value of \texttt{Module:terminate/2}. If the specified event handler is not installed, the function returns \{\texttt{error,module_not_found}\}. If the callback function fails with \texttt{Reason}, the function returns \{\texttt{\texttt{'EXIT'},\texttt{Reason}}\}.

\texttt{swap_handler(EventMgrRef, \{Handler1,Args1\}, \{Handler2,Args2\}) -&gt; Result}

\textbf{Types:}
- \texttt{EventMgrRef} = \texttt{Name} | \{\texttt{Name,Node} \} | \{\texttt{global,Name} \} | \texttt{pid()}
- \texttt{Name} = \texttt{Node} = \texttt{atom()}
- \texttt{Handler1} = \texttt{Handler2} = \texttt{Module} | \{\texttt{Module,Id} \}
- \texttt{Module} = \texttt{atom()}
- \texttt{Id} = \texttt{term()}
- \texttt{Args1} = \texttt{Args2} = \texttt{term()}
- \texttt{Result} = \texttt{ok | \{error,Error\}}
- \texttt{Error} = \{\texttt{\texttt{'EXIT'},\texttt{Reason}}\} | \texttt{term()}
- \texttt{Reason} = \texttt{term()}

STDLIB Reference Manual
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.

\[
\text{swap_sup_handler(EventMgrRef, \{Handler1, Args1\}, \{Handler2, Args2\}) -> Result}
\]

\[
\text{Types:}
\]
\[
\text{\quad \cdot \ \text{EventMgrRef = Name} | \{Name, Node\} | \{global, Name\} | \text{pid()}}
\]
\[
\text{\quad \quad \text{Name = Node = atom()}}
\]
\[
\text{\quad \quad \text{Handler1 = Handler2 = Module} | \{Module, Id\}}
\]
\[
\text{\quad \quad \text{Module = atom()}}
\]
\[
\text{\quad \quad \text{Id = term()}}
\]
\[
\text{\quad \quad \text{Args1 = Args2 = term()}}
\]
\[
\text{\quad \quad \text{Result = ok | \{error, Error\}}} \quad \text{\quad \text{Error = ['EXIT', Reason] | term()}}
\]
\[
\text{\quad \quad \text{Reason = term()}}
\]

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.

\[
\text{which_handlers(EventMgrRef) -> \{Handler\}}
\]

\[
\text{Types:}
\]
\[
\text{\quad \cdot \ \text{EventMgrRef = Name} | \{Name, Node\} | \{global, Name\} | \text{pid()}}
\]
\[
\text{\quad \quad \text{Name = Node = atom()}}
\]
\[
\text{\quad \quad \text{Handler = Module} | \{Module, Id\}}
\]
\[
\text{\quad \quad \text{Module = atom()}}
\]
\[
\text{\quad \quad \text{Id = term()}}
\]

Returns a list of all event handlers installed in the event manager EventMgrRef.

See add_handler/3 for a description of EventMgrRef and Handler.
stop(EventMgrRef) -> ok

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

Terminates the event manager EventMgrRef. Before terminating, the event manager
will call Module:terminate(stop,...) for each installed event handler.

See add_handler/3 for a description of the argument.

CALLBACK FUNCTIONS

The following functions should be exported from a gen_event callback module.

Exports

Module:init(InitArgs) -> {ok,State}

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)`.

```erlang
Module:handle_call(Request, State) -> Result
```

Types:
- `Request` = `term()`
- `State` = `term()`
- `Result` = `{ok, Reply, NewState}`
  - `Reply` = `term()`
  - `NewState` = `term()`
- `{swap_handler, Reply, Args1, NewState, Handler2, Args2}`
- `{remove_handler, Reply}`
- `Reply` = `term()`
- `NewState` = `term()`
- `Args1` = `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`.

```erlang
Module:handle_info(Info, State) -> Result
```

Types:
- `Info` = `term()`
- `State` = `term()`
- `Result` = `{ok, NewState}`
  - `NewState` = `term()`
  - `Args1` = `term()`
  - `Handler2` = `Module2` | `{Module2, Id}`
  - `Module2` = `atom()`
  - `Id` = `term()`
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 338], sys(3) [page 348]
A behaviour module for implementing a finite state machine. A generic finite state machine process (gen_fsm) implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into an OTP supervision tree. Refer to [OTP Design Principles] for more information.

A gen_fsm assumes all specific parts to be located in a callback module exporting a pre-defined set of functions. The relationship between the behaviour functions and the callback functions can be illustrated as follows:

```
gen_fsm module              Callback module
--------------------          ---------------
gen_fsm:start_link          Module:init/1
-gen_fsm:send_event         Module:StateName/2
-gen_fsm:send_all_state_event Module:handle_event/3
-gen_fsm:sync_send_event    Module:StateName/3
-gen_fsm:sync_send_all_state_event Module:handle_sync_event/4
-                             Module:handle_info/3
-                             Module:terminate/3
-                             Module:code_change/4
```

If a callback function fails or returns a bad value, the gen_fsm will terminate. The sys module can be used for debugging a gen_fsm.

Note that a gen_fsm does not trap exit signals automatically, this must be explicitly initiated in the callback module.

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

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

Types:
- \(\text{FsmName} = \{\text{local,Name}\} \mid \{\text{global,GlobalName}\}\)
- \(\text{Name} = \text{atom}\()
- \(\text{GlobalName} = \text{term}\()
- \(\text{Module} = \text{atom}\()
- \(\text{Args} = \text{term}\()
- \(\text{Options} = [\text{Option}]\)
- \(\text{Option} = \{\text{debug,Dbgs} \mid \text{timeout,Time} \mid \{\text{spawn_opt},SOpts}\}\)
- \(\text{Dbgs} = [\text{Dbg}]\)
- \(\text{Dbg} = \text{trace} \mid \text{log} \mid \text{statistics}\)
- \(\{\text{log_to_file,FileName}\} \mid \{\text{install,\{Func,FuncState\}}\}\)
- \(\text{SOpts} = [\text{SOpt}]\)
- \(\text{SOpt} - \text{see erlang:spawn_opt/2,3,4,5}\)
- \(\text{Result} = \{\text{ok,Pid}\} \mid \text{ignore} \mid \{\text{error,Error}\}\)
- \(\text{Pid} = \text{pid}\()
- \(\text{Error} = \{\text{alreadystarted,Pid}\} \mid \text{term}\)

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 start-up procedure, \text{start_link/3,4} does not return until \text{Module:init/1} has returned.

If \text{FsmName} = \{\text{local,Name}\}, the \text{gen_fsm} is registered locally as \text{Name} using \text{register/2}.

If \text{FsmName} = \{\text{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}. See \text{sys(3)} [page 348].

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. See \text{erlang(3)}.

**Note:**
Using the spawn option \text{monitor} is currently not allowed, but will cause the function to fail with reason \text{badarg}.

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
FsmName, the function returns \{error,\{already_started,Pid\}\} where Pid is the pid of that process.

If Module:init/1 fails with Reason, the function returns \{error,Reason\}. If Module:init/1 returns \{stop,Reason\} or ignore, the process is terminated and the function returns \{error,Reason\} or ignore, respectively.

\[\text{start}(\text{Module}, \text{Args}, \text{Options}) \rightarrow \text{Result}\]
\[\text{start}(\text{FsmName}, \text{Module}, \text{Args}, \text{Options}) \rightarrow \text{Result}\]

Types:
- \text{FsmName} = \text{local},Name | \text{global},GlobalName
- Name = \text{atom}()
- GlobalName = \text{term}()
- Module = \text{atom}()
- Args = \text{term}()
- Options = [Option]
- Option = \{debug,D bgs\} | \{timeout,Time\} | \{spawn_opt,SOpts\}
- D bgs = [D bgs]
- D bgs = \{trace\} | \{log\} | \{statistics\}
- \text{SOpts} = [term()]
- Result = \{ok,Pid\} | ignore | \{error,Error\}
- Pid = \text{pid}()
- Error = \{already_started,Pid\} | term()

Creates a stand-alone \text{gen fsm} process, i.e. a \text{gen fsm} which is not part of a supervision tree and thus has no supervisor.

See \text{start_link/3,4} [page 184] for a description of arguments and return values.

\[\text{send_event}(\text{FsmRef}, \text{Event}) \rightarrow \text{ok}\]

Types:
- \text{FsmRef} = Name | \{Name,Node\} | \{global,GlobalName\} | \text{pid}()
- Name = \text{Node} = \text{atom}()
- GlobalName = \text{term}()
- Event = \text{term}()

Sends an event asynchronously to the \text{gen fsm} \text{FsmRef} and returns \text{ok} immediately. The \text{gen fsm} will call \text{Module:StateName/2} to handle the event, where \text{StateName} is the name of the current state of the \text{gen fsm}.

\text{FsmRef} can be:
- the \text{pid},
- Name, if the \text{gen fsm} is locally registered,
- \{Name,Node\}, if the \text{gen fsm} is locally registered at another node, or
- \{global,GlobalName\}, if the \text{gen fsm} is globally registered.

\text{Event} is an arbitrary term which is passed as one of the arguments to \text{Module:StateName/2}. 
send_all_state_event(FsmRef, Event) -> ok

Types:
  - FsmRef = Name | {Name,Node} | {global,GlobalName} | pid()
  - Name = Node = atom()
  - GlobalName = term()
  - Event = term()

Sends an event asynchronously to the gen_fsm FsmRef and returns ok immediately. The gen_fsm will call Module:handle_event/3 to handle the event.

See send_event/2 [page 185] 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.

csync_send_event(FsmRef, Event) -> Reply
csync_send_event(FsmRef, Event, Timeout) -> Reply

Types:
  - FsmRef = Name | {Name,Node} | {global,GlobalName} | pid()
  - Name = Node = atom()
  - GlobalName = term()
  - Event = term()
  - Timeout = int() > 0 | infinity
  - Reply = term()

Sends an event to the gen_fsm FsmRef and waits until a reply arrives or a timeout occurs. The gen_fsm will call Module:StateName/3 to handle the event, where StateName is the name of the current state of the gen_fsm.

See send_event/2 [page 185] 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.

csync_send_all_state_event(FsmRef, Event) -> Reply
csync_send_all_state_event(FsmRef, Event, Timeout) -> Reply

Types:
  - FsmRef = Name | {Name,Node} | {global,GlobalName} | pid()
  - Name = Node = atom()
- GlobalName = term()
- Event = term()
- Timeout = int(0 | infinity
- Reply = term()

Sends an event to the gen_fsm FsmRef and waits until a reply arrives or a timeout occurs. The gen_fsm will call Module:handle_sync_event/4 to handle the event.

See send_event/2 [page 185] for a description of FsmRef and Event. See sync_send_event/3 [page 186] for a description of Timeout and Reply.

See send_all_state_event/2 [page 186] for a description of the difference between sync_send_event and sync_send_all_state_event.

\[
\text{reply}(\text{Caller}, \text{Reply}) \rightarrow \text{true}
\]

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

This function can be used by a gen_fsm to explicitly send a reply to a client process that called sync_send_event/2,3 [page 186] or sync_send_all_state_event/2,3 [page 186], when the reply cannot be defined in the return value of Module:State/3 or Module:handle_sync_event/4.

Caller must be the From argument provided to the callback function. Reply is an arbitrary term, which will be given back to the client as the return value of sync_send_event/2,3 or sync_send_all_state_event/2,3.

\[
\text{send_event_after}(\text{Time}, \text{Event}) \rightarrow \text{Ref}
\]

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

Sends a delayed event internally in the gen_fsm that calls this function after Time ms. Returns immediately a reference that can be used to cancel the delayed send using cancel_timer/1 [page 188].

The gen_fsm will call Module:StateName/2 to handle the event, where StateName is the name of the current state of the gen_fsm at the time the delayed event is delivered. Event is an arbitrary term which is passed as one of the arguments to Module:StateName/2.

\[
\text{start_timer}(\text{Time}, \text{Msg}) \rightarrow \text{Ref}
\]

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 \[page 188\].}

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, \{\texttt{timeout, Ref, Msg}\}, as one of the arguments to \texttt{Module:StateName/2}.

cancel_timer(\texttt{Ref}) \rightarrow \text{RemainingTime | false}

Types:
\begin{itemize}
  \item \texttt{Ref = reference()}
  \item \texttt{RemainingTime = integer()}
\end{itemize}

Cancels an internal timer referred by \texttt{Ref} in the \texttt{gen_fsm} that calls this function. \texttt{Ref} is a reference returned from \texttt{send_event_after/2 \[page 187\]} or \texttt{start_timer/2 \[page 187\].}

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 \texttt{Ref} referred to an active timer, \texttt{false} otherwise.

\begin{verbatim}
enter_loop(Module, Options, StateName, StateData)
\end{verbatim}
\begin{verbatim}
enter_loop(Module, Options, StateName, StateData, FsmName)
\end{verbatim}
\begin{verbatim}
enter_loop(Module, Options, StateName, StateData, Timeout)
\end{verbatim}
\begin{verbatim}
enter_loop(Module, Options, StateName, StateData, FsmName, Timeout)
\end{verbatim}

Types:
\begin{itemize}
  \item \texttt{Module = atom()}
  \item \texttt{Options = [Option]}
  \item \texttt{Option = \{debug,Dbgs\}}
  \item \texttt{Dbgs = [Dbg]}
  \item \texttt{Dbg = trace | log | statistics}
  \item \texttt{LocalName = \{local,Name\} | \{global,GlobalName\}}
  \item \texttt{Name = atom()}
  \item \texttt{GlobalName = term()}
  \item \texttt{Timeout = int() | infinity}
\end{itemize}
Make 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 255]. 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 184]. 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 189]. 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**

\[
\text{Module:} \text{init} \left( \text{Args} \right) \rightarrow \text{Result}
\]

Types:
- \text{Args} = \text{term()}
- \text{Return} = \{ \text{ok,StateName,StateData} \} \mid \{ \text{ok,StateName,StateData,Timeout} \}
- \{ \text{stop,Reason} \} \mid \text{ignore}
- \text{StateName} = \text{atom()}
- \text{StateData} = \text{term()}
- \text{Timeout} = \text{int()} \mid \text{infinity}
- \text{Reason} = \text{term()}

Whenever a gen_fsm is started using gen_fsm:start/3,4 [page 185] or gen_fsm:start_link/3,4 [page 184], this function is called by the new process to initialize. \text{Args} is the \text{Args} argument provided to the start function.

If initialization is successful, the function should return \{ \text{ok,StateName,StateData} \} or \{ \text{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 
\{\text{stop}, \text{Reason}\}, where \text{Reason} is any term, or \text{ignore}.

\textbf{Module:StateName} (Event, StateName, StateData) \rightarrow \text{Result}

Types:
- Event = \text{timeout} \mid \text{term()}
- StateData = \text{term()}
- Result = \{\text{next_state}, NextStateName, NewStateData\} \mid 
  \{\text{next_state}, NextStateName, NewStateData, Timeout\}
- \{\text{stop}, Reason, NewStateData\}
- NextStateName = \text{atom()}
- NewStateData = \text{term()}
- Timeout = \text{int()} > 0 \mid \text{infinity}
- Reason = \text{term()}

There should be one instance of this function for each possible state name. Whenever a \text{gen fsm} receives an event sent using \text{gen fsm:send_event/2} [page 185], 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.

\text{Event} is either the atom \text{timeout}, if a timeout has occurred, or the \text{Event} argument provided to \text{send_event/2}.

\text{StateData} is the state data of the \text{gen fsm}.

If the function returns \{\text{next_state}, NextStateName, NewStateData\} or 
\{\text{next_state}, NextStateName, NewStateData, Timeout\}, the \text{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 \{\text{stop}, Reason, NewStateData\}, the \text{gen fsm} will call Module:terminate(Reason, NewStateData) and terminate.

\textbf{Module:handle_event} (Event, StateName, StateData) \rightarrow \text{Result}

Types:
- Event = \text{term()}
- StateName = \text{atom()}
- StateData = \text{term()}
- Result = \{\text{next_state}, NextStateName, NewStateData\} \mid 
  \{\text{next_state}, NextStateName, NewStateData, Timeout\}
- \{\text{stop}, Reason, NewStateData\}
- NextStateName = \text{atom()}
- NewStateData = \text{term()}
- Timeout = \text{int()} > 0 \mid \text{infinity}
- Reason = \text{term()}

Whenever a \text{gen fsm} receives an event sent using \text{gen fsm:send_all_state_event/2} [page 186], this function is called to handle the event.

\text{StateName} is the current state name of the \text{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 
  Reason = normal | term() 

There should be one instance of this function for each possible state name. Whenever a gen_fsm receives an event sent using gen_fsm:sync_send_event/2,3 [page 186], the instance of this function with the same name as the current state name StateName is called to handle the event.

Event is the Event argument provided to sync_send_event.

From is a tuple {Pid,Tag} where Pid is the pid of the process which called sync_send_event/2,3 and Tag is a unique tag.

StateData is the state data of the gen_fsm.

If the function returns {reply,Reply,NextStateName,NewStateData} or 
{reply,Reply,NextStateName,NewStateData,Timeout}, Reply will be given back to From as the return value of sync_send_event/2,3. The gen_fsm then continues executing with the current state name set to NextStateName and with the possibly updated state data NewStateData. See Module:init/1 for a description of Timeout.

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 [page 187].

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}
Whenever a gen_fsm receives an event sent using gen_fsm:sync_send_all_state_event/2,3 [page 186], 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} | {stop, Reason, NewStateData, Timeout}
- NextStateName = atom()
- NewStateData = term()
- Timeout = int() = 0 | infinity
- Reason = 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 superviser to terminate, this function will be called with Reason=shutdown if the following conditions apply:

- the gen_fsm has been set to trap exit signals, and
- the shutdown strategy as defined in the supervisor’s child specification is an integer timeout value, not brutal_kill.

Otherwise, the gen_fsm will be immediately terminated.

Note that for any other reason than normal or shutdown, the gen_fsm is assumed to terminate due to an error and an error report is issued using [error_logger:format/2].

Module:code_change(OldVsn, StateName, StateData, Extra) -> {ok, NextStateName, NewStateData}

Types:
- OldVsn = Vsn | {down, Vsn}
- Vsn = term()
- StateName = NextStateName = atom()
- StateData = NewStateData = term()
- Extra = term()

This function is called by a gen_fsm when it should update its internal state data during a release upgrade/downgrade, i.e. when the instruction {update, Module, Change,...} where Change={advanced,Extra} is given in the appup file. See [OTP Design Principles].

In the case of an upgrade, OldVsn is Vsn, and in the case of a downgrade, OldVsn is {down, Vsn}. Vsn is defined by the vsn attribute(s) of the old version of the callback module Module. If no such attribute is defined, the version is the checksum of the BEAM file.

StateName is the current state name and StateData the internal state data of the gen_fsm.

Extra is passed as-is from the {advanced,Extra} part of the update instruction.

The function should return the new current state name and updated internal data.

SEE ALSO

gen_event(3) [page 173], gen_server(3) [page 194], supervisor(3) [page 338],
proc_lib(3) [page 255], sys(3) [page 348]
**gen_server**

Erlang Module

A behavior 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 anOTP 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 behavior functions and the callback functions can be illustrated as follows:

```
gen_server module                  Callback module
-----------------------------------
gen_server:start_link    ----> Module:init/1

gen_server:call

gen_server:multi_call      ----> Module:handle_call/3

gen_server:cast

gen_server:abcast         ----> Module:handle_cast/2

-                         ----> Module:handle_info/2
-                         ----> Module:terminate/2
-                         ----> Module:code_change/3
```

If a callback function fails or returns a bad value, the gen_server will terminate.

The sys module can be used for debugging a gen_server.

Note that a gen_server does not trap exit signals automatically, this must be explicitly initiated in the callback module.

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

**Exports**

- `start_link(Module, Args, Options) -> Result`
- `start_link(ServerName, Module, Args, Options) -> Result`

**Types:**
- `ServerName = {local, Name} | {global, GlobalName}`
- `Name = atom()`
- `GlobalName = term()`
- `Module = atom()`
- `Args = term()`
- `Options = [Option]`
- `Option = [debug,Dbgs] | [timeout,Time] | [spawn_opt,SOpts]`
- `Dbgs = [Dbg]`
- `Dbg = [trace | log | statistics | [log_to_file,FileName] | [install,{Func,FuncState}]]
- `SOpts = [term()]`
- `Result = [ok,Pid] | ignore | [error,Error]`
- `Pid = pid()`
- `Error = [already_started,Pid] | term()`

Creates a gen_server process as part of a supervision tree. The function should be called, directly or indirectly, by the supervisor. It will, among other things, ensure that the gen_server is linked to the supervisor.

The gen_server process calls `Module:init/1` to initialize. To ensure a synchronized start-up procedure, `start_link/3,4` does not return until `Module:init/1` has returned.

If `ServerName=local,Name` the gen_server is registered locally as `Name` using `register/2`. If `ServerName=global,GlobalName` the gen_server is registered globally as `GlobalName` using `global:register_name/2`. If no name is provided, the gen_server is not registered.

`Module` is the name of the callback module.

`Args` is an arbitrary term which is passed as the argument to `Module:init/1`.

If the option `{timeout,Time}` is present, the gen_server is allowed to spend `Time` milliseconds initializing or it will be terminated and the start function will return `{error,timeout}`.

If the option `{debug,Dbgs}` is present, the corresponding `sys` function will be called for each item in `Dbgs`. See `sys(3)` [page 348].

If the option `{spawn_opt,SOpts}` is present, `SOpts` will be passed as option list to the `spawn_opt` BIF which is used to spawn the gen_server. See [erlang(3)].

**Note:**
Using the spawn option `monitor` is currently not allowed, but will cause the function to fail with reason `badarg`.

If the gen_server is successfully created and initialized the function returns `{ok,Pid}`, where `Pid` is the pid of the gen_server. If there already exists a process with the specified `ServerName` the function returns `{error,[already_started,Pid]}`, where `Pid` is the pid of that process.

If `Module:init/1` fails with `Reason`, the function returns `{error,Reason}`. If `Module:init/1` returns `{stop,Reason}` or `ignore`, the process is terminated and the function returns `{error,Reason}` or `ignore`, respectively.

```
start(Module, Args, Options) -> Result
start(ServerName, Module, Args, Options) -> Result
```

Types:
- ServerName = \{local,Name\} | \{global,GlobalName\}
- Name = atom()
- GlobalName = term()
- Module = atom()
- Args = term()
- Options = \[Option\]
  - Option = \{debug,Dbgs\} | \{timeout,Time\} | \{spawn_opt,SOpts\}
- Dbgs = \[Dbg\]
  - Dbg = trace | log | statistics | \{log_to_file,FileName\} | \{install,\{Func,FuncState\}\}
  - SOpts = \[term()\]
- Result = \{ok,Pid\} | ignore | \{error,Error\}
- Pid = pid()
- Error = \{already_started,Pid\} | term()

Creates a stand-alone gen_server process, i.e. a gen_server which is not part of a supervision tree and thus has no supervisor. See start_link/3,4 [page 194] for a description of arguments and return values.

call(ServerRef, Request) -> Reply

call(ServerRef, Request, Timeout) -> Reply

Types:
- ServerRef = Name | \{Name,Node\} | \{global,GlobalName\} | pid()
- Node = atom()
- GlobalName = term()
- Request = term()
- Timeout = int() > 0 | infinity
- Reply = term()

Makes a synchronous call to the gen_server ServerRef by sending a request and waiting until a reply arrives or a timeout occurs. The gen_server will call Module:handle_call/3 to handle the request.

ServerRef can be:
- the pid,
- Name, if the gen_server is locally registered,
- \{Name,Node\}, if the gen_server is locally registered at another node, or
- \{global,GlobalName\}, if the gen_server is globally registered.

Request is an arbitrary term which is passed as one of the arguments to Module:handle_call/3.

Timeout is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom infinity to wait indefinitely. Default value is 5000. If no reply is received within the specified time, the function call fails.

The return value Reply is defined in the return value of Module:handle_call/3.

The call may fail for several reasons, including timeout and the called gen_server dying before or during the call.

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.

\textbf{multi_call(Name, Request) -> Result} \\
\textbf{multi_call(Nodes, Name, Request) -> Result} \\
\textbf{multi_call(Nodes, Name, Request, Timeout) -> Result}

\textbf{Types:}
- \textit{Nodes} = \[\text{Node}\]
- \textit{Node} = \text{atom()}
- \textit{Name} = \text{atom()}
- \textit{Request} = \text{term()}
- \textit{Timeout} = \text{int()} \geq 0 | \text{infinity}
- \textit{Result} = \{\text{Replies,BadNodes}\}
- \textit{Replies} = \{(\text{Node,Reply})\}
- \textit{Reply} = \text{term()}
- \textit{BadNodes} = \{\text{Node}\}

Makes a synchronous call to all gen_servers locally registered as \textit{Name} at the specified nodes by first sending a request to every node and then waiting for the replies. The gen_servers will call \text{Module:handle_call/3} to handle the request.

The function returns a tuple \{(\text{Replies,BadNodes})\} where \textit{Replies} is a list of \{(\text{Node,Reply})\} and \textit{BadNodes} is a list of node that either did not exist, or where the gen_server \textit{Name} did not exist or did not reply.

\textit{Nodes} is a list of node names to which the request should be sent. Default value is the list of all known nodes \{\text{node()}, \text{nodes()}\}.

\textit{Name} is the locally registered name of each gen_server.

\textit{Request} is an arbitrary term which is passed as one of the arguments to \text{Module:handle_call/3}.

\textit{Timeout} is an integer greater than zero which specifies how many milliseconds to wait for each reply, or the atom \text{infinity} to wait indefinitely. Default value is \text{infinity}. If no reply is received from a node within the specified time, the node is added to \textit{BadNodes}.

When a reply \textit{Reply} is received from the gen_server at a node \textit{Node}, \{(\text{Node,Reply})\} is added to \textit{Replies}. \textit{Reply} is defined in the return value of \text{Module:handle_call/3}.

\textbf{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 \textit{Timeout}, which may be \text{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` [page 196] for a description of `ServerRef`.

`Request` is an arbitrary term which is passed as one of the arguments to `Module:handle_cast/2`.

**abcast(Name, Request) -> abcast**

**abcast(Nodes, Name, Request) -> abcast**

Types:
- `Nodes = [Node]`
- `Node = atom()`
- `Name = atom()`
- `Request = term()`

Sends an asynchronous request to the `gen_servers` locally registered as `Name` at the specified nodes. The function returns immediately and ignores nodes that does not exist, or where the `gen_server` `Name` does not exist. The `gen_servers` will call `Module:handle_cast/2` to handle the request.

See `multi_call/2,3,4` [page 197] 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/2,3` or `multi_call/2,3,4`, when the reply cannot be defined in the return value of `Module:handle_call/3`.

`Client` must be the `From` argument provided to the callback function. `Reply` is an arbitrary term, which will be given back to the client as the return value of `call/2,3` or `multi_call/2,3,4`. 
enter_loop(Module, Options, State)
enter_loop(Module, Options, State, ServerName)
enter_loop(Module, Options, State, Timeout)
enter_loop(Module, Options, State, ServerName, Timeout)

Types:
- Module = atom()
- Options = [Option]
- Option = {debug,Dbgs}
- Dbgs = [Dbg]
- Dbg = trace | log | statistics
- | {log_to_file,FileName} | {install,Func,FuncState}
- State = term()
- ServerName = {local,Name} | {global,GlobalName}
- Name = atom()
- GlobalName = term()
- Timeout = int() | infinity

Makes an existing process into a gen_server. Does not return, instead the calling process will enter the gen_server receive loop and become a gen_server process. The process must have been started using one of the start functions in proc_lib, see proc_lib(3) [page 255]. 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 194]. 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 200]. Also, the callback module Module does not need to export an init/1 function.

Failure: If the calling process was not started by a proc_lib start function, or if it is not registered according to ServerName.

CALLBACK FUNCTIONS

The following functions should be exported from a gen_server callback module.
Exports

Module:init(Args) -> Result

Types:
- Args = term()
- Result = {ok,State} | {ok,State,Timeout} |
  {stop,Reason} | ignore
- State = term()
- Timeout = int() => 0 | infinity
- Reason = term()

Whenever a gen_server is started using gen_server:start/3,4 [page 195] or
gen_server:start_link/3.4 [page 194], this function is called by the new process to
initialize.

Args is the Args argument provided to the start function.

If the initialization is successful, the function should return {ok,State} or
{oek,State,Timeout}, where State is the internal state of the gen_server.

If an integer timeout value is provided, a timeout will occur unless a request or a message
is received within Timeout milliseconds. A timeout is represented by the atom timeout
which should be handled by the handle_info/2 callback function. The atom
infinity can be used to wait indefinitely, this is the default value.

If something goes wrong during the initialization the function should return
{stop,Reason} where Reason is any term, or ignore.

Module:handle_call(Request, From, State) -> Result

Types:
- Request = term()
- From = {pid(),Tag}
- State = term()
- Result = {reply,Reply,NewState} | {reply,Reply,NewState,Timeout} |
  {noreply,NewState} | {noreply,NewState,Timeout} |
  {stop,Reason,Reply,NewState} | {stop,Reason,NewState}
- Reply = term()
- NewState = term()
- Timeout = int() => 0 | infinity
- Reason = term()

Whenever a gen_server receives a request sent using gen_server:call/2,3 [page 196] or
gen_server:multi_call/2,3,4 [page 197], this function is called to handle the request.

Request is the Request argument provided to call or multi_call.

From is a tuple {Pid,Tag} where Pid is the pid of the client and Tag is a unique tag.

State is the internal state of the gen_server.

If the function returns {reply,Reply,NewState} or
{noreply,Reply,NewState} or included in the return value of multi_call/2,3,4. The gen_server
then continues executing with the possibly updated internal state NewState. See Module:init/1 for a description of Timeout.

If the functions returns \{noreply,NewState\} or \{noreply,NewState,Timeout\}, the gen_server will continue executing with NewState. Any reply to From must be given explicitly using gen_server:reply/2 [page 198].

If the function returns \{stop,Reason,Reply,NewState\}, Reply will be given back to From. If the function returns \{stop,Reason,NewState\}, any reply to From must be given explicitly using gen_server:reply/2. The gen_server will then call Module:terminate(Reason,NewState) and terminate.

Module:handle_cast(Request, State) -> Result

Types:
- Request = term()
- State = term()
- Result = \{noreply,NewState\} | \{noreply,NewState,Timeout\}
- \{stop,Reason,NewState\}
- NewState = term()
- Timeout = int() \(\geq 0\) | infinity
- Reason = term()

Whenever a gen_server receives a request sent using gen_server:cast/2 [page 198] or gen_server:abcast/2,3 [page 198], 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() \(\geq 0\) | infinity
- Reason = normal | term()

This function is called by a gen_server when a timeout occurs or when it receives any other message than a synchronous or asynchronous request (or a system message). Info is either the atom timeout, if a timeout has occurred, or the received message. See Module:handle_call/3 for a description of the other arguments and possible return values.

Module:terminate(Reason, State)

Types:
- Reason = normal | shutdown | term()
- State = term()
This function is called by a gen_server when it is about to terminate. It should be the opposite of Module:init/1 and do any necessary cleaning up. When it returns, the gen_server terminates with Reason. The return value is ignored.

Reason is a term denoting the stop reason and State is the internal state of the gen_server.

Reason depends on why the gen_server is terminating. If it is because another callback function has returned a stop tuple {stop,...}, Reason will have the value specified in that tuple. If it is due to a failure, Reason is the error reason.

If the gen_server is part of a supervision tree and is ordered by its supervisor to terminate, this function will be called with Reason=shutdown if the following conditions apply:

- the gen_server has been set to trap exit signals, and
- the shutdown strategy as defined in the supervisor’s child specification is an integer timeout value, not brutal kill.

Otherwise, the gen_server will be immediately terminated.

Note that for any other reason than normal or shutdown, the gen_server is assumed to terminate due to an error and an error report is issued using [error_logger:format/2].

Module:code_change(OldVsn, State, Extra) -> {ok, NewState}

Types:
- OldVsn = Vsn | {down, Vsn}
- Vsn = term()
- State = NewState = term()
- Extra = term()

This function is called by a gen_server when it should update its internal state during a release upgrade/downgrade, i.e. when the instruction {update, Module, Change,...} where Change={advanced, Extra} is given in the appup file. See [OTP Design Principles] for more information.

In the case of an upgrade, OldVsn is Vsn, and in the case of a downgrade, OldVsn is {down, Vsn}. Vsn is defined by the vsn attribute(s) of the old version of the callback module Module. If no such attribute is defined, the version is the checksum of the BEAM file.

State is the internal state of the gen_server.

Extra is passed as-is from the {advanced, Extra} part of the update instruction.

The function should return the updated internal state.

SEE ALSO

gen_event(3) [page 173], gen_fsm(3) [page 183], supervisor(3) [page 338], proc_lib(3) [page 255], sys(3) [page 348]
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 = iodata()` – see `erlang(3)`

Writes the characters of `IoData` to the standard output (`IoDevice`).

`nl([IoDevice]) -> ok`

Types:
- `IoDevice = io_device()`

Writes new line to the standard output (`IoDevice`).

`get_chars([IoDevice,] Prompt, Count) -> string() | eof`

Types:
- `IoDevice = io_device()`
- `Prompt = atom() | string()`
- `Count = int()`

Reads `Count` characters from standard input (`IoDevice`), prompting it with `Prompt`. It returns:
String  The input characters.
eof  End of file was encountered.

get_line([IoDevice,] Prompt) -> string() | eof

Types:
- IoDevice = io_device()
- Prompt = atom() | string()

Reads a line from the standard input (IoDevice), prompting it with Prompt. It returns:

String  The characters in the line terminated by a LF (or end of file).
eof  End of file was encountered.

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

Types:
- IoDevice = io_device()
- Opts = [Opt]
- Opt = binary | list
- Reason = term()

Set options for standard input/output (IoDevice). Possible options are:

binary  Makes get_chars/2,3 and get_line/1,2 return binaries instead of lists of
chars.
list  Makes get_chars/2,3 and get_line/1,2 return lists of chars, which is the
default.

expand_fun  Provide a function for tab-completion (expansion) like the erlang shell.

This function is called when the user presses the Tab key. The expansion is active
when calling line-reading functions such as get_line/1,2.

The function is called with the current line, upto the cursor, as a reversed string. It
should return a three-tuple: {yes|no, string(), [string(), ...]}. The first
element gives a beep if no, otherwise the expansion is silent, the second is a string
that will be entered at the cursor position, and the third is a list of possible
expansions. If this list is non-empty, the list will be printed and the current input
line will be written once again.

Trivial example (beep on anything except empty line, which is expanded to “quit”):

fun("") -> {yes, "quit", []};
(._) -> {no, ",", ["quit"]} end

Note:
The binary option does not work against IO servers on remote nodes running an
older version of Erlang/OTP than R9C.

write([IoDevice,] Term) -> ok
Types:
- IoDevice = io_device()
- Term = term()

Writes the term Term to the standard output (IoDevice).

read([IoDevice], Prompt) -> Result

Types:
- IoDevice = io_device()
- Prompt = atom() | string()
- Result = {ok, Term} | {eof} | {error, ErrorInfo}
- Term = term()
- ErrorInfo - see section Error Information below

Reads a term Term from the standard input (IoDevice), prompting it with Prompt. It returns:

- {ok, Term} The parsing was successful.
- {eof} End of file was encountered.
- {error, ErrorInfo} The parsing failed.

read(IoDevice, Prompt, StartLine) -> Result

Types:
- IoDevice = io_device()
- Prompt = atom() | string()
- StartLine = int()
- Result = {ok, Term, EndLine} | {eof, EndLine} | {error, ErrorInfo, EndLine}
- Term = term()
- EndLine = int()
- ErrorInfo - see section Error Information below

Reads a term Term from IoDevice, prompting it with Prompt. Reading starts at line number StartLine. It returns:

- {ok, Term, EndLine} The parsing was successful.
- {eof, EndLine} End of file was encountered.
- {error, ErrorInfo, EndLine} The parsing failed.

fwrite(Format) -> ok
fwrite([IoDevice], Format, Data) -> ok
format(Format) -> ok
format([IoDevice], Format, Data) -> ok

Types:
- IoDevice = io_device()
- Format = atom() | string() | binary()
- Data = [term()]
Wrote the items in `Data [[]]` on the standard output (`IoDevice`) in accordance with `Format`. `Format` contains plain characters which are copied to the output device, and control sequences for formatting, see below. If `Format` is an atom or a binary, it is first converted to a list with the aid of `atom_to_list/1` or `binary_to_list/1`.

```erlang
1> io:fwrite("Hello world!\n", []).
Hello world!
ok
```

The general format of a control sequence is `~F.P.PadC`. The character `C` determines the type of control sequence to be used, `F` and `P` are optional numeric arguments. If `F`, `P`, or `Pad` is `*`, the next argument in `Data` is used as the numeric value of `F` or `P`.

`F` is the field width of the printed argument. A negative value means that the argument will be left justified within the field, otherwise it will be right justified. If no field width is specified, the required print width will be used. If the field width specified is too small, then the whole field will be filled with `*` characters.

`P` is the precision of the printed argument. A default value is used if no precision is specified. The interpretation of precision depends on the control sequences. Unless otherwise specified, the argument within is used to determine print width.

`Pad` is the padding character. This is the character used to pad the printed representation of the argument so that it conforms to the specified field width and precision. Only one padding character can be specified and, whenever applicable, it is used for both the field width and precision. The default padding character is `' '` (space).

The following control sequences are available:

- The character `~` is written.
- 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:

```erlang
2> io:fwrite("\~10.5c\~10.5c\~5c\~\n", ['$a', '$b', '$c']).
| aaaaa| bbbbb | cccc |
ok
```

- The argument is a float which is written as `[-]ddd.d`, where the precision is the number of digits after the decimal point. The default precision is 6 and it cannot be less than 1.
- 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.
- The argument is a float which is written as `f`, if it is `>= 0.1` and `< 10000.0`. Otherwise, it is written in the `e` format. The precision is the number of significant digits. It defaults to 6 and should not be less than 2. If the absolute value of the float does not allow it to be written in the `f` format with the desired number of significant digits, it is also written in the `e` format.
- Prints the argument with the string syntax. The argument is an `[I/O list]`, a binary, or an atom. The characters are printed without quotes. In this format, the printed argument is truncated to the given precision and field width. This format can be used for printing any object and truncating the output so it fits a specified field:
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 g format.

prints the data with standard syntax in the same way as ~w, but breaks terms whose printed representation is longer than one line into many lines and indents each line sensibly. It also tries to detect lists of printable characters and to output these as strings. For example:

```
5> T = [{attributes,[[id,age,1.50000],[mode,explicit],
                     {typename,"INTEGER"}],
             [id,cho],[mode,explicit],[{typename,'Cho'}]],
             {typename,'Person'},
             tag,'PRIVATE',3},
             [mode,implicit}].
6> io:fwrite("~w~n", [T]).
   [{attributes,[[id,age,1.50000],[mode,explicit],
                 {typename,"INTEGER"}],
             [id,cho],[mode,explicit],[{typename,'Cho'}]],
             {typename,'Person'},
             tag,'PRIVATE',3},
             [mode,implicit}]
ok
7> io:fwrite("~62p~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 io:fwrite or io:format. For example, using T above:

```
8> io:fwrite("Here T = ~62p~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

\section*{W}

W writes data in the same way as \~w, but takes an extra argument which is the maximum depth to which terms are printed. Anything below this depth is replaced with \ldots\, For example, using T above:

\begin{verbatim}
9> io:fwrite("W-n", [T,9]).
[[attributes,[[[id,age,1.50000],[mode,explicit],[typename, ...]],[[id,cho],[mode,...],[...]]],[typename,'Person'],{tag,{PRIVATE',3}},{mode,implicit}]
ok
\end{verbatim}

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.

\section*{P}

P writes data in the same way as \~p, but takes an extra argument which is the maximum depth to which terms are printed. Anything below this depth is replaced with \ldots\, For example:

\begin{verbatim}
10> io:fwrite("-62P-n", [T,9]).
[[attributes,[[id,age,1.50000]],
    {mode,explicit},
    {typename,...}],
    {typename,'Person'},
    {tag,{PRIVATE',3}},
    {mode,implicit}]
ok
\end{verbatim}

\section*{B}

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:fwrite("-.16B-n", [31]).
1F
12> io:fwrite("-.2B-n", [-19]).
-10011
13> io:fwrite("-.36B-n", [5*36+35]).
5Z
ok
\end{verbatim}

\section*{X}

X like B, but takes an extra argument that is a prefix to insert before the number, but after the leading dash, if any. The prefix can be a possibly deep list of characters or an atom.
Like B, but prints the number with an Erlang style '#'-separated base prefix.

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:

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()
Reads characters from the standard input (IoDevice), prompting it with Prompt.
Interprets the characters in accordance with Format. Format contains control sequences
which directs the interpretation of the input.

Format may contain:

- White space characters (SPACE, TAB and NEWLINE) which cause input to be
  read to the next non-white space character.
- Ordinary characters which must match the next input character.
- Control sequences, which have the general format "*FC. The character * is an
  optional return suppression character. It provides a method to specify a field which
  is to be omitted. F is the field width of the input field and C determines the type
  of control sequence.

Unless otherwise specified, leading white-space is ignored for all control sequences.
An input field cannot be more than one line wide. The following control sequences
are available:

- A single ~ is expected in the input.
- 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.
- f 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 a, 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 a. 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:
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"s10c"c").
enter> alan_joe
{ok, ["alan", "joe"]}

scan_erl_exprs(Prompt) ->
scan_erl_exprs([IoDevice,] Prompt, StartLine) -> Result

Types:
- IoDevice = io_device()
- Prompt = atom() | string()
- StartLine = int()
- Result = {ok, Tokens, EndLine} | {eof, EndLine} | {error, ErrorInfo, EndLine}
- Tokens - see erl_scan(3)
- EndLine = int()
- ErrorInfo - see section Error Information below

Reads data from the standard input (IoDevice), prompting it with Prompt. Reading
starts at line number StartLine (1). The data is tokenized as if it were a sequence of
Erlang expressions until a final '.' is reached. This token is also returned. It returns:

{ok, Tokens, EndLine} The tokenization succeeded.
{eof, EndLine} End of file was encountered.
{error, ErrorInfo, EndLine} An error occurred.

Example:
23> io:scan_erl_exprs('enter>').
enter> abc(), "hey".
{ok, [{atom,1,abc}, {'(',1}, {')',1}, {',',1}, {string,1,"hey"}, {.,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 expressions/1,2,3 above.

parse Erl expressions(Prompt) ->
parse Erl expressions([IoDevice,] Prompt, StartLine) -> Result

Types:
- IoDevice = io_device()
- Prompt = atom() | string()
- StartLine = int()
- Result = {ok, Expr_list, EndLine} | {eof, EndLine} | {error, ErrorInfo, EndLine}
- Expr_list – see erl parse(3)
- EndLine = int()
- ErrorInfo – see section Error Information below

Reads data from the standard input (IoDevice), prompting it with Prompt. Starts reading at line number StartLine (1). The data is tokenized and parsed as if it were a sequence of Erlang expressions until a final '.' is reached. It returns:

{ok, Expr_list, EndLine} The parsing was successful.
{eof, EndLine} End of file was encountered.
{error, ErrorInfo, EndLine} An error occurred.

Example:
25> io:parse Erl expressions('enter>').
enter> abc(), "hey".
{ok, [[call,1,atom,1,abc],[[]],[string,1,"hey"]],2}
26> io:parse Erl expressions('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: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 = atom() | string() | binary()
- Data = [term()]

Returns a character list which represents Data formatted in accordance with Format. See io:fwrite/1,2,3 [page 205] 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 209] 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-5,24.5000],[]}
Continuation = see below
String = Format = string()
Return = \{ done, Result, LeftOverChars \} \| \{ more, Continuation \}
Result = \{ ok, InputList \} \| \text{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.
\text{eof} End of file has been encountered. LeftOverChars are the input characters not
used.
\{ error, What \} An error occurred and the parameter What gives a hint about the
error.
\{ more, Continuation \} More data is required to build a term. Continuation must be
passed to fread/3, when more data becomes available.

write_atom(Atom) -> chars()
Types:
\* Atom = atom()

Returns the list of characters needed to print the atom Atom.

write_string(String) -> chars()
Types:
\* String = string()

Returns the list of characters needed to print String as a string.

write_char(Integer) -> chars()
Types:
\* Integer = int()

Returns the list of characters needed to print a character constant.

indentation(String, StartIndent) -> int()
Types:
\* String = string()
\* StartIndent = int()
Returns the indentation if String has been printed, starting at StartIndent.

\texttt{char\_list(Term) -> bool()}

Types:
- \texttt{Term = term()}

Returns \texttt{true} if \texttt{Term} is a flat list of characters, otherwise it returns \texttt{false}.

\texttt{deep\_char\_list(Term) -> bool()}

Types:
- \texttt{Term = term()}

Returns \texttt{true} if \texttt{Term} is a, possibly deep, list of characters, otherwise it returns \texttt{false}.

\texttt{printable\_list(Term) -> bool()}

Types:
- \texttt{Term = term()}

Returns \texttt{true} if \texttt{Term} is a flat list of printable characters, otherwise it returns \texttt{false}. 
lib

Erlang Module

Warning:
This module is retained for compatibility. It may disappear without warning in a future release.

Exports

flush_receive() -> void()

Flushes the message buffer of the current process.

error_message(Format, Args) -> ok

Types:
- Format = atom() | string() | binary()
- Args = [term()]

Prints error message Args in accordance with Format. Similar to io:format/2, see io(3) [page 205].

proname() -> atom()

Returns the name of the script that started the current Erlang session.

nonl(String1) -> String2

Types:
- String1 = String2 = string()

Removes the last newline character, if any, in String1.

send(To, Msg)

Types:
- To = pid() | Name | {Name,Node}
- Name = Node = atom()
- Msg = term()

This function to makes it possible to send a message using the apply/3 BIF.
sendw(To, Msg)

Types:
- To = pid() | Name | {Name,Node}
- Name = Node = atom()
- Msg = term()

As send/2, but waits for an answer. It is implemented as follows:

```erlang
sendw(To, Msg) ->
    To ! {self(),Msg},
    receive
        Reply -> Reply
    end.
```

The message returned is not necessarily a reply to the message sent.
This module contains functions for list processing. The functions are organized in two groups: those in the first group perform a particular operation on one or more lists, whereas those in the second group are higher-order functions, using a fun as argument to perform an operation on one list.

Unless otherwise stated, all functions assume that position numbering starts at 1. That is, the first element of a list is at position 1.

Exports

```erlang
append(ListOfLists) -> List1
Types:
  • ListOfLists = [List]
  • List = List1 = [term()]
Returns a list in which all the sub-lists of ListOfLists have been appended. For example:
  > lists:append([[1, 2, 3], [a, b], [4, 5, 6]]).
  [1,2,3,a,b,4,5,6]
```

```erlang
append(List1, List2) -> List3
Types:
  • List1 = List2 = List3 = [term()]
Returns a new list List3 which is made from the elements of List1 followed by the elements of List2. For example:
  > lists:append("abc", "def").
  "abcdef"
```

```erlang
lists:append(A, B) is equivalent to A ++ B.
```

```erlang
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.

```prolog
> lists:concat([doc, '/', file, '.', 3]).
"doc/file.3"
```

delete(Elem, List1) -> List2

Types:
- Elem = term()
- List1 = List2 = [term()]

Returns a copy of `List1` where the first occurrence of `Elem`, if present, is deleted.

duplicate(N, Elem) -> List

Types:
- N = int()
- Elem = term()
- List = [term()]

Returns a list which contains `N` copies of the term `Elem`. For example:

```prolog
> 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 = [term() | DeepList]
- List = [term()]

Returns a flattened version of `DeepList`.

flatten(DeepList, Tail) -> List

Types:
- DeepList = [term() | DeepList]
- Tail = List = [term()]

Returns a flattened version of `DeepList` with the tail `Tail` appended.

keydelete(Key, N, TupleList1) -> TupleList2

Types:
- Key = term()
- N = 1..size(Tuple)
- TupleList1 = TupleList2 = [Tuple]
- Tuple = tuple()

Returns a copy of TupleList1 where the first occurrence of a tuple whose Nth element is Key is deleted, if present.

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:
last(List) -> Last
Types:
- List = [term()], length(List) > 0
- Last = term()
Returns the last element in List.

max(List) -> Max
Types:
- List = [term()], length(List) > 0
- Max = term()
Returns the maximum element of List.

member(Elem, List) -> bool()
Types:
- Elem = term()
- List = [term()]
Returns true if Elem is an element of List, otherwise false.

merge(ListOfLists) -> List1
Types:
- ListOfLists = [List]
- List = List1 = [term()]
Returns the sorted list formed by merging all the sub-lists of ListOfLists. All sub-lists must be sorted prior to evaluating this function.

merge(List1, List2) -> List3
Types:
- List1 = List2 = List3 = [term()]
Returns the sorted list formed by merging List1 and List2. Both List1 and List2 must be sorted prior to evaluating this function.

merge(Fun, List1, List2) -> List3
Types:
- Fun = fun(A, B) -> bool()
- List1 = [A]
- List2 = [B]
- List3 = [A | B]
- \( A = B = \text{term()} \)

Returns the sorted list formed by merging \( \text{List1} \) and \( \text{List2} \). Both \( \text{List1} \) and \( \text{List2} \) must be sorted according to the ordering function \( \text{Fun} \) prior to evaluating this function. \( \text{Fun}(A, B) \) should return \text{true} if \( A \) comes before \( B \) in the ordering, \text{false} otherwise.

\text{merge3}(\text{List1}, \text{List2}, \text{List3}) \rightarrow \text{List4}

Types:
- \( \text{List1} = \text{List2} = \text{List3} = \text{List4} = \text{[term()]} \)

Returns the sorted list formed by merging \( \text{List1}, \text{List2} \) and \( \text{List3} \). All of \( \text{List1}, \text{List2} \) and \( \text{List3} \) must be sorted prior to evaluating this function.

\text{min}(\text{List}) \rightarrow \text{Min}

Types:
- \( \text{List} = \text{[term()]}, \text{length(\text{List})} > 0 \)
- \( \text{Min} = \text{term()} \)

Returns the minimum element of \( \text{List} \).

\text{nth}(\text{N}, \text{List}) \rightarrow \text{Elem}

Types:
- \( \text{N} = 1..\text{length(\text{List})} \)
- \( \text{List} = \text{[term()]} \)
- \( \text{Elem} = \text{term()} \)

Returns the \( \text{N} \)th element of \( \text{List} \). For example:

\$ \text{lists:nth(3, [a, b, c, d, e])}. \$

\$ \text{c} \$

\text{nthtail}(\text{N}, \text{List1}) \rightarrow \text{Tail}

Types:
- \( \text{N} = 0..\text{length(\text{List1})} \)
- \( \text{List1} = \text{Tail} = \text{[term()]} \)

Returns the \( \text{N} \)th tail of \( \text{List} \), that is, the sublist of \( \text{List} \) starting at \( \text{N}+1 \) and continuing up to the end of the list. For example:

\$ \text{lists:nthtail(3, [a, b, c, d, e])}. \$

\$ \text{[d,e]} \$

\$ \text{tl(tl(tl([a, b, c, d, e])))}. \$

\$ \text{[d,e]} \$

\$ \text{lists:nthtail(0, [a, b, c, d, e])}. \$

\$ \text{[a,b,c,d,e]} \$

\$ \text{lists:nthtail(5, [a, b, c, d, e])}. \$

\$ \text{[]} \$

\text{prefix}(\text{List1}, \text{List2}) \rightarrow \text{bool()}

\$ \text{lists:prefix(\text{[a, b, c, d, e]}, \text{[f, g, h]}).} \$

\$ \text{false} \$

\$ \text{lists:prefix(\text{[f, g, h]}, \text{[a, b, c, d, e]}).} \$

\$ \text{false} \$

\$ \text{lists:prefix(\text{[a, b, c, d, e]}, \text{[a, b, c, d, e]}).} \$

\$ \text{true} \$
reverse(List1) -> List2

Types:
  - List1 = List2 = [term()]

Returns true if List1 is a prefix of List2, otherwise false.

reverse(List1, Tail) -> List2

Types:
  - List1 = Tail = List2 = [term()]

Returns a list with the top level elements in List1 in reverse order, with the tail Tail appended. For example:

> lists:reverse([1, 2, 3, 4], [a, b, c]).
[4,3,2,1,a,b,c]

seq(From, To) -> Seq

seq(From, To, Incr) -> Seq

Types:
  - From = To = Incr = int()
  - Seq = [int()]

Returns a sequence of integers which starts with From and contains the successive results of adding Incr to the previous element, until To has been reached or passed (in the latter case, To is not an element of the sequence). Incr defaults to 1.

Failure: If To < From and Incr is positive, or if To > From and Incr is negative, or if Incr==0 and From /= To.

Examples:

> lists:seq(1, 10).
[1,2,3,4,5,6,7,8,9,10]
> lists:seq(1, 20, 3).
[1,4,7,10,13,16,19]
> lists:seq(1, 1, 0).
[1]

sort(List1) -> List2

Types:
  - List1 = List2 = [term()]

Returns a list containing the sorted elements of List1.

sort(Fun, List1) -> List2

Types:
- Fun = fun(Elem1, Elem2) -> bool()
- Elem1 = Elem2 = term()
- List1 = List2 = [term()]

Returns a list containing the sorted elements of List1, according to the ordering function Fun. Fun(A, B) should return true if A comes before B in the ordering, false otherwise.

\[ \text{split}(N, \text{List1}) \rightarrow \{\text{List2, List3}\} \]

Types:
- \( N = 0..\text{length(List1)} \)
- \( \text{List1} = \text{List2} = \text{List3} = [\text{term()}] \)

Splits List1 into List2 and List3. List2 contains the first \( N \) elements and List3 the rest of the elements (the \( N \)th tail).

\[ \text{sublist}(\text{List1}, \text{Len}) \rightarrow \text{List2} \]

Types:
- \( \text{List1} = \text{List2} = [\text{term()}] \)
- \( \text{Len} = \text{int()} \)

Returns the sub-list of List1 starting at position 1 and with (max) Len elements. It is not an error for Len to exceed the length of the list – in that case the whole list is returned.

\[ \text{sublist}(\text{List1}, \text{Start}, \text{Len}) \rightarrow \text{List2} \]

Types:
- \( \text{List1} = \text{List2} = [\text{term()}] \)
- \( \text{Start} = 1..(\text{length(List1)}+1) \)
- \( \text{Len} = \text{int()} \)

Returns the sub-list of List1 starting at Start and with (max) Len elements. It is not an error for Start+Len to exceed the length of the list.

\[
> \text{lists:sublist}([[1,2,3,4], 2, 2]).
[2,3]
> \text{lists:sublist}([[1,2,3,4], 2, 5]).
[2,3,4]
> \text{lists:sublist}([[1,2,3,4], 5, 2]).
[]
\]

\[ \text{subtract}(\text{List1}, \text{List2}) \rightarrow \text{List3} \]

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:
lists:subtract("123212", "212").
"312".

lists:subtract(A,B) is equivalent to A -- B.

suffix(List1, List2) -> bool()
Returns true if List1 is a suffix of List2, otherwise false.

sum(List) -> number()
Types:
• List = [number()]
Returns the sum of the elements in List.

ukeymerge(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 while removing consecutive duplicate keys. The merge is performed on the Nth element of each tuple. Both TupleList1 and TupleList2 must be key-sorted without duplicates prior to evaluating this function. When elements in the input lists compare equal, elements from TupleList1 are picked.

ukeysort(N, TupleList1) -> TupleList2
Types:
• N = 1..size(Tuple)
• TupleList1 = TupleList2 = [Tuple]
• Tuple = tuple()
Returns a list containing the sorted elements of TupleList1 where all but the first element of the elements comparing equal have been removed. Sorting is performed on the Nth element of the tuples.

umerge(ListOfLists) -> List1
Types:
• ListOfLists = [List]
• List = List1 = [term()]
Returns the sorted list formed by merging all the sub-lists of ListOfLists while removing duplicates. All sub-lists must be sorted and contain no duplicates prior to evaluating this function.

umerge(List1, List2) -> List3
Types:
• List1 = List2 = List3 = [term()]

> lists:subtract("123212", "212").
"312".
Returns the sorted list formed by merging List1 and List2 while removing duplicates. Both List1 and List2 must be sorted and contain no duplicates prior to evaluating this function.

\[ \text{unmerge}(\text{Fun}, \text{List1}, \text{List2}) \rightarrow \text{List3} \]

Types:
- \( \text{Fun} = \text{fun}(A, B) \rightarrow \text{bool}() \)
- \( \text{List1} = [A] \)
- \( \text{List2} = [B] \)
- \( \text{List3} = [A \mid B] \)
- \( A = B = \text{term}() \)

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.

\[ \text{unmerge3}(\text{List1}, \text{List2}, \text{List3}) \rightarrow \text{List4} \]

Types:
- \( \text{List1} = \text{List2} = \text{List3} = \text{List4} = \text{term}() \)

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.

\[ \text{unzip}(\text{List1}) \rightarrow \{ \text{List2}, \text{List3} \} \]

Types:
- \( \text{List1} = \{X, Y\} \)
- \( \text{List2} = [X] \)
- \( \text{List3} = [Y] \)
- \( X = Y = \text{term}() \)

"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.

\[ \text{unzip3}(\text{List1}) \rightarrow \{ \text{List2}, \text{List3}, \text{List4} \} \]

Types:
- \( \text{List1} = \{X, Y, Z\} \)
- \( \text{List2} = [X] \)
- \( \text{List3} = [Y] \)
- \( \text{List4} = [Z] \)
- \( X = Y = Z = \text{term}() \)

"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.

\[ \text{usort}(\text{List1}) \rightarrow \text{List2} \]

Types:
list = List2 = term()

Returns a list containing the sorted elements of List1 without duplicates.

usort(Fun, List1) -> List2

Types:
- Fun = fun(Elem1, Elem2) -> bool()
- Elem1 = Elem2 = term()
- List1 = List2 = term()

Returns a list which contains the sorted elements of List1 where all but the first element of the elements comparing equal according to the ordering function Fun have been removed. 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 $\text{Combine}(X, Y)$.

\[
\text{zipwith} (\text{fun}(X, Y) \rightarrow \{X,Y\} \text{ end}, \text{List1, List2}) \text{ is equivalent to } \text{zip} (\text{List1, List2}).
\]

Examples:

\[
> \text{lists:zipwith} (\text{fun}(X, Y) \rightarrow X+Y \text{ end}, [1,2,3], [4,5,6]).
[5,7,9]
\]

\[
\text{zipwith3}(\text{Combine}, \text{List1, List2, List3}) \rightarrow \text{List4}
\]

Types:

- $\text{Combine} = \text{fun}(X, Y, Z) \rightarrow T$
- $\text{List1} = [X]$
- $\text{List2} = [Y]$
- $\text{List3} = [Z]$
- $\text{List4} = [T]$
- $X = Y = Z = T = \text{term()}$

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 $\text{Combine}(X, Y, Z)$.

\[
\text{zipwith3} (\text{fun}(X, Y, Z) \rightarrow \{X,Y,Z\} \text{ end}, \text{List1, List2, List3}) \text{ is equivalent to } \text{zip3} (\text{List1, List2, List3}).
\]

Examples:

\[
> \text{lists:zipwith3} (\text{fun}(X, Y, Z) \rightarrow X+Y+Z \text{ end}, [1,2,3], [4,5,6], [7,8,9]).
[12,15,18]
\]

\[
> \text{lists:zipwith3} (\text{fun}(X, Y, Z) \rightarrow X,Y,Z \text{ end}, [a,b,c], [x,y,z], [1,2,3]).
[[a,x,1],[b,y,2],[c,z,3]]
\]

\[
\text{all} (\text{Pred, List}) \rightarrow \text{bool()}
\]

Types:

- $\text{Pred} = \text{fun}(\text{Elem}) \rightarrow \text{bool()}$
- $\text{Elem} = \text{term()}$
- $\text{List} = [\text{term()}]$

Returns true if $\text{Pred}(\text{Elem})$ returns true for all elements $\text{Elem}$ in $\text{List}$, otherwise false.

\[
\text{any} (\text{Pred, List}) \rightarrow \text{bool()}
\]

Types:

- $\text{Pred} = \text{fun}(\text{Elem}) \rightarrow \text{bool()}$
- $\text{Elem} = \text{term()}$
- $\text{List} = [\text{term()}]$

Returns true if $\text{Pred}(\text{Elem})$ returns true for at least one element $\text{Elem}$ in $\text{List}$. 
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.

flatmap(Fun, List1) -> List2

Types:
- Fun = fun(A) -> [B]
- List1 = [A]
- List2 = [B]
- A = B = term()

Takes a function from As to lists of Bs, and a list of As (List1) and produces a list of Bs by applying the function to every element in List1 and appending the resulting lists. That is, flatmap behaves as if it had been defined as follows:

\[
\text{flatmap}(\text{Fun}, \text{List1}) \rightarrow \text{append}(\text{map}(\text{Fun}, \text{List1}))
\]

Example:

> lists:flatmap(fun(X) -> [X, X] end, [a, b, c]).
[a, a, b, b, c, c]

foldl(Fun, Acc0, List) -> Acc1

Types:
- Fun = fun(Elem, AccIn) -> AccOut
- Elem = term()
- Acc0 = Acc1 = AccIn = AccOut = term()
- List = [term()]

Calls Fun(Elem, AccIn) on successive elements A of List, starting with AccIn = Acc0. Fun/2 must return a new accumulator which is passed to the next call. The function returns the final value of the accumulator. Acc0 is returned if the list is empty. For example:
lists

> lists:foldl(fun(X, Sum) -> X + Sum end, 0, [1,2,3,4,5]).
15
> lists:foldl(fun(X, Prod) -> X * Prod end, 1, [1,2,3,4,5]).
120

foldr(Fun, Acc0, List) -> Acc1

Types:
- Fun = fun(Elem, AccIn) -> AccOut
- Elem = term()
- Acc0 = Acc1 = AccIn = AccOut = term()
- List = [term()]

Like foldl/3, but the list is traversed from right to left. For example:

> P = fun(A, AccIn) -> io:format("p ", [A]), AccIn end.
#Fun<erl.eval.12.2225172>
> lists:foldl(P, void, [1,2,3]).
1 2 3 void
> lists:foldr(P, void, [1,2,3]).
3 2 1 void

foldl/3 is tail recursive and would usually be preferred to foldr/3.

foreach(Fun, List) -> void()

Types:
- Fun = fun(Elem) -> void()
- Elem = term()
- List = [term()]

Calls Fun(Elem) for each element Elem in List. This function is used for its side effects and the evaluation order is defined to be the same as the order of the elements in the list.

keymap(Fun, N, TupleList1) -> TupleList2

Types:
- Fun = fun(Term1) -> Term2
- Term1 = Term2 = term()
- N = 1..size(Tuple)
- TupleList1 = TupleList2 = [tuple()]

Returns a list of tuples where, for each tuple in TupleList1, the Nth element Term1 of the tuple has been replaced with the result of calling Fun(Term1).

Examples:

> Fun = fun(Atom) -> atom_to_list(Atom) end.
#Fun<erl.eval.6.10732646>
2> lists:keymap(Fun, 2, [[name,jane,22],[name,lizzie,20],[name,lydia,15]]).
[ [name,"jane",22],[name,"lizzie",20],[name,"lydia",15]]
map(Fun, List1) -> List2
Types:
- Fun = fun(A) -> B
- List1 = [A]
- List2 = [B]
- A = B = term()

Takes a function from As to Bs, and a list of As and produces a list of Bs by applying the function to every element in the list. This function is used to obtain the return values. The evaluation order is implementation dependent.

mapfoldl(Fun, Acc0, List1) -> {List2, Acc1}
Types:
- Fun = fun(A, AccIn) -> {B, AccOut}
- Acc0 = Acc1 = AccIn = AccOut = term()
- List1 = [A]
- List2 = [B]
- A = B = term()

mapfold combines the operations of map/2 and foldl/3 into one pass. An example, summing the elements in a list and double them at the same time:

```eclipse
data_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,c,d,e],[1,2,3,4]}

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:

splitwith(Pred, List) ->
  {takewhile(Pred, List), dropwhile(Pred, List)}.

Examples:
> lists:splitwith(fun(A) -> A rem 2 == 1 end, [1,2,3,4,5,6,7]).
{[1],[2,3,4,5,6,7]}
> lists:splitwith(fun(A) -> is_atom(A) end, [a,b,1,c,d,2,3,4,e]).
{[a,b],[1,c,d,2,3,4,e]}

See also partition/2 for a different way to partition a list.

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)`

`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 file. `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 173], `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

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) \rightarrow \text{float()}

Types:
- \( X = \text{number()} \)

Returns the error function of \( X \), where

\[
\text{erf}(X) = \frac{2}{\sqrt{\pi}} \int_{0}^{X} e^{-t^2} \, dt.
\]

erfc(X) \rightarrow \text{float()}

Types:
- \( X = \text{number()} \)

\text{erfc}(X) \text{ returns } 1.0 - \text{erf}(X), \text{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 `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 to 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, [{'#emp{empno = '$1', dept = sales, _='_'}, [], ['$1']}]).
```

Even though the record syntax is used, it's still somewhat hard to read and even harder to write. The first element of the tuple, `#emp{empno = '$1', dept = sales, _='_'}` tells what to match, elements not matching this will not be returned at all, as in the `ets:match` example. The second element, the empty list is a list of guard expressions, which we need none, and the third element is the list of expressions constructing the return value (in ets this almost always is a list containing one single term). In our case `$1` is bound to the employee number in the head (first element of tuple), and hence it is the employee number that is returned. The result is `"011103", "076324"]`, just as in the `ets:foldr` example, but the result is retrieved much more efficiently in terms of execution speed and memory consumption.

We have one efficient but hardly readable way of doing it and one inefficient but fairly readable (at least to the skilled Erlang programmer) way of doing it. With the use of
ets:fun2ms, one could have something that is as efficient as possible but still is written as a filter using the fun syntax:

-include_lib("stdlib/include/ms_transform.hrl").

% ... 

ets:select(emp_tab, ets:fun2ms(
    fun(#emp{empno = E, dept = sales}) ->
        E
    end)).

This may not be the shortest of the expressions, but it requires no special knowledge of match specifications to read. The fun’s head should simply match what you want to filter out and the body returns what you want returned. As long as the fun can be kept within the limits of the match specifications, there is no need to transfer all data of the table to the process for filtering as in the ets:foldr example. In fact it’s even easier to read then the ets:foldr example, as the select call in itself discards anything that doesn’t match, while the fun of the foldr call needs to handle both the elements matching and the ones not matching.

It’s worth noting in the above ets:fun2ms example that one needs to include ms_transform.hrl in the source code, as this is what triggers the parse transformation of the ets:fun2ms call to a valid match specification. This also implies that the transformation is done at compile time (except when called from the shell of course) and therefore will take no resources at all in runtime. So although you use the more intuitive fun syntax, it gets as efficient in runtime as writing match specifications by hand.

Let’s look at some more ets examples. Let’s say one wants to get all the employee numbers of any employee hired before the year 2000. Using ets:match isn’t an alternative here as relational operators cannot be expressed there. Once again, an ets:foldr could do it (slowly, but correct):

ets:foldr(fun(#emp{empno = E, empyear = Y},Acc) when Y < 2000 -> [E | Acc];
    (_,Acc) -> Acc
end,
    [],
    emp_tab).

The result will be ["052341", "076324", "535216", "789789", "989891"], as expected. Now the equivalent expression using a handwritten match specification would look something like this:

ets:select(emp_tab,[#{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 foldr example. Let’s jump on to writing it using ets:fun2ms.
-include_lib("stdlib/include/ms_transform.hrl").

% ...  

ets:select(emp_tab, ets:fun2ms(
    fun(#emp{empno = E, empyear = Y}) when Y < 2000 ->
      E
    end)).

Obviously readability is gained by using the parse transformation.
I’ll show some more examples without the tiresome comparing-to-alternatives stuff.
Let’s say we’d want the whole object matching instead of only one element. We could of course assign a variable to every part of the record and build it up once again in the body of the fun, but it’s easier to do like this:

ets:select(emp_tab, ets:fun2ms(
    fun(Obj = #emp{empno = E, empyear = Y})
    when Y < 2000 ->
      Obj
    end)).

Just as in ordinary Erlang matching, you can bind a variable to the whole matched object using a “match in then match”, i.e. a =. Unfortunately this is not general in fun’s translated to match specifications, only on the “top level”, i.e. matching the whole object arriving to be matched into a separate variable, is it allowed. For the one’s used to writing match specifications by hand, I’ll have to mention that the variable A will simply be translated into ‘$’. It’s not general, but it has very common usage, why it is handled as a special, but useful, case. If this bothers you, the pseudo function object also returns the whole matched object, see the part about caveats and limitations below.

Let’s do something in the fun’s body too: Let’s say that someone realizes that there are a few people having an employee number beginning with a zero (0), which shouldn’t be allowed. All those should have their numbers changed to begin with a one (1) instead and one wants the list [[<Old empno>,<New empno>]] created:

ets:select(emp_tab, ets:fun2ms(
    fun(#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>}:

ets:select(emp_tab, ets:fun2ms(
    fun(#emp{empno = E, surname = "Smith" }) ->
      {guru,E};
      (#emp{empno = E, empyear = Y}) when Y < 1997 ->
        {inventory, E};
    end)).

The result will be:

```
[{rookie,"011103"},
 {rookie,"041231"},
 {guru,"052341"},
 {guru,"076324"},
 {newbie,"122334"},
 {rookie,"535216"},
 {inventory,"789789"},
 {newbie,"963721"},
 {rookie,"989891"}]
```

and so the Smith's will be happy...

So, what more can you do? Well, the simple answer would be; look in the documentation of match specifications in ERTS users guide. However let's briefly go through the most useful "built in functions" that you can use when the fun is to be translated into a match specification by ets:fun2ms (it's worth mentioning, although it might be obvious to some, that calling other functions than the one's allowed in match specifications cannot be done. No "usual" Erlang code can be executed by the fun being translated by fun2ms, the fun is after all limited exactly to the power of the match specifications, which is unfortunate, but the price one has to pay for the execution speed of an ets:select compared to ets:foldl/foldr).

The head of the fun is obviously a head matching (or mismatching) one parameter, one object of the table we select from. The object is always a single variable (can be \texttt{_{}}) 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 \texttt{=} 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 (\texttt{is_integer(X)} instead of \texttt{is\_integer(X)}). As the new type tests (the \texttt{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: \texttt{is\_atom}, \texttt{is\_constant}, \texttt{is\_float}, \texttt{is\_integer}, \texttt{is\_list}, \texttt{is\_number}, \texttt{is\_pid}, \texttt{is\_port}, \texttt{is\_reference}, \texttt{is\_tuple}, \texttt{is\_binary}, \texttt{is\_function}, \texttt{is\_record}
- The boolean operators: \texttt{not}, \texttt{and}, \texttt{or}, \texttt{andalso}, \texttt{orelse}
- The relational operators: \texttt{\_}, \texttt{=}, \texttt{>, =<, =:=, /=, /=}
- Arithmetics: \texttt{+}, \texttt{-}, \texttt{*}, \texttt{div}, \texttt{rem}
- Bitwise operators: \texttt{band}, \texttt{bor}, \texttt{bxor}, \texttt{bnot}, \texttt{bsl}, \texttt{bsr}
- The guard bif's: \texttt{abs}, \texttt{element}, \texttt{hd}, \texttt{length}, \texttt{node}, \texttt{round}, \texttt{size}, \texttt{tl}, \texttt{trunc}, \texttt{self}
- The obsolete type test (only in guards): \texttt{atom}, \texttt{constant}, \texttt{float}, \texttt{integer}, \texttt{list}, \texttt{number}, \texttt{pid}, \texttt{port}, \texttt{reference}, \texttt{tuple}, \texttt{binary}, \texttt{function}, \texttt{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
\[
\text{ets:new(toy_table,<something>):}
\]
\[
3> \text{dbg:tp(ets.new,dbg:fun2ms(fun([toy_table,\_] \to \text{true} end)).}
\{
\text{ok,[[matched,nonode@nohost,1],[saved,1]]}}
\]
As can be seen, the fun’s used with \text{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 \text{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:
\[
(<0.86.0>) \text{call ets:new(toy_table,[ordered_set])}
\]
Let’s play we haven’t spotted the problem yet, and want to see what \text{ets:new} returns. We do a slightly different trace pattern:
\[
4> \text{dbg:tp(ets.new,dbg:fun2ms(fun([toy_table,\_] \to \text{return_trace()} end)).}
\]
Resulting in the following trace output when we run the test:
\[
(<0.86.0>) \text{call ets:new(toy_table,[ordered_set])}
\]
\[
(<0.86.0>) \text{returned from ets:new/2 \to 24}
\]
The call to \text{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 \text{dbg} match specification.

As the test now fails with \{badmatch,24\}, it’s obvious that the badmatch is because the atom \text{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:
\[
\text{start(Args) \to}
\]
\[
\text{toy_table = ets:new(toy_table,[named_table |Args]).}
\]
And with the same tracing turned on, we get the following trace output:
\[
(<0.86.0>) \text{call ets:new(toy_table,[named_table,ordered_set])}
\]
\[
(<0.86.0>) \text{returned from ets:new/2 \to \text{toy_table}}
\]
Very well. Let’s say the module now passes all testing and goes into the system. After a while someone realizes that the table \text{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:
\[
1> \text{dbg:tracer().}
\{
\text{ok,<<0.88.0>>}}
\]
\[
2> \text{dbg:p(all,call).}
\{
\text{ok,[[matched,nonode@nohost,25]]}}
\]
\[
3> \text{dbg:tpl(toy,store,dbg:fun2ms(fun([A,\_] when is_atom(A) \to \text{true} end)).}
\{
\text{ok,[[matched,nonode@nohost,1],[saved,1]]}}
\]
We use `dbg:tpl` here to make sure to catch local calls (let’s say the module has grown since the smaller version and we’re not sure this inserting of atoms is not done locally...). When in doubt always use local call tracing.

Let’s say nothing happens when we trace in this way. Our function is never called with these parameters. We make the conclusion that someone else (some other module) is doing it and we realize that we must trace on `ets:insert` and want to see the calling function. The calling function may be retrieved using the match specification function `caller` and to get it into the trace message, one has to use the match spec function `message`. The filter call looks like this (looking for calls to `ets:insert`):

```erl
4> dbg:tpl(ets,insert,dbg:fun2ms(fun([toy_table,{A,_}] when is_atom(A) -> message(caller()) end)).
{ok,[{matched,nonode@nohost,1},{saved,2}])
```

The caller will now appear in the “additional message” part of the trace output, and so after a while, the following output comes:

```erl
(<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 is_list(X) will be translated to
  {is_list, '$1'} ('$1' is just an example, the numbering may vary). If one
tries to call a function that is not a match_spec builtin, it will cause an error.

• Variables occurring in the head of the fun will be replaced by match_spec variables
  in the order of occurrence, so that the fragment fun({A,B,C}) will be replaced by
  {='$1', '$2', '$3'} etc. Every occurrence of such a variable later in the
  match_spec will be replaced by a match_spec variable in the same way, so that the
  fun fun({A,B}) when is_atom(A) -> B end will be translated into
  [{='$1', '$2'}, [{is_atom, '$1'}], ['$2']].

• Variables that are not appearing in the head are imported from the environment
  and made into match_spec const expressions. Example from the shell:

    1> X = 25.
    X = 25
    2> ets:fun2ms(fun({A,B}) when A > X -> B end).
    [{='$1', '$2'}, [{>', '$1', {const, 25}}], ['$2']]}

• Matching with = cannot be used in the body. It can only be used on the top level in
  the head of the fun. Example from the shell again:

    1> ets:fun2ms(fun({A,[B|C]} = D) when A > B -> D end).
    [{='$1', ['$2', '$3']}, [{>', '$1', '$2'}], ['$_']].
    2> ets:fun2ms(fun({A,[B|C]=D}) when A > B -> D end).
    Error: fun with head matching (= in head) cannot be translated into match_spec
    {error, transform_error}
    3> ets:fun2ms(fun({A,[B|C]}) when A > B -> D = [B|C], D end).
    Error: fun with body matching (= in body) is illegal as match_spec
    {error, transform_error}

All variables are bound in the head of a match_spec, so the translator can not allow
multiple bindings. The special case when matching is done on the top level makes
the variable bind to '$_' in the resulting match_spec, it is to allow a more natural
access to the whole matched object. The special function object() could be used
instead, see below. The following expressions are translated equally:

    ets:fun2ms(fun({A,_,} = A) -> A end).
    ets:fun2ms(fun({A,_,}) -> object() end).

• The special match_spec variables '$_' and '$*' can be accessed through the
  pseudo functions object() (for '$_') and bindings() (for '$*'). as an example,
  one could translate the following ets:match_object/2 call to a ets:select call:

    ets:match_object(Table, {'$1',test,'$2'}).

    ...is the same as...

    ets:select(Table, ets:fun2ms(fun({A,test,B}) -> object() end)).

(This was just an example, in this simple case the former expression is probably
preferable in terms of readability). The ets:select/2 call will conceptually look
like this in the resulting code:

    ets:select(Table, [{'$1',test,'$2'}, [], ['$_']]).

Matching on the top level of the fun head might feel like a more natural way to
access '$_', see above.
Term constructions/literals are translated as much as is needed to get them into valid match specs, so that tuples are made into match spec tuple constructions (a one element tuple containing the tuple) and constant expressions are used when importing variables from the environment. Records are also translated into plain tuple constructions, calls to element etc. The guard test is_record/2 is translated into match spec code using the three parameter version that's built into match specs so that is_record(A,t) is translated into {is_record,'$1',t,5} given that the record size of record type t is 5.

Language constructions like case, if, catch etc that are not present in match specs are not allowed.

If the header file ms_transform.hrl is not included, the fun won't be translated, which may result in a runtime error (depending on if the fun is valid in a pure Erlang context). Be absolutely sure that the header is included when using ets and dbg:fun2ms/1 in compiled code.

If the pseudo function triggering the translation is ets:fun2ms/1, the fun's head must contain a single variable or a single tuple. If the pseudo function is dbg:fun2ms/1 the fun's head must contain a single variable or a single list.

The translation from fun's to match specs is done at compile time, so runtime performance is not affected by using these pseudo functions. The compile time might be somewhat longer though.

For more information about match specs, please read about them in ERTS users guide.

Exports

parse_transform(Forms, Options) -> Forms

Types:
- Forms = Erlang abstract code format, see the erl_parse module description
- Options = Option list, required but not used

Implements the actual transformation at compile time. This function is called by the compiler to do the source code transformation if and when the ms_transform.hrl header file is included in your source code. See the ets and dbg:fun2ms/1 function manual pages for documentation on how to use this parse_transform, see the match spec chapter in ERTS users guide for a description of match specifications.

transform_from_shell(Dialect, Clauses, BoundEnvironment) -> term()

Types:
- Dialect = ets | dbg
- Clauses = Erlang abstract form for a single fun
- BoundEnvironment = [{atom(), term()}, ...], list of variable bindings in the shell environment

Implements the actual transformation when the fun2ms functions are called from the shell. In this case the abstract form is for one single fun (parsed by the Erlang shell), and all imported variables should be in the key-value list passed as BoundEnvironment. The result is a term, normalized, i.e. not in abstract format.
format_error(Errcode) -> ErrorMessage

Types:
- Errcode = term()
- ErrorMessage = string()

Takes an error code returned by one of the other functions in the module and creates a textual description of the error. Fairly uninteresting function actually.
orddict

Erlang Module

Orddict implements a Key - Value dictionary. An orddict is a representation of a dictionary, where a list of pairs is used to store the keys and values. The list is ordered after the keys.

This module provides exactly the same interface as the module dict but with a defined representation.

See Also

dict(3) [page 87], gb_trees(3) [page 168]
ordsets

Erlang Module

Sets are collections of elements with no duplicate elements. An ordset is a representation of a set, where an ordered list is used to store the elements of the set. An ordered list is more efficient than an unordered list. This module provides exactly the same interface as the module sets but with a defined representation.

See Also

gb_sets(3) [page 162], sets(3) [page 293]
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

```erlang
create(PgName) -> ok | {error, Reason}
Types:
  • PgName = term()
  • Reason = already_created | term()
Creates an empty group named PgName on the current node.

create(PgName, Node) -> ok | {error, Reason}
Types:
  • PgName = term()
  • Node = node()
  • Reason = already_created | term()
Creates an empty group named PgName on the node Node.

join(PgName, Pid) -> Members
Types:
  • PgName = term()
  • Pid = pid()
  • Members = [pid()]
```
Joins the pid Pid to the process group PgName. Returns a list of all old members of the group.

```
send(PgName, Msg) -> void()

Types:
  • PgName = Msg = term()
Sends the tuple {pg_message, From, PgName, Msg} to all members of the process group PgName.
Failure: {badarg, {PgName, Msg}} if PgName is not a process group (a globally registered name).
```

```
esend(PgName, Msg) -> void()

Types:
  • PgName = Msg = term()
Sends the tuple {pg_message, From, PgName, Msg} to all members of the process group PgName, except ourselves.
Failure: {badarg, {PgName, Msg}} if PgName is not a process group (a globally registered name).
```

```
members(PgName) -> Members

Types:
  • PgName = term()
  • Members = [pid()]
Returns a list of all members of the process group PgName.
```
pool

Erlang Module

pool can be used to run a set of Erlang nodes as a pool of computational processors. It is organized as a master and a set of slave nodes and includes the following features:

- The slave nodes send regular reports to the master about their current load.
- Queries can be sent to the master to determine which node will have the least load.

The BIF `statistics(run_queue)` is used for estimating future loads. It returns the length of the queue of ready to run processes in the Erlang runtime system.

The slave nodes are started with the `slave` module. This effects, tty IO, file IO, and code loading.

If the master node fails, the entire pool will exit.

Exports

```erlang
start(Name) ->
start(Name, Args) -> Nodes
```

Types:
- Name = atom()
- Args = string()
- Nodes = [node()]

Starts a new pool. The file `.hosts.erlang` is read to find host names where the pool nodes can be started. See section Files [page 254] below. The start-up procedure fails if the file is not found.

The slave nodes are started with `slave:start/2,3`, passing along `Name` and, if provided, `Args`. `Name` is used as the first part of the node names, `Args` is used to specify command line arguments. See slave(3) [page 306].

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.

```erlang
attach(Node) -> alreadyAttached | attached
```

Types:
- Node = node()
This function ensures that a pool master is running and includes Node in the pool master's pool of nodes.

\[ \text{stop()} \rightarrow \text{stopped} \]

Stop the pool and kills all the slave nodes.

\[ \text{get_nodes()} \rightarrow \text{Nodes} \]

Types:
- **Nodes** = [node()]

Returns a list of the current member nodes of the pool.

\[ \text{pspawn(Mod, Fun, Args)} \rightarrow \text{pid()} \]

Types:
- **Mod** = Fun = atom()
- **Args** = [term()]

Spawns a process on the pool node which is expected to have the lowest future load.

\[ \text{pspawn_link(Mod, Fun, Args)} \rightarrow \text{pid()} \]

Types:
- **Mod** = Fun = atom()
- **Args** = [term()]

Spawn links a process on the pool node which is expected to have the lowest future load.

\[ \text{get_node()} \rightarrow \text{node()} \]

Returns the node with the expected lowest future load.

**Files**

*.hosts.erlang* is used to pick hosts where nodes can be started. See [*net_adm(3)*] for information about format and location of this file.

*$\text{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.
This module is used to start processes adhering to the [OTP Design Principles]. Specifically, the functions in this module are used by the OTP standard behaviors (gen_server, gen_fsm, ...) when starting new processes. The functions can also be used to start special processes, user defined processes which comply to the OTP design principles. See [Sys and Proc_Lib] in OTP Design Principles for an example.

Some useful information is initialized when a process starts. The registered names, or the process identifiers, of the parent process, and the parent ancestors, are stored together with information about the function initially called in the process.

While in “plain Erlang” a process is said to terminate normally only for the exit reason normal, a process started using proc_lib is also said to terminate normally if it exits with reason shutdown. This is the reason used when an application (supervision tree) is stopped.

When a process started using proc_lib terminates abnormally - that is, with another exit reason than normal or shutdown - a crash report is generated, which is written to terminal by the default SASL event handler. That is, the crash report is normally only visible if the SASL application is started. See [sasl(6)] and [SASL User's Guide].

The crash report contains the previously stored information such as ancestors and initial function, the termination reason, and information regarding other processes which terminate as a result of this process terminating.

Exports

spawn(Fun) -> pid()
spawn(Node, Fun) -> pid()
spawn(Module, Function, Args) -> pid()
spawn(Node, Module, Function, Args) -> pid()

Types:
- Node = node()
- Fun = fun() -> void()
- Module = Function = atom()
- Args = [term()]

Spawns a new process and initializes it as described above. The process is spawned using the [spawn] BIFs.

spawn_link(Fun) -> pid()
spawn_link(Node, Fun) -> pid()
spawn_link(Module, Function, Args) -> pid()
spawn_link(Node, Module, Function, Args) -> pid()

Types:
- Node = node()
- Fun = fun() -> void()
- Module = Function = atom()
- Args = [term()]

Spawns a new process and initializes it as described above. The process is spawned using the [spawn_link] BIFs.

spawn_opt(Fun, SpawnOpts) -> pid()
spawn_opt(Node, Fun, SpawnOpts) -> pid()
spawn_opt(Module, Function, Args, SpawnOpts) -> pid()
spawn_opt(Node, Module, Func, Args, SpawnOpts) -> pid()

Types:
- Node = node()
- Fun = fun() -> void()
- Module = Function = atom()
- Args = [term()]
- SpawnOpts – see erlang:spawn_opt/2,3,4,5

Spawns a new process and initializes it as described above. The process is spawned using the [spawn_opt] BIFs.

**Note:**
Using the spawn option monitor is currently not allowed, but will cause the function to fail with reason badarg.

start(Module, Function, Args) -> Ret
start(Module, Function, Args, Time) -> Ret
start(Module, Function, Args, Time, SpawnOpts) -> Ret
start_link(Module, Function, Args) -> Ret
start_link(Module, Function, Args, Time) -> Ret
start_link(Module, Function, Args, Time, SpawnOpts) -> Ret

Types:
- Module = Function = atom()
- Args = [term()]
- Time = int() >= 0 | infinity
- SpawnOpts – see erlang:spawn_opt/2,3,4,5
- Ret = term() | {error, Reason}
Starts a new process synchronously. Spawns the process and waits for it to start. When the process has started, it must call init_ack(Parent, Ret) [page 257] or init_ack(Ret) [page 257], where Parent is the process that evaluates this function. At this time, Ret is returned.

If the start_link/3,4,5 function is used and the process crashes before it has called init_ack/1,2, {error, Reason} is returned if the calling process traps exits.

If Time is specified as an integer, this function waits for Time milliseconds for the new process to call init_ack, or {error, timeout} is returned, and the process is killed.

The SpawnOpts argument, if given, will be passed as the last argument to the spawn_opt/2,3,4,5 BIF.

**Note:**
Using the spawn option monitor is currently not allowed, but will cause the function to fail with reason badarg.

`init_ack(Parent, Ret) -> void()`
`init_ack(Ret) -> void()`

**Types:**
- Parent = pid()
- Ret = term()

This function must be used by a process that has been started by a start_link/3,4,5 [page 256] function. It tells Parent that the process has initialized itself, has started, or has failed to initialize itself.

The init_ack/1 function uses the parent value previously stored by the start function used.

If this function is not called, the start function will return an error tuple (if a link and/or a timeout is used) or hang otherwise.

The following example illustrates how this function and proc_lib:start_link/3 are used.

```erlang
-module(my_proc).
-export([start_link/0]).
-export([init/1]).

start_link() ->
    proc_lib:start_link(my_proc, init, [self()]).

init(Parent) ->
    case do_initialization() of
        ok ->
            proc_lib:init_ack(Parent, {ok, self()});
        {error, Reason} ->
            exit(Reason)
    end,
    loop().
```

...
format(CrashReport) -> string()

Types:
- CrashReport = term()

This function can be used by a user defined event handler to format a crash report. The crash report is sent using \texttt{error_logger:report} \texttt{(crash\_report, CrashReport)}. That is, the event to be handled is of the format \{error_report, GL, \{Pid, crash\_report, CrashReport\}\} where GL is the group leader pid of the process Pid which sent the crash report.

\texttt{initial\_call(Process) -> \{Module,Function,Args\} | Fun | false}

Types:
- Process = pid() | \{X,Y,Z\} | ProcInfo
- X = Y = Z = int()
- ProcInfo = term()
- Module = Function = atom()
- Args = \[term()\]
- Fun = fun() -> void()

Extracts the initial call of a process that was started using one of the \texttt{spawn} or \texttt{start} functions described above. Process can either be a pid, an integer tuple (from which a pid can be created), or the process information of a process Pid fetched through an \texttt{erlang:process_info(Pid)} function call.

\texttt{translate\_initial\_call(Process) -> \{Module,Function,Arity\} | Fun}

Types:
- Process = pid() | \{X,Y,Z\} | ProcInfo
- X = Y = Z = int()
- ProcInfo = term()
- Module = Function = atom()
- Arity = int()
- Fun = fun() -> void()

This function is used by the \texttt{c:i/0} and \texttt{c:regs/0} functions in order to present process information.

Extracts the initial call of a process that was started using one of the \texttt{spawn} or \texttt{start} functions described above, and translates it to more useful information. Process can either be a pid, an integer tuple (from which a pid can be created), or the process information of a process Pid fetched through an \texttt{erlang:process_info(Pid)} function call.

If the initial call is to one of the system defined behaviors such as \texttt{gen\_server} or \texttt{gen\_event}, it is translated to more useful information. If a \texttt{gen\_server} is spawned, the returned Module is the name of the callback module and Function is \texttt{init} (the function that initiates the new server).

A \texttt{supervisor} and a \texttt{supervisor\_bridge} are also \texttt{gen\_server} processes. In order to return information that this process is a \texttt{supervisor} and the name of the \texttt{callback} module, Module \texttt{is supervisor} and Function is the name of the supervisor \texttt{callback} module. Arity is 1 since the \texttt{init/1} function is called initially in the \texttt{callback} module.
By default, \{proc_lib,init,p,5\} is returned if no information about the initial call can be found. It is assumed that the caller knows that the process has been spawned with the proc_lib module.

hibernate(Module, Function, Args)

Types:
- Module = Function = atom()
- Args = [term()]

This function does the same as (and does call) the BIF \{hibernate/3\}, but ensures that exception handling and logging continues to work as expected when the process wakes up. Always use this function instead of the BIF for processes started using proc_lib functions.

SEE ALSO

[error_logger(3)]
proplists

Erlang Module

Property lists are ordinary lists containing entries in the form of either tuples, whose first elements are keys used for lookup and insertion, or atoms, which work as shorthand for tuples \{Atom, true\}. (Other terms are allowed in the lists, but are ignored by this module.) If there is more than one entry in a list for a certain key, the first occurrence normally overrides any later (irrespective of the arity of the tuples).

Property lists are useful for representing inherited properties, such as options passed to a function where a user may specify options overriding the default settings, object properties, annotations, etc.

Exports

append_values(Key, List) -> List

Types:
• Key = term()
• List = [term()]

Similar to get_all_values/2, but each value is wrapped in a list unless it is already itself a list, and the resulting list of lists is concatenated. This is often useful for “incremental” options; e.g., append_values(a, \[[a, [1,2]], [b, 0], [a, 3], [c, -1], [a, [4]]\]) will return the list \[1,2,3,4\].

compact(List) -> List

Types:
• List = [term()]

Minimizes the representation of all entries in the list. This is equivalent to \[property(P) || P <- List\].

See also: property/1, unfold/1.

delete(Key, List) -> List

Types:
• Key = term()
• List = [term()]

Deletes all entries associated with Key from List.

expand(Expansions, List) -> List

Types:
- Key = term()
- Expansions = [{Property,[term()]}]
- Property = atom() | tuple()

Expands particular properties to corresponding sets of properties (or other terms). For each pair {Property, Expansion} in Expansions, if E is the first entry in List with the same key as Property, and E and Property have equivalent normal forms, then E is replaced with the terms in Expansion, and any following entries with the same key are deleted from List.

For example, the following expressions all return [fie, bar, baz, fum]:

```
expand([{foo, [bar, baz]}],
       [fie, foo, fum])
expand([{{foo, true}, [bar, baz]}],
       [fie, foo, fum])
expand([{{foo, false}, [bar, baz]}],
       [fie, {foo, false}, fum])
```

However, no expansion is done in the following call:

```
expand([{{foo, true}, [bar, baz]}],
       [{foo, false}, fie, foo, fum])
```

because {foo, false} shadows foo.

Note that if the original property term is to be preserved in the result when expanded, it must be included in the expansion list. The inserted terms are not expanded recursively. If Expansions contains more than one property with the same key, only the first occurrence is used.

See also: normalize/2.

```prolog
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.
```

```prolog
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.
```

```prolog
get_keys(List) -> [term()]

Types:
- List = [term()]
```
Returns an unordered list of the keys used in \texttt{List}, not containing duplicates.

\begin{verbatim}
get\_value(Key, List) -\rightarrow term()
\end{verbatim}

Types:
- Key = term()
- List = [term()]

Equivalent to \texttt{get\_value(Key, List, undefined)}.

\begin{verbatim}
get\_value(Key, List, Default) -\rightarrow term()
\end{verbatim}

Types:
- Key = term()
- Default = term()
- List = [term()]

Returns the value of a simple key/value property in \texttt{List}. If \texttt{lookup(Key, List)} would yield \{Key, Value\}, this function returns the corresponding Value, otherwise \texttt{Default} is returned.

See also: \texttt{get\_all\_values/2, get\_bool/2, get\_value/1, lookup/2}.

\begin{verbatim}
is\_defined(Key, List) -\rightarrow bool()
\end{verbatim}

Types:
- Key = term()
- List = [term()]

Returns true if \texttt{List} contains at least one entry associated with \texttt{Key}, otherwise \texttt{false} is returned.

\begin{verbatim}
lookup(Key, List) -\rightarrow none | tuple()
\end{verbatim}

Types:
- Key = term()
- List = [term()]

Returns the first entry associated with \texttt{Key} in \texttt{List}, if one exists, otherwise returns \texttt{none}. For an atom \texttt{A} in the list, the tuple \{\texttt{A}, true\} is the entry associated with \texttt{A}.

See also: \texttt{get\_bool/2, get\_value/2, lookup\_all/2}.

\begin{verbatim}
lookup\_all(Key, List) -\rightarrow [tuple()]
\end{verbatim}

Types:
- Key = term()
- List = [term()]

Returns the list of all entries associated with \texttt{Key} in \texttt{List}. If no such entry exists, the result is the empty list.

See also: \texttt{lookup/2}.

\begin{verbatim}
normalize(List, Stages) -\rightarrow List
\end{verbatim}

Types:
• List = [term()]  
• Stages = [Operation]  
• Operation = [aliases, Aliases] | {negations, Negations} | {expand, Expansions}  
• Aliases = [[Key, Key]]  
• Negations = [[Key, Key]]  
• Key = term()  
• Expansions = [{Property, [term()]})]  
• Property = atom() | tuple()

Passes List through a sequence of substitution/expansion stages. For an aliases operation, the function substitute_aliases/2 is applied using the given list of aliases; for a negations operation, substitute_negations/2 is applied using the given negation list; for an expand operation, the function expand/2 is applied using the given list of expansions. The final result is automatically compacted (cf. compact/1).

Typically you want to substitute negations first, then aliases, then perform one or more expansions (sometimes you want to pre-expand particular entries before doing the main expansion). You might want to substitute negations and/or aliases repeatedly, to allow such forms in the right-hand side of aliases and expansion lists.

See also: compact/1, expand/2, substitute_aliases/2, substitute_negations/2.

property(Property) -> Property

Types:
• Property = atom() | tuple()

Creates a normal form (minimal) representation of a property. If Property is {Key, true} where Key is an atom, this returns Key, otherwise the whole term Property is returned.

See also: property/2.

property(Key, Value) -> Property

Types:
• Key = term()  
• Value = term()  
• Property = atom() | tuple()

Creates a normal form (minimal) representation of a simple key/value property. Returns Key if Value is true and Key is an atom, otherwise a tuple {Key, Value} is returned.

See also: property/1.

split(List, Keys) -> {Lists, Rest}

Types:
• List = [term()]  
• Keys = [term()]  
• Lists = [[term()]])  
• Rest = [term()]

property(Property) -> Property

Types:
• Property = atom() | tuple()

Creates a normal form (minimal) representation of a property. If Property is {Key, true} where Key is an atom, this returns Key, otherwise the whole term Property is returned.

See also: property/2.

property(Key, Value) -> Property

Types:
• Key = term()  
• Value = term()  
• Property = atom() | tuple()

Creates a normal form (minimal) representation of a simple key/value property. Returns Key if Value is true and Key is an atom, otherwise a tuple {Key, Value} is returned.

See also: property/1.

split(List, Keys) -> {Lists, Rest}

Types:
• List = [term()]  
• Keys = [term()]  
• Lists = [[term()]])  
• Rest = [term()]
Partitions List into a list of sublists and a remainder. Lists contains one sublist for each key in Keys, in the corresponding order. The relative order of the elements in each sublist is preserved from the original List. Rest contains the elements in List that are not associated with any of the given keys, also with their original relative order preserved.

Example: split([a, c, 2], [e, 1], [a, b, c]) returns

```
{[a], [{b, 5}, b, {c, 2}, [c, 3, 4]]}, [{e, 1}, d])
```

\[\text{substitute\_aliases}(\text{Aliases}, \text{List}) \rightarrow \text{List}\]

Types:
- \(\text{Aliases} = \{\text{Key, Key}\}\)
- \(\text{Key} = \text{term}\(())\)
- \(\text{List} = \{\text{term}\(())\}\)

Substitutes keys of properties. For each entry in List, if it is associated with some key \(K_1\) such that \(K_1, K_2\) occurs in Aliases, the key of the entry is changed to \(K_2\). If the same \(K_1\) 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.

\[\text{substitute\_negations}(\text{Negations}, \text{List}) \rightarrow \text{List}\]

Types:
- \(\text{Negations} = \{\text{Key, Key}\}\)
- \(\text{Key} = \text{term}\(())\)
- \(\text{List} = \{\text{term}\(())\}\)

Substitutes keys of boolean-valued properties and simultaneously negates their values. For each entry in List, if it is associated with some key \(K_1\) such that \(K_1, \text{true}\) occurs in Negations, then if the entry was \(K_1, \text{true}\) it will be replaced with \(K_2, \text{false}\), otherwise it will be replaced with \(K_2, \text{true}\), thus changing the name of the option and simultaneously negating the value given by \text{get\_bool}(\text{List}). If the same \(K_1\) 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.

\[\text{unfold}(\text{List}) \rightarrow \text{List}\]

Types:
- \(\text{List} = \{\text{term}\(())\}\)

Unfolds all occurrences of atoms in List to tuples {Atom, true}. 

264
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 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 Mnomesyne. 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: Mnomesyne 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 Mnomesyne and to be more versatile by means of QLC tables.

While ordinary list comprehensions evaluate to lists, calling qlc:q/1,2 returns a Query Handle. To obtain all the answers to a query, qlc:eval/1,2 should be called with the query handle as first argument. Query handles are essentially functions created in the module calling q/1,2. As the functions 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 with a query handle as first argument. Query cursors are essentially Erlang processes. One answer at a time is sent from the query cursor process to the process that created the cursor.

Syntax

Syntactically QLCs have the same parts as ordinary list comprehensions:

[Expression || Qualifier1, Qualifier2, ...]
Expression (the template) is an arbitrary Erlang expression. Qualifiers are either filters or generators. Filters are Erlang expressions returning bool(). Generators have the form Pattern<->ListExpression, where ListExpression is an expression evaluating to a query handle or a list. Query handles are returned from qlc:table/2, qlc:append/1,2, qlc:sort/1,2, qlc:keysort/2,3, qlc:q/1,2, and qlc:string_to_handle/1,2,3.

Evaluation

The evaluation of a query handle begins by the inspection of options and the collection of information about tables. As a result qualifiers are modified during the optimization phase. Next all list expressions are evaluated. If a cursor has been created evaluation takes place in the cursor process. For those list expressions that are QLCs, the list expressions of the QLCs’ generators are evaluated as well. One has to be careful if list expressions have side effects since the order in which list expressions are evaluated is unspecified. Finally the answers are found by evaluating the qualifiers from left to right, backtracking when some filter returns false, or collecting the template when all filters return true.

Filters that do not return bool() but fail are handled differently depending on their syntax: if the filter is a guard it returns false, otherwise the query evaluation fails. This behavior makes it possible for QLC to do some optimizations without affecting the meaning of a query. For example, when some position of a table is compared to one or more constants, only the objects with matching values are candidates for further evaluation. The other objects are guaranteed to make the filter return false, but never fail. The (small) set of candidate objects can often be found by looking up some key values of the table or by traversing the table using a match specification. It is necessary to place the guard filters immediately after the table’s generator, otherwise it could happen that some table object that would make the query evaluation fail is excluded by looking up a key or running a match specification.

Join

QLC supports fast join of two query handles. Fast join is possible if some position (P1) of one query handler is compared to or matched against some position (P2) of another query handle. Two fast join methods have been implemented:

- Lookup join traverses all objects of one query handle and finds objects of the other handle (a QLC table) such that the values at P1 and P2 match. QLC does not create any indices but looks up values using the key position and the indexed positions of the QLC table.
- Merge join sorts the objects of each query handle if necessary and filters out objects where the values at P1 and P2 do not compare equal. If there are many objects with the same value of P2 a temporary file will be used for the equivalence classes.

QLC warns at compile time if a query list comprehension combines QLC handles in such a way that more than one join is possible. In other words, there is no query planner that can choose a good order between possible join operations. It is up to the user to order the joins by introducing query handles.

The join is to be expressed as a guard filter. The filter must be placed immediately after the two joined generators, possibly after guard filters that use variables from no other generators but the two joined generators. QLC inspects the operands of =:=/2, ==/2,
is_record/2, element/2, and logical operators (and/2, or/2, andalso/2, orelse/2, xor/2) when determining which joins to consider.

Common options

The following options are accepted by cursor/2, eval/2, fold/4, and info/2:

- \{cache_all, CacheAll\} where CacheAll is equal to ets or list adds a \{cache,CacheAll\} option to every list expression of the query except tables and lists. Default is \{cache_all, no\}. The option cache_all is equivalent to \{cache_all, ets\}.

- \{max_list_size, MaxListSize\} where MaxListSize is the size in bytes of terms on the external format. If the accumulated size of collected objects exceeds MaxListSize the objects are written onto a temporary file. This option is used by the \{cache, list\} option as well as by the merge join method. Default is 512*1024 bytes.

- \{tmpdir, TempDirectory\} sets the directory used by merge join for temporary files and by the \{cache, list\} option. The option also overrides the <c>tmpdir option of keysort/3 and sort/2. The default value is "" which means that the directory returned by file:get_cwd() is used.

- \{unique_all, true\} adds a \{unique, true\} option to every list expression of the query. Default is \{unique_all, false\}. The option unique_all is equivalent to \{unique_all, true\}.

Common data types

- QueryCursor = \{qlc,cursor, term()\}
- QueryHandle = \{qlc,handle, term()\}
- QueryHandleOrList = QueryHandle | list()
- Answers = [Answer]
- Answer = term()
- AbstractExpression = -parse trees for Erlang expressions, see the [abstract format] documentation in ERTS User's Guide-
- MatchExpression = -match specifications, see the [match specification] documentation in the ERTS User's Guide and ms_transform(3) [page 238]-
- SpawnOptions = default | spawn options()
- SortOptions = [SortOption] | SortOption
- SortOption = \{compressed, bool()\} | \{no_files, NoFiles\} | \{order, Order\} | \{size, Size\} | \{tmpdir, TempDirectory\} | \{unique, bool()\} -see file_sorter(3) [page 148]-
- Order = ascending | descending | OrderFun
- OrderFun = fun(Term, Term) -> bool()
- TempDirectory = "" | filename()
- Size = int() > 0
- NoFiles = int() > 1
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

\[ X \left\lvert Y \left\lvert L \right\rceil , X \left\lceil Y \right\rceil \]

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 270] describes the implementation of a function calling qlc:table/2.

Besides qlc:table/2 there are other functions that return query handles. They might not be used as often as tables, but are useful from time to time. qlc:append traverses objects from several tables or lists after each other. If, for instance, you want to traverse all answers to a query QH and then finish off by a term \{finished\}, you can do that by calling qlc:append(QH, \{\{finished\}\}). append first returns all objects of QH, then \{finished\}. If there is one tuple \{finished\} among the answers to QH it will be returned twice from append.

As another example, consider concatenating the answers to two queries QH1 and QH2 while removing all duplicates. The means to accomplish this is to use the unique option:

qlc:q([X \left\lvert X \left\lceil \text{qlc:append(QH1, QH2)} \right\rceil, \{unique, true\}] )
The cost is substantial: every returned answer will be stored in an ETS table. Before returning an answer it is looked up in the ETS table to check if it has already been returned. Without the *unique* options all answers to QH1 would be returned followed by all answers to QH2. The *unique* options keeps the order between the remaining answers.

If the order of the answers is not important there is the alternative to sort the answers uniquely:

```erlang
qlc:sort(qlc:q([X || X <- qlc:append(QH1, QH2)], {unique, true})).
```

This query also removes duplicates but the answers will be sorted. If there are many answers temporary files will be used. Note that in order to get the first unique answer all answers have to be found and sorted.

To return just a few answers cursors can be used. The following code returns no more than five answers using an ETS table for storing the unique answers:

```erlang
c = qlc:cursor(qlc:q([X || X <- qlc:append(QH1, QH2)],{unique,true})),
R = qlc:next_answers(C, 5),
ok = qlc:delete_cursor(C),
R.
```

Query list comprehensions are convenient for stating constraints on data from two or more tables. An example that does a natural join on two query handles on position 2:

```erlang
qlc:q([{X1,X2,X3,Y1} ||
   {X1,X2,X3} <- QH1,
   {Y1,Y2} <- QH2,
   X2 =:= Y2])
```

QLC will evaluate this differently depending on the query handles QH1 and QH2. If, for example, X2 is matched against the key of a QLC table the lookup join method will traverse the objects of QH2 while looking up key values in the table. On the other hand, if neither X2 nor Y2 is matched against the key or an indexed position of a QLC table, the merge join method will make sure that QH1 and QH2 are both sorted on position 2 and next do the join by traversing the objects one by one.

The *join* option can be used to force QLC to use a certain join method. For the rest of this section it is assumed that the excessively slow join method called “nested loop” has been chosen:

```erlang
qlc:q([{X1,X2,X3,Y1} ||
   {X1,X2,X3} <- QH1,
   {Y1,Y2} <- QH2,
   X2 =:= Y2],
   {join, nested_loop})
```

In this case the filter will be applied to every possible pair of answers to QH1 and QH2, one at a time. If there are M answers to QH1 and N answers to QH2 the filter will be run M*N times.

If QH2 is a call to the function for gb_trees as defined in the Implementing a QLC table [page 270] section, gb_table:table/1, the iterator for the gb-tree will be initiated for each answer to QH1 after which the objects of the gb-tree will be returned one by one. This is probably the most efficient way of traversing the table in that case since it takes minimal computational power to get the following object. But if QH2 is not a table but a more complicated QLC, it can be more efficient use some RAM memory for collecting the answers in a cache, particularly if there are only a few answers. It must then be assumed that evaluating QH2 has no side effects so that the
meaning of the query does not change if Q H 2 is evaluated only once. One way of caching the answers is to evaluate Q H 2 first of all and substitute the list of answers for Q H 2 in the query. Another way is to use the cache option. It is stated like this:

\[
QH2' = \text{qlc:q}([X \mid X \leftarrow QH2], \{\text{cache}, \text{ets}\})
\]

or just

\[
QH2' = \text{qlc:q}([X \mid X \leftarrow QH2], \text{cache})
\]

The effect of the cache option is that when the generator Q H 2’ is run the first time every answer is stored in an ETS table. When next answer of Q H 1 is tried, answers to Q H 2’ are copied from the ETS table which is very fast. As for the unique option the cost is a possibly substantial amount of RAM memory. The \{cache, list\} option offers the possibility to store the answers in a list on the process heap. While this has the potential of being faster than ETS tables since there is no need to copy answers from the table it can often result in slower evaluation due to more garbage collections of the process' heap as well as increased RAM memory consumption due to bigger heaps. Another drawback with cache lists is that if the size of the list exceeds a limit a temporary file will be used. Reading the answers from a file is very much slower than copying them from an ETS table. But if the available RAM memory is scarce setting the limit to some low value is an alternative.

There is an option all that can be set to ets or list when evaluating a query. It adds a cache or ets 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 function the implementation of a QLC table for the gb_trees 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,
    ..., end,
```

270
FormatFun =
    fun(all) ->
      Vals = a_few(T),
      {gb_trees, from_orddict, [Vals]};
      ({lookup, 1, KeyValues}) ->
        ValsS = io_lib:format("gb_trees:from_orddict(~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(_, 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. If the query has the form
and P is a tuple, the qlc module analyzes P and F in compile time to find positions of the tuple P that are matched or compared to constants. If such a position at runtime turns out to be the key position, the lookup function can be used, otherwise all objects of the table have to be traversed. It is the info function InfoFun that returns the key position. There can be indexed positions as well, also returned by the info function. An index is an extra table that makes lookup on some position fast. Mnesia maintains indices upon request, thereby introducing so called secondary keys. The key is always preferred before secondary keys regardless of the number of constants to look up.

Exports

append(QHL) -\(\rightarrow\) QH

Types:
- \(\text{QHL = \{\text{QueryHandleOrList}\}}\)
- \(\text{QH = QueryHandle}\)

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) -\(\rightarrow\) QH3

Types:
- \(\text{QH1 = QH2 = \{\text{QueryHandleOrList}\}}\)
- \(\text{QH3 = QueryHandle}\)

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]) -\(\rightarrow\) QueryCursor

Types:
- \(\text{Options = \{\text{Option}\}}\)|\text{Option}
- \(\text{Option = \{\text{cache\_all, Cache\}}|\text{cache\_all}|\{\text{max\_list\_size, MaxList\_Size}\}|\{\text{spawn\_options, Spawn\_Options}\}|\{\text{tmpdir, Temp\_Directory}\}|\{\text{unique\_all, bool()}\}}\)

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].
 delet_cursor(QueryCursor) -> ok

Deletes a query cursor. Only the owner of the cursor can delete the cursor.

eval(QueryHandleOrList [, Options]) -> Answers | Error
e(QueryHandleOrList [, Options]) -> Answers

 Types:
  - Options = [Option] | Option
  - Option = {cache_all, Cache} | cache_all | {max_list_size, MaxListSize} | {tmpdir, TempDirectory} | {unique_all, bool()} | unique_all
  - Error = {error, module(), Reason}
  - Reason = as returned by filesorter(3)

Evaluates a query handle in the calling process and collects all answers in a list.

fold(Function, Acc0, QueryHandleOrList [, Options]) -> Acc1 | Error

 Types:
  - Function = fun(Answer, AccIn) -> AccOut ut
  - Acc0 = Acc1 = AccOut = term()
  - Options = [Option] | Option
  - Option = {cache_all, Cache} | cache_all | {max_list_size, MaxListSize} | {tmpdir, TempDirectory} | {unique_all, bool()} | unique_all
  - Error = {error, module(), Reason}
  - Reason = as returned by filesorter(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.
- Error = {error, module(), term()}
- Chars = [char() | Chars]

Returns a descriptive string in English of an error tuple returned by some of the functions of the qlc module or the parse transform. This function is mainly used by the compiler invoking the parse transform.

info(QueryHandleOrList [, Options]) -> Info

Types:
- Options = [Option] | Option
- Option = EvalOption | ReturnOption
- EvalOption = {cache_all, Cache} | cache_all | {max_list_size, MaxListSize} | {tmpdir, TempDirectory} | {unique_all, bool()} | unique_all
- ReturnOption = {flat, bool()} | {format, Format} | {n_elements, NElements}
- Format = abstract | 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\} 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.

1> QH = qlc:q([[X,Y] || 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 =:= Y, Y =:= V2], \{unique,true\})
   end

In this example two simple QLCs have been inserted just to hold the \{unique,true\} option.

1> E1 = ets:new(e1, []),
   E2 = ets:new(e2, []),
   true = ets:insert(E1, [{1,a},{2,b}]),
   true = ets:insert(E2, [{a,i},{b,2}]),
   Q = qlc:q([[X,Z,W] ||
              \{X,Z\} =:= ets:table(E1),
              \{W,Y\} =:= ets:table(E2),
              X =:= Y]),
   io:format("s~n", [qlc:info(Q)]).
   begin
     V1 = qlc:q([P0 || P0 =:= \{W,Y\} =:= ets:table(18)]),
     V2 =
In this example the query list comprehension \( V_2 \) has been inserted to show the joined generators and the join method chosen. A convention is used for lookup join: the first generator \( (G_2) \) is the one traversed, the second one \( (G_1) \) is the table where constants are looked up.

keysort(KeyPos, QH1 \[, SortOptions\]) \(-\) QH2

Types:
- QH1 = QueryHandleOrList
- QH2 = QueryHandle

Returns a query handle. When evaluating the query handle QH2 the answers to the query handle QH1 are sorted by file \( \text{sorter:} \text{keysort/4} \) \{page 148\} 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 \( \text{Size} \) bytes, where \( \text{Size} \) is the value of the size option.

next_answers(QueryCursor \[, NumberofAnswers\]) \(-\) Answers | Error

Types:
- NumberofAnswers = all_remaining | int() \( >0 \)
- Error = \{error, module(), Reason\}
- Reason = \{as returned by file \( \text{sorter(3)} \)-

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 \( \text{next answers} \) will return \([]\).

q(QueryListComprehension \[, Options\]) \(-\) QueryHandle

Types:
- QueryListComprehension = -\{literal query list comprehension\}-
- Options = \{Option\} | Option
- Option = \{\( \text{maxLookup, MaxLookup} \) | \{cache, Cache\} | cache | \{join, Join\} | \{lookup, Lookup\} | \{unique, bool()\} | unique
- MaxLookup = int() \( \geq0 \) | infinity
- Join = any | lookup | merge | nested_loop
- Lookup = bool() | any

Returns a query handle for a query list comprehension. The query list comprehension must be the first argument to qlc:q/1,2 or it will be evaluated as an ordinary list comprehension. It is also necessary to add the line

\[-\text{include_lib("stdlib/include/qlc.hrl")}.\]
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:

A = [X || X <- [{1},{2}]],
QH = qlc:q(A),

...  

The variable A will be bound to the evaluated value of the list comprehension ([1,2]).
The compiler complains with an error message ("argument is not a query list
comprehension"); the shell process stops with a badarg reason.

The {cache,ets} option can be used to cache the answers to a query list
comprehension. The answers are stored in one ETS table for each cached query list
comprehension. When a cached query list comprehension is evaluated again, answers
are fetched from the table without any further computations. As a consequence, when
all answers to a cached query list comprehension have been found, the ETS tables used
for caching answers to the query list comprehension's qualifiers can be emptied. The
option cache is equivalent to {cache,ets}.

The {cache,list} option can be used to cache the answers to a query list
comprehension just like {cache,ets}. The difference is that the answers are kept in a
list (on the process heap). If the answers would occupy more than a certain amount of
RAM memory a temporary file is used for storing the answers. The option
max_list_size sets the limit in bytes and the temporary file is put on the directory set
by the tmpdir option.

The cache option has no effect if it is known that the query list comprehension will be
evaluated at most once. This is always true for the top-most query list comprehension
and also for the list expression of the first generator in a list of qualifiers. Note that in
the presence of side effects in filters or callback functions the answers to query list
comprehensions can be affected by the cache option.

The {unique,true} option can be used to remove duplicate answers to a query list
comprehension. The unique answers are stored in one ETS table for each query list
comprehension. The table is emptied every time it is known that there are no more
answers to the query list comprehension. The option unique is equivalent to
{unique,true}. If the unique option is combined with the {cache,ets} option, two
ETS tables are used, but the full answers are stored in one table only. If the unique
option is combined with the {cache,list} option the answers are sorted twice using
keysort/3; once to remove duplicates, and once to restore the order.

The cache and unique options apply not only to the query list comprehension itself but
also to the results of looking up constants, running match specifications, and joining
handles.

1> Q = qlc:q([[A,X,Z,W] ||
            A <- [a,b,c],
            [X,Z] <- [{a,1},{b,4},{c,6}],
            [W,Y] <- [{2,a},{3,b},{4,c}],
            X =:= Y],
            {cache, list}),
begin
V1 = qlc:q([P0 ||
P0 = [X,Z] <- qlc:keysорт(1, [[a,1],[b,4],[c,6]], [])]),
V2 = qlc:q([P0 ||
P0 = [W,Y] <- qlc:keysорт(2, [[2,a],[3,b],[4,c]], [])]),
V3 = qlc:q([[G1|G2] ||
G1 <- V1, G2 <- V2, element(1, G1) == element(2, G2),
[[join,merge],[cache,list]]),
qlc:q([[A,X,Z,W] || A <- [a,b,c], [[X,Z]|[W,Y]] <- V3, X =:= Y])
end

In this example the cached results of the merge join are traversed for each value of A. Note that without the cache option the join would have been carried out three times, once for each value of A. sort/1, 2 and keysорт/2, 3 can also be used for caching answers and for removing duplicates. When sorting answers are cached in a list, possibly stored on a temporary file, and no ETS tables are used.

Sometimes (see qclc:table/2 [page 280] below) traversal of tables can be done by looking up key values, which is assumed to be fast. Under certain (rare) circumstances it could happen that there are too many key values to look up. The {max_lookup, MaxLookup} option can then be used to limit the number of lookups: if more than MaxLookup lookups would be required no lookups are done but the table traversed instead. The default value is infinity which means that there is no limit on the number of keys to look up.

1> T = gb_trees:empty(),
   QH = qlc:q([X ||
   ((X =:= 1) or (X =:= 2)),
   ((Y =:= a) or (Y =:= b) or (Y =:= c))],
io:format("\n", [qlc:info(QH)]).
   ets:match_spec_run(
   lists:flatmap(fun(K) ->
      case
gb_trees:lookup(K,
       gb_trees:from_orddict([]))
of
{value,V} ->
   [[K,V]];
   none ->
   []
end
e,[[1,a],[1,b],[1,c],[2,a],[2,b],[2,c]]).
   ets:match_spec_compile([[{'andalso'},
   {'or'},
   {'='='','$1',1},
   {'='='','$1',2}],
   {'or'},
   {'='='','$1',1},
   {'='='','$1',2}])

In this example using the `gb_table` module from the Implementing a QLC table [page 270] section there are six keys to look up: `{1,a}`, `{1,b}`, `{1,c}`, `{2,a}`, `{2,b}`, and `{2,c}`. The reason is that the two elements of the key \(X,Y\) are matched separately. The `{lookup, true}` option can be used to ensure that QLC will look up constants in some QLC table. If there are more than one QLC table among the generators' list expressions, constants have to be looked up in at least one of the tables. The evaluation of the query fails if there are no constants to look up. This option is useful in situations when it would be unacceptable to traverse all objects in some table. Setting the `lookup` option to `false` ensures that no constants will be looked up (\(\{\text{max\_lookup}, 0\}\) has the same effect). The default value is `any` which means that constants will be looked up whenever possible.

The `{join, Join}` option can be used to ensure that a certain join method will be used: `{join, lookup}` invokes the lookup join method; `{join, merge}` invokes the merge join method; and `{join, nested_loop}` invokes the method of matching every pair of objects from two handles. The last method is mostly very slow. The evaluation of the query fails if QLC cannot carry out the chosen join method. The default value is `any` which means that some fast join method will be used if possible.

```
	sort(QH1 [, SortOptions]) -> QH2

Types:
- QH1 = QueryHandleOrList
- QH2 = QueryHandle

Returns a query handle. When evaluating the query handle QH2 the answers to the query handle QH1 are sorted by file_sorter:sort/3 [page 148] according to the options. The sorter will use temporary files only if QH1 does not evaluate to a list and the size of the binary representation of the answers exceeds `Size` bytes, where `Size` is the value of the `size` option.

```

```

string_to_handle(QueryString [, Options [, Bindings]]) -> QueryHandle | Error

Types:
- QueryString = string()
- Options = [Option] | Option
- Option = {max_lookup, MaxLookup} | {cache, Cache} | cache | {join, Join} | {lookup, Lookup} | {unique, bool()} | unique
- MaxLookup = int() >= 0 | infinity
- Join = any | lookup | merge | nested_loop
- Lookup = bool() | any
- Bindings = as returned by erl_eval:bindings/1-
- Error = {error, module(), Reason}
- Reason = -ErrorInfo as returned by erl_scan:string/1 or erl_parse:parse_exprs/1-

```
A string version of \texttt{qlc:q/1,2}. When the query handle is evaluated the fun created by the parse transform is interpreted by \texttt{erl_eval(3)}. The query string is to be one single query list comprehension terminated by a period.

\begin{verbatim}
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]
\end{verbatim}

This function is probably useful mostly when called from outside of Erlang, for instance from a driver written in C.

table(TraverseFun, Options) -> QueryHandle

Types:
- TraverseFun = TraverseFun0 | TraverseFun1
  - TraverseFun0 = fun() -> TraverseResult
  - TraverseFun1 = fun(MatchExpression) -> TraverseResult
- TraverseResult = O bjects | term()
- O bjects = [] | [term()] | O bjectList
- O bjectList = TraverseFun0 | O bjects
- O ptions = [Option] | O ption
- O ption = [format_fun, FormatFun] | [info_fun, InfoFun] | [lookup_fun, LookupFun] | [parent_fun, ParentFun] | [post_fun, PostFun] | [pref_fun, PreFun]
- FormatFun = undefined | fun(SelectedO bjects) -> FormattedTable
- SelectedO bjects = all | [match_spec, MatchExpression] | [lookup, {Position, Keys}]
- FormattedTable = [M od, Fun, A rgs] | A bstractExpression | character_list()
- InfoFun = undefined | fun(InfoTag) -> InfoValue
- InfoTag = indices | is_u nique_objects | keypos | num_of_objects
- InfoValue = undefined | term()
- LookupFun = undefined | fun(Position, Keys) -> LookupResult
- LookupResult = [term()] | term()
- ParentFun = undefined | fun() -> ParentFunValue
- PostFun = undefined | fun() -> void()
- PreFun = undefined | fun([PreArg]) -> void()
- PreArg = [parent_value, ParentFunValue] | [stop_fun, StopFun]
- ParentFunValue = undefined | term()
- StopFun = undefined | fun() -> void()
- Position = int() > 0
- Keys = [term()]
- M od = Fun = atom()
- A rgs = [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 \texttt{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 \texttt{qlc:table/2}.
The callback function \texttt{TraverseFun} is used for traversing the table. It is to return a list of objects terminated by either \texttt{[]} or a nullary fun to be used for traversing the not yet traversed objects of the table. Any other return value is immediately returned as value of the query evaluation. Unary \texttt{TraverseFun}s 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 \texttt{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, \texttt{TraverseFun} will be called with a match specification returning every object. Modules that can utilize match specifications for optimized traversal of tables should call \texttt{qlc:table/2} with a unary \texttt{TraverseFun} while other modules can provide a nullary \texttt{TraverseFun}. \texttt{ets:table/2} is an example of the former; \texttt{gb_table:table/1} in the Implementing a QLC table [page 270] section is an example of the latter.

\texttt{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 \texttt{PostFun} is called once after the table was last read. The return value, which is caught, is ignored. If \texttt{PreFun} has been called for a table, \texttt{PostFun} is guaranteed to be called for that table, even if the evaluation of the query fails for some reason. The order in which pre (post) functions for different tables are evaluated is not specified. Other table access than reading, such as calling \texttt{InfoFun}, is assumed to be OK at any time. The argument \texttt{PreArgs} is a list of tagged values. Currently there are two tags, \texttt{parent} and \texttt{stop}, used by Mnesia for managing transactions. The value of \texttt{parent} is the value returned by \texttt{ParentFun}, or undefined if there is no \texttt{ParentFun}. \texttt{ParentFun} is called once just before the call of \texttt{PreFun} in the context of the process calling \texttt{eval}, \texttt{fold}, or \texttt{cursor}. The value of \texttt{stop} is a nullary fun that deletes the cursor if called from the parent, or undefined if there is no cursor.

The binary callback function \texttt{LookupFun} is used for looking up objects in the table. The first argument \texttt{Position} is the key position or an indexed position and the second argument \texttt{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 \texttt{Position} is a member of \texttt{Keys}. Any other return value is immediately returned as value of the query evaluation. \texttt{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 \texttt{Position} in such a way that only \texttt{Keys} need to be looked up in order to find all potential answers. The key position is obtained by calling \texttt{InfoFun(keypos)} and the indexed positions by calling \texttt{InfoFun(indices)}. If the key position can be used for lookup it is always chosen, otherwise the indexed position requiring the least number of lookups is chosen. If there is a tie between two indexed positions the one occurring first in the list returned by \texttt{InfoFun} is chosen. Positions requiring more than max_lookup [page 277] lookups are ignored.

The unary callback function \texttt{InfoFun} is to return information about the table. \texttt{undefined} should be returned if the value of some tag is unknown:

- \texttt{indices}. Returns a list of indexed positions, a list of positive integers.
- \texttt{is_unique_objects}. Returns true if the objects returned by \texttt{TraverseFun} are unique.
- \texttt{keypos}. Returns the position of the table’s key, a positive integer.
- \texttt{is_sorted_key}. Returns true if the objects returned by \texttt{TraverseFun} are sorted on the key.
- \texttt{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 146], `dets(3)` [page 85] and `mnesia(3)` for the various options recognized by `table/1,2` in respective module.

**See Also**

`dets(3)` [page 71], [Erlang Reference Manual], `erl_eval(3)` [page 105], [erlang(3)], `ets(3)` [page 128], [file(3)], [file_sorter(3)] [page 148], [mnemosyne(3)], [mnesia(3)], [Programming Examples], shell(3) [page 296]
This module implements FIFO queues in an efficient manner. All operations have 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

#### \texttt{cons(Item, Q1) \rightarrow Q2}

Types:
- \texttt{Item = term()}
- \texttt{Q1 = Q2 = queue()}
Inserts \texttt{Item} at the head of queue \texttt{Q1}. Returns the new queue \texttt{Q2}.

#### \texttt{daeh(Q) \rightarrow Item}

The same as \texttt{last(Q)} and the opposite of \texttt{head(Q)}.

#### \texttt{from_list(L) \rightarrow queue()}

Types:
- \texttt{L = list()}
Returns a queue containing the items in \texttt{L}, in the same order - the head item of the list will be the head item of the queue.

#### \texttt{head(Q) \rightarrow Item}

Types:
- \texttt{Item = term()}
- \texttt{Q = queue()}
Returns \texttt{Item} from the head of queue \texttt{Q}. Fails with reason \texttt{empty} if \texttt{Q} is empty.

#### \texttt{in(Item, Q1) \rightarrow Q2}

Types:
- \texttt{Item = term()}
- \texttt{Q1 = Q2 = queue()}

Inserts Item at the tail of queue Q1. Returns a new queue Q2. This is the same as snoc(Q1, Item).

\texttt{in_r(Item, Q1)} -> Q2

Types:
- Item = \text{term()}
- Q_1 = Q_2 = \text{queue()}

Inserts Item at the head of queue Q1. Returns a new queue Q2. This is the same as cons(Item, Q1).

\texttt{init(Q1)} -> Q2

Types:
- Item = \text{term()}
- Q_1 = Q_2 = \text{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.

\texttt{is_empty(Q)} -> true | false

Types:
- Q = \text{queue()}

Tests if Q is empty and returns true if so and false otherwise.

\texttt{join(Q1, Q2)} -> Q3

Types:
- Q_1 = Q_2 = Q_3 = \text{queue()}

Returns a queue Q3 that is the result of joining Q1 and Q2 with Q1 before (at the head) Q2.

\texttt{lait(Q1)} -> Q2

The same as \texttt{init(Q1)} and the opposite of tail(Q1).

\texttt{last(Q)} -> Item

Types:
- Item = \text{term()}
- Q = \text{queue()}

Returns the last item of queue Q. This is the opposite of head(Q).
Fails with reason empty if Q is empty.

\texttt{len(Q)} -> N

Types:
- Q = \text{queue()}
- N = \text{integer()}
Calculates and returns the length of queue Q.

\textbf{new(\textit{Q}) -> Q}

Types:
- \textit{Q} = queue()

Returns an empty queue.

\textbf{out(\textit{Q1}) -> Result}

Types:
- Result = \{(\text{value, Item}), Q2\} \mid \{\text{empty, Q1}\}
- Q1 = Q2 = queue()

Removes the head item from the queue \textit{Q1}. Returns the tuple \{(\text{value, Item}), Q2\}, where Item is the item removed and Q2 is the new queue. If \textit{Q1} is empty, the tuple \{\text{empty, Q1}\} is returned.

\textbf{out\textunderscore r(\textit{Q1}) -> Result}

Types:
- Result = \{(\text{value, Item}), Q2\} \mid \{\text{empty, Q1}\}
- Q1 = Q2 = queue()

Removes the last item from the queue \textit{Q1}. Returns the tuple \{(\text{value, Item}), Q2\}, where Item is the item removed and Q2 is the new queue. If \textit{Q1} is empty, the tuple \{\text{empty, Q1}\} is returned.

\textbf{reverse(\textit{Q1}) -> Q2}

Types:
- Q1 = Q2 = queue()

Returns a queue Q2 that contains the items of \textit{Q1} in the reverse order.

\textbf{snoc(\textit{Q1}, \textit{Item}) -> Q2}

Types:
- Item = term()
- Q1 = Q2 = queue()

Inserts \textit{Item} as the last item of queue \textit{Q1}. Returns the new queue \textit{Q2}. This is the opposite of \texttt{cons(Item, Q1)}.

\textbf{split(\textit{N}, \textit{Q1}) -> \{Q2, Q3\}}

Types:
- \textit{N} = integer()
- Q1 = Q2 = Q3 = queue()

Splits \textit{Q1} into a queue Q2 of length \textit{N} with items from the head end, and a queue Q3 with the rest of the items.

\textbf{tail(\textit{Q1}) -> Q2}
Types:
- Item = term()
- Q1 = Q2 = queue()

Returns a queue Q2 that is the result of removing the head item from Q1.
Fails with reason empty if Q1 is empty.

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_s/0 or uniform_s/1 without setting a seed first, seed/0 is called automatically.
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,erordesc()}
- 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,erordesc()}
- Start = Length = integer()

Finds the first match of the regular expression RegExp in String. This call is usually faster than match and it is also a useful way to ascertain that a match exists. It returns as follows:

{match,Start,Length} if the match succeeded. Start is the starting position of the match and Length is the length of the matching string.

nomatch if there were no matching characters.

{error,Error} if there was an error in RegExp.

matches(String, RegExp) -> MatchRes

Types:
- String = RegExp = string()
• MatchRes = (match, Matches) | {error, errordesc()}
• Matches = list()

Finds all non-overlapping matches of the expression RegExp in String. It returns as follows:

(match, Matches) if the regular expression was correct. The list will be empty if there was no match. Each element in the list looks like {Start, Length}, where Start is the starting position of the match, and Length is the length of the matching string.

{error, Error} if there was an error in RegExp.

sub(String, RegExp, New) -> SubRes

Types:
• String = RegExp = New = string()
• SubRes = {ok, NewString, RepCount} | {error, errordesc()}
• RepCount = integer()

Substitutes the first occurrence of a substring matching RegExp in String with the string New. A & in the string New is replaced by the matched substring of String. \& puts a literal & into the replacement string. It returns as follows:

{ok, NewString, RepCount} if RegExp is correct. RepCount is the number of replacements which have been made (this will be either 0 or 1).

{error, Error} if there is an error in RegExp.

gsub(String, RegExp, New) -> SubRes

Types:
• String = RegExp = New = string()
• SubRes = {ok, NewString, RepCount} | {error, errordesc()}
• RepCount = integer()

The same as sub, except that all non-overlapping occurrences of a substring matching RegExp in String are replaced by the string New. It returns:

{ok, NewString, RepCount} if RegExp is correct. RepCount is the number of replacements which have been made.

{error, Error} if there is an error in RegExp.

split(String, RegExp) -> SplitRes

Types:
• String = RegExp = string()
• SubRes = {ok, FieldList} | {error, errordesc()}
• FieldList = [string()]

String is split into fields (sub-strings) by the regular expression RegExp. If the separator expression is " " (a single space), then the fields are separated by blanks and/or tabs and leading and trailing blanks and tabs are discarded. For all other values of the separator, leading and trailing blanks and tabs are not discarded. It returns:
{ok, FieldList} to indicate that the string has been split up into the fields of FieldList.
{error, Error} if there is an error in RegExp.

sh_to_awk(ShRegExp) -> AwkRegExp

Types:
- ShRegExp AwkRegExp = string()
- SubRes = {ok, NewString, RepCount} | {error, errordesc()}
- RepCount = integer()

Converts the sh type regular expression ShRegExp into a full AWK regular expression. Returns the converted regular expression string. sh expressions are used in the shell for matching file names and have the following special characters:
* matches any string including the null string.
? matches any single character.
[...] matches any of the enclosed characters. Character ranges are specified by a pair of characters separated by a -. If the first character after [ is a !, then any character not enclosed is matched.

It may sometimes be more practical to use sh type expansions as they are simpler and easier to use, even though they are not as powerful.

parse(RegExp) -> ParseRes

Types:
- RegExp = string()
- ParseRes = {ok, RE} | {error, errordesc()}

 Parses the regular expression RegExp and builds the internal representation used in the other regular expression functions. Such representations can be used in all of the other functions instead of a regular expression string. This is more efficient when the same regular expression is used in many strings. It returns:

{ok, RE} if 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....
r1 | r2 alternation. It matches either r1 or r2.
r1r2 concatenation. It matches r1 and then r2.
r+ matches one or more rs.
r* matches zero or more rs.
r? matches zero or one rs.
(r) grouping. It matches r.

The escape sequences allowed are the same as for Erlang strings:
\b backspace
\f form feed
\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 "...\n". The following examples define Erlang data types:
Atoms [a-z][0-9a-zA-Z]*
Variables [A-Z][0-9a-zA-Z]*
Floats (\+\-)?[0-9]+\.\.[0-9]+((E|e)(\+\-)?[0-9]+)?
Regular expressions are written as Erlang strings when used with the functions in this module. This means that any `\` or `"` characters in a regular expression string must be written with `\` as they are also escape characters for the string. For example, the regular expression string for Erlang floats is:

```
"(\+|-)?[0-9]+\.[0-9]+((E|e)(\+|-)?[0-9]+)".
```

It is not really necessary to have the escape sequences as part of the regular expression syntax as they can always be generated directly in the string. They are included for completeness and can 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

new() -> Set
Types:
• Set = set()
Returns a new empty ordered set.

is_set(Set) -> bool()
Types:
• Set = term()
Returns true if Set is an ordered set of elements, otherwise false.

size(Set) -> int()
Types:
• Set = term()
Returns the number of elements in Set.

to_list(Set) -> List
Types:
• Set = set()
• List = [term()]
Returns the elements of Set as a list.

from_list(List) -> Set
Types:
• List = [term()]
• Set = set()
Returns an ordered set of the elements in List.

is_element(Element, Set) -> bool()
Types:
  - Element = term()
  - Set = set()
Returns true if Element is an element of Set, otherwise false.

`add_element(Element, Set1) -> Set2`
Types:
  - Element = term()
  - Set1 = Set2 = set()
Returns a new ordered set formed from Set1 with Element inserted.

`del_element(Element, Set1) -> Set2`
Types:
  - Element = term()
  - Set1 = Set2 = set()
Returns Set1, but with Element removed.

`union(Set1, Set2) -> Set3`
Types:
  - Set1 = Set2 = Set3 = set()
Returns the merged (union) set of Set1 and Set2.

`union(SetList) -> Set`
Types:
  - SetList = [set()]
  - Set = set()
Returns the merged (union) set of the list of sets.

`intersection(Set1, Set2) -> Set3`
Types:
  - Set1 = Set2 = Set3 = set()
Returns the intersection of Set1 and Set2.

`intersection(SetList) -> Set`
Types:
  - SetList = [set()]
  - Set = set()
Returns the intersection of the non-empty list of sets.

`subtract(Set1, Set2) -> Set3`
Types:
  - Set1 = Set2 = Set3 = set()
Returns only the elements of Set1 which are not also elements of Set2.

\[\text{is\_subset(Set1, Set2)} \rightarrow \text{bool()}\]

Types:
- Set1 = Set2 = set()
Returns true when every element of Set1 is also a member of Set2, otherwise false.

\[\text{fold(Function, Acc0, Set)} \rightarrow \text{Acc1}\]

Types:
- Function = fun (E, AccIn) \rightarrow AccOut
- Acc0 = Acc1 = AccIn = AccOut = term()
- Set = set()
Fold Function over every element in Set returning the final value of the accumulator.

\[\text{filter(Pred, Set1)} \rightarrow \text{Set2}\]

Types:
- Pred = fun (E) \rightarrow bool()
- Set1 = Set2 = set()
Filter elements in Set1 with boolean function Fun.

**See Also**

ordsets(3) [page 250], gb_sets(3) [page 162]
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

```
-include_lib("kernel/include/file.hrl").
```

to `user_default` makes the definition of `file_info` readily available in the shell.

The shell runs in two modes:

- Normal (possibly restricted) mode, in which commands can be edited and expressions evaluated.
- Job Control Mode JCL, in which jobs can be started, killed, detached and connected.

Only the currently connected job can ‘talk’ to the shell.
Shell Commands

b() Prints the current variable bindings.

f() Removes all variable bindings.

f(X) Removes the binding of variable X.

h() Prints the history list.

history(N) Sets the number of previous commands to keep in the history list to N. The previous number is returned. The default number is 20.

results(N) Sets the number of results from previous commands to keep in the history list to N. The previous number is returned. The default number is 20.

e(N) Repeats the command N, if N is positive. If it is negative, the nth previous command is repeated (i.e., e(-1) repeats the previous command).

v(N) Uses the return value of the command N in the current command, if N is positive. If it is negative, the return value of the nth previous command is used (i.e., v(-1) uses the value of the previous command).

help() Evaluates shell_default:help().

c(File) Evaluates shell_default:c(File). This compiles and loads code in File and purges old versions of code, if necessary. Assumes that the file and module names are the same.

rd(RecordName, RecordDefinition) Defines a record in the shell. RecordName is an atom and RecordDefinition lists the field names and the default values. Usually record definitions are made known to the shell by use of the rr commands described below, but sometimes it is handy to define records on the fly.

rf() Removes all record definitions, then reads record definitions from the modules shell_default and user_default (if loaded). Returns the names of the records defined.

rf(RecordNames) Removes selected record definitions. RecordNames is a record name or a list of record names. Use '_' to remove all record definitions.

rl() Prints all record definitions.

rl(RecordNames) Prints selected record definitions. RecordNames is a record name or a list of record names.

rp(Term) Prints a term using the record definitions known to the shell. All of Term is printed; the depth is not limited as is the case when a return value is printed.

rr(Module) Reads record definitions from a module's BEAM file. If there are no record definitions in the BEAM file, the source file is located and read instead. Returns the names of the record definitions read. Module is an atom.

rr(Wildcard) Reads record definitions from files. Existing definitions of any of the record names read are replaced. Wildcard is a wildcard string as defined in filelib(3) but not an atom.

rr(WildcardOrModule, RecordNames) Reads record definitions from files but discards record names not mentioned in RecordNames (a record name or a list of record names).

rr(WildcardOrModule, RecordNames, Options) Reads record definitions from files. The compiler options {i,Dir}, {d,Macro}, and {d,Macro,Value} are recognized and used for setting up the include path and macro definitions. Use '_' as value of RecordNames to read all record definitions.
Example

The following example is a long dialogue with the shell. Commands starting with > are inputs to the shell. All other lines are output from the shell. All commands in this example are explained at the end of the dialogue.

```
strider 1> erl
Erlang (BEAM) emulator version 5.3 [hipe] [threads:0]

Eshell V5.3 (abort with ^G)
1> Str = "abcd".
   "abcd"
2> L = length(Str).
   4
3> Descriptor = [L, list_to_atom(Str)].
   [4,abcd]
4> L.
   4
5> b().
   Descriptor = [4,abcd]
   Str = "abcd"
   ok
6> f(L).
   ok
7> b().
   Descriptor = [4,abcd]
   Str = "abcd"
   ok
8> f(L).
   ** 1: variable 'L' is unbound **
9> [L, _] = Descriptor.
   [4,abcd]
10> L.
   4
   ** exited: [{badmatch,[4,abcd]},{erl_eval,expr,3}]] **
12> P.
   ** 1: variable 'P' is unbound **
13> Descriptor.
   [4,abcd]
14> [P, Q] = Descriptor.
   [4,abcd]
15> P.
   4
16> f().
   ok
17> put(aa, hello).
   undefined
18> get(aa).
   hello
19> Y = test1:demo(1).
   11
```
20> get().
    [{aa, worked}]
21> put(aa, hello).
    worked
22> Z = test1:demo(2).

=ERROR REPORT==== 19-Feb-2003::10:04:14 ===
Error in process <0.40.0> with exit value: 
  [{badmatch,1},
   {test1, demo, 1},
   {erl_eval, expr, 4},
   {shell, eval_loop, 2}]
** exited: 
  [{badmatch,1},
   {test1, demo, 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: 
    [undefined,
     {test1, demo, 1},
     {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().
strider 2>

Comments

Command 1 sets the variable Str to the string "abcd".
Command 2 sets L to the length of the string evaluating the BIF atom_to_list.
Command 3 builds the tuple Descriptor.
Command 4 prints the value of the variable L.
Command 5 evaluates the internal shell command b(), which is an abbreviation of "bindings". This prints the current shell variables and their bindings. The ok at the end is the return value of the b() function.
Command 6 f(L) evaluates the internal shell command f(L) (abbreviation of "forget"). The value of the variable L is removed.
Command 7 prints the new bindings.
Command 8 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 \( \text{Descriptor} \) 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 \texttt{test1:demo(X)} is defined in the following way:

\[
\text{demo}(X) \rightarrow \\
\quad \text{put}(aa, \text{worked}), \\
\quad X = 1, \\
\quad X + 10.
\]

Commands 17 and 18 set and inspect the value of the item \( aa \) in the process dictionary.

Command 19 evaluates \texttt{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 \texttt{hello} and call \texttt{test1:demo(2)}. Evaluation fails and the changes made to the dictionary in \texttt{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 \texttt{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 \texttt{e(28)}. This re-evaluates command 28. Command 30 is \texttt{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 \texttt{ex.erl} defines a record like this:

\[
-\text{record}(rec, \{a, b = \text{val()}})).
\]

\[
\text{val()} \rightarrow \\
\quad 3.
\]

Commands 31 and 32 compiles the file \texttt{ex.erl} and reads the record definitions in \texttt{ex.beam}. If the compiler did not output any record definitions on the BEAM file, \texttt{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 \( \text{val/0} \) 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 \texttt{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 \texttt{rec} has been carried out.

For the next command, it is assumed that \texttt{test1:loop(N)} is defined in the following way:

\begin{verbatim}
loop(N) ->
    io:format("Hello Number: ~w~n", [N]),
    loop(N+1).
\end{verbatim}

Command 41 evaluates \texttt{test1:loop(0)}, which puts the system into an infinite loop. At this point the user types \texttt{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 \texttt{i} command ("interrupt") is used to terminate the looping program, and the \texttt{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 \texttt{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 \texttt{^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:

\begin{verbatim}
---> ?
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
\end{verbatim}

The JCL commands have the following meaning:

\begin{itemize}
    \item \texttt{c [nn]} Connects to job number \texttt{<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.
    \item \texttt{i [nn]} Stops the current evaluator process for job number \texttt{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.
\end{itemize}
**k** [nn] Kills job number nn or the current job. All spawned processes in the job are killed, provided they have not evaluated the group_leader/1 BIF and are located on the local machine. Processes spawned on remote nodes will not be killed.

**j** Lists all jobs. A list of all known jobs is printed. The current job name is prefixed with "*".

**s** Starts a new job. This will be assigned the new index [nn] which can be used in references.

**r** [node] Starts a remote job on node. This is used in distributed Erlang to allow a shell running on one node to control a number of applications running on a network of nodes.

**q** Quits Erlang. Note that this option is disabled if Erlang is started with the ignore_break, +Bi, system flag (which may be useful e.g. when running a restricted shell, see below).

**?** Displays this message.

It is possible to alter the behavior of shell escape by means of the stdlib application variable shell_esc. The value of the variable can be either jcl (erl -stdlib shell_esc jcl) or abort (erl -stdlib shell_esc abort). The first option sets "G to activate JCL mode (which is also default behavior). The latter sets "G to terminate the current shell and start a new one. JCL mode cannot be invoked when shell_esc is set to abort.

If you want an Erlang node to have a remote job active from the start (rather than the default local job), you start Erlang with the -remsh flag. Example: erl -sname this_node -remsh other_node@other_host

### Restricted Shell

The shell may be started in a restricted mode. In this mode, the shell evaluates a function call only if allowed. This feature makes it possible to, for example, prevent a user from accidentally calling a function from the prompt that could harm a running system (useful in combination with the the system flag +Bi).

When the restricted shell evaluates an expression and encounters a function call, it calls a callback function (with information about the function call in question). This callback function returns true to let the shell go ahead with the evaluation, or false to abort it. There are two possible callback functions for the user to implement:

```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
nonlocal_allowed(FuncSpec, ArgList, State) -> {true,NewState} | {false,NewState} | [{redirect,NewFuncSpec,NewArgList},NewState]
```

to determine if the call to non-local function FuncSpec (Module,Func) or a fun) with arguments ArgList should be allowed. The return value [{redirect,NewFuncSpec,NewArgList}] can be used to let the shell evaluate some other function than the one specified by FuncSpec and ArgList.

These callback functions are in fact called from local and non-local evaluation function handlers, described in the erl_eval [page 105] manual page. (Arguments in ArgList are evaluated before the callback functions are called).
The `State` argument is a tuple `{ShellState, ExprState}`. The return value `NewState` has the same form. This may be used to carry a state between calls to the callback functions. Data saved in `ShellState` lives through an entire shell session. Data saved in `ExprState` lives only through the evaluation of the current expression.

There are two ways to start a restricted shell session:

- Use the stdlib application variable `restricted_shell` and specify, as its value, the name of the callback module. Example (with callback functions implemented in `callback_mod.erl`):
  ```
  $ erl -stdlib restricted shell callback_mod
  ```
- From a normal shell session, call function `shell:start_restricted/1`. This exits the current evaluator and starts a new one in restricted mode.

**Notes:**

- When restricted shell mode is activated or deactivated, new jobs started on the node will run in restricted or normal mode respectively.
- If restricted mode has been enabled on a particular node, remote shells connecting to this node will also run in restricted mode.
- The callback functions cannot be used to allow or disallow execution of functions called from compiled code (only functions called from expressions entered at the shell prompt).

### Exports

- **history(N) -> integer()**
  - *Types:*
  - N = integer()  
  - Sets the number of previous commands to keep in the history list to N. The previous number is returned. The default number is 20.

- **results(N) -> integer()**
  - *Types:*
  - N = integer()  
  - Sets the number of results from previous commands to keep in the history list to N. The previous number is returned. The default number is 20.

- **start_restricted(Module) -> ok**
  - *Types:*
  - Module = atom()  
  - Exits a normal shell and starts a restricted shell. Module specifies the callback module for the functions `local_allowed/3` and `non_local_allowed/3`. The function is meant to be called from the shell.

- **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.
This module provides functions for starting Erlang slave nodes. All slave nodes which are started by a master will terminate automatically when the master terminates. All TTY output produced at the slave will be sent back to the master node. File I/O is done via the master.

Slave nodes on other hosts than the current one are started with the program `rsh`. The user must be allowed to `rsh` to the remote hosts without being prompted for a password. This can be arranged in a number of ways (refer to the `rsh` documentation for details). A slave node started on the same host as the master inherits certain environment values from the master, such as the current directory and the environment variables. For what can be assumed about the environment when a slave is started on another host, read the documentation for the `rsh` program.

An alternative to the `rsh` program can be specified on the command line to `erl` as follows: `-rsh Program`.

The slave node should use the same file system at the master. At least, Erlang/OTP should be installed in the same place on both computers and the same version of Erlang should be used.

Currently, a node running on Windows NT can only start slave nodes on the host on which it is running.

The master node must be alive.

### Exports

```erlang
start(Host) ->
start(Host, Name) ->
start(Host, Name, Args) -> {ok, Node} | {error, Reason}
```

Types:
- `Host = Name = atom()`
- `Args = string()`
- `Node = node()`
- `Reason = timeout | no_rsh | {already_running, Node}`
Starts a slave node on the host \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 \cite{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:

- \texttt{Host = Name = atom()}
- \texttt{Args = string()}
- \texttt{Node = node()}
- \texttt{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:

```erlang
% 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:

```erlang
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 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 elements 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 \( \{a=a=(x,y) \text{ for some } x \in X \text{ and for some } y \in Y\} \). 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 \( xRy \). Since relations are sets, the definitions of the last paragraph (subset, union, and so on) apply to relations as well. The domain of \( R \) is the set \( \{x: xRy \text{ for some } y \in Y\} \). The range of \( R \) is the set \( \{y: xRy \text{ for some } x \in X\} \). 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 \( \{y: xRy \text{ for some } x \in A\} \). If \( B \) is a subset of \( Y \), then the inverse image of \( B \) is the set \( \{x: xRy \text{ for some } y \in B\} \). 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 \( xTz \) if and only if there exists an element \( y \) in \( Y \) such that \( xRy \) and \( ySz \). The restriction of \( R \) to \( A \) is the set \( S \) defined so that \( xSy \) if and only if there exists an element \( x \) in \( A \) such that \( xRy \). If \( S \) is a restriction of \( R \) to \( A \), then \( R \) is an extension of \( S \) to \( X \). If \( X =Y \) then we call \( R \) a relation in \( X \). The field of a relation \( R \) in \( X \) is the union of...
the domain of \( R \) and the range of \( R \). If \( R \) is a relation in \( X \), and if \( S \) is defined so that \( xSy \) if \( xRy \) and not \( x=y \), then \( S \) is the strict relation corresponding to \( R \), and vice versa, if \( S \) is a relation in \( X \), and if \( R \) is defined so that \( xSy \) if \( xRy \) 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 \( X \times Y \), 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 \( F_1 \) and \( F_2 \) is called the composite of \( F_1 \) and \( F_2 \) if the range of \( F_1 \) is a subset of the domain of \( F_2 \).

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 \((n\text{-ary})\) relation, and say that the relation is a subset of the Cartesian product \( X[1]...X[n] \) where \( x[i] \) is an element of \( X[i] \), \( 1\leq i\leq n \). The projection of an \( n\text{-ary} \) relation \( R \) onto coordinate \( i \) is the set \( \{x[i] \mid (x[1],...,x[i],...,x[n]) \in R \text{ for some } x[j] \in X[j], 1\leq j\leq n \text{ and not } i=j \} \). 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 \( n\text{-ary} \) relations as follows. Let \( TR \) be an ordered set \((R[1],...,R[n])\) of binary relations from \( X[1] \) to \( Y[1] \) 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 \( X[1] \) to \( Z \) defined so that \( xTz \) if and only if there exists an element \( y[i] \) in \( Y[i] \) for each \( 1\leq i\leq n \) 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 \( X[1] \) to \( Y[1] \) and \( S \) a subset of \( X[1]...X[n] \). The multiple relative product of \( TR \) and \( S \) is defined to be the set \( z \) of \((x[1],...,x[n]), (y[1],...,y[n])) \) for some \((x[1],...,x[n])\) in \( S \) and for some \((x[i],y[i]) \in R[i], 1\leq i\leq n \). The natural join of an \( n\text{-ary} \) relation \( R \) and an \( m\text{-ary} \) 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 \( \text{"\text{"} \} \) and for every term X, (T,X) belongs to Sets (atomic sets);
- \( \{\text{"\text{"} \},[]} \) 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 \( 1 \leq i \leq n \) 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 \( 1 \leq i \leq n \), 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 320] 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 327], checks that the arguments represent binary relations by matching \([A,B]\) against the type of the first argument (Arg1 say), and \([C,D]\) against the type of the second argument (Arg2 say). The fact that \([A,B]\) matches the type of Arg1 is to be interpreted as Arg1 representing a binary relation from X to Y, where X is defined as all sets Set(x) for some element x in Sets the type of which is A, and similarly for Y. In the same way Arg2 is interpreted as representing a binary relation from W to Z. Finally it is checked that B matches C, which is sufficient to ensure that W is equal to Y. The untyped empty set is handled separately: its type, \( \text{"\text{"} \} \), 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 \([\text{external}, \text{Fun}]\), 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 \([\text{external}, \text{fun}(X) \rightarrow \text{element}(I, X)]\), but is to be preferred since it makes it possible to handle this case even more efficiently. Examples of SetFuns:

\[
\{\text{sofs, union} \}
\text{fun}(S) \rightarrow \text{sofs:partition}(1, S) \text{ end}
\{\text{external, fun(A) \rightarrow A} \text{ end}\}
\{\text{external, fun([A,..,C]) \rightarrow \{C, A} \text{ end}\}
\{\text{external, fun([..,[],C])} \rightarrow C \text{ end}\}
\{\text{external, fun([..,[],{E],C})} \rightarrow \{E,\{E, C\}} \text{ end}\}
\]

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: \text{from_external}, \text{is_empty_set}, \text{is_set}, \text{is_sofs_set}, \text{to_external}, \text{type}.

The functions of this module exit the process with a \text{badarg}, \text{bad_function}, or \text{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
    | \{\text{external, fun(external_set())} \rightarrow \text{external_set()}}
    | \text{fun(anyset())} \rightarrow \text{anyset()}
spec_fun() = \{\text{external, fun(external_set())} \rightarrow \text{bool()}\}
    | \text{fun(anyset())} \rightarrow \text{bool()}
type() = -a type-
Exports

\texttt{a_function(Tuples [, Type]) -> Function}

Types:
- Function = function()
- Tuples = [tuple()]
- Type = type()

Creates a function [page 310]. \texttt{a_function(F,T)} is equivalent to \texttt{from_term(F,T)}, if the result is a function. If no type [page 311] is explicitly given, \texttt{[atom, atom]} is used as type of the function.

\texttt{canonical_relation(SetOfSets) -> BinRel}

Types:
- BinRel = binary_relation()
- SetOfSets = set_of_sets()

Returns the binary relation containing the elements \((E, Set)\) such that \(Set\) belongs to \(SetOfSets\) and \(E\) belongs to \(Set\). If \(SetOfSets\) is a partition [page 310] of a set \(X\) and \(R\) is the equivalence relation in \(X\) induced by \(SetOfSets\), then the returned relation is the canonical map [page 310] 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) -> Function3}

Types:
- Function1 = Function2 = Function3 = function()

Returns the composite [page 310] 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) -> Function}

Types:
- AnySet = anyset()
- Function = function()
- Set = set()

Creates the function [page 310] that maps each element of the set \(Set\) onto \(AnySet\).
sofs:STDLIB Reference Manual

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()
Returs the converse [page 309] 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 309] 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 310] 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 311] 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 309] 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 309] of BinRel1 to Set.

```erlang
1> R1 = sofs:relation([[1,a],[2,b],[3,c]]),
   S = sofs:set([2,4,6]),
   R2 = sofs:drestriction(R1, S),
   sofs:to_external(R2).
[[1,a],[3,c]]
drestriction(R,S) is equivalent to difference(R,restriction(R,S)).
```

drestriction(SetFun, Set1, Set2) -> Set3

Types:
- SetFun = set_fun()
- Set1 = Set2 = Set3 = set()

Returns a subset of Set1 containing those elements that do not yield an element in Set2 as the result of applying SetFun.

```erlang
1> SetFun = {external, fun({A,B,C}) -> {B,C} end},
   R1 = sofs:relation([[a,aa,1],[b,bb,2],[c,cc,3]]),
   R2 = sofs:relation([[bb,2],[cc,3],[dd,4]]),
   R3 = sofs:drestriction(SetFun, R1, R2),
   sofs:to_external(R3).
[[a,aa,1]]
drestriction(F,S1,S2) is equivalent to difference(S1,restriction(F,S1,S2)).
```

eempty_set() -> Set

Types:
- Set = set()

Returns the untyped empty set [page 311]. empty_set() is equivalent to from_term([], []).

eextension(BinRel1, Set, AnySet) -> BinRel2

Types:
- AnySet = anyset()
- BinRel1 = BinRel2 = binary_relation()
- Set = set()

Returns the extension [page 309] of BinRel1 such that for each element E in Set that does not belong to the domain [page 309] of BinRel1, BinRel2 contains the pair (E, AnySet).

```erlang
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 310]. family(F,T) is equivalent to from_term(F,T), if the result is a family. If no type [page 311] 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 310], 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.

```erlang
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 310] 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 309] of Family1[i].

```erlang
1> FR = sofs:from_term([[a,[]],[2,b],[3,c]],[],[b,[]],[c,[4,5,6,7]]),
   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 310] 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 309] of Family1[i].

```erlang
1> FR = sofs:from_term([[a,[]],[2,b],[3,c]],[],[b,[]],[c,[4,5,6,7]]),
   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 310] 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 309] of Family1[i].

If Family1[i] is an empty set for some i, then the process exits with a badarg message.

1> F1 = sofs:from_term([[a,[[1,2,3],[2,3,4]]],[b,[[x,y,z],[x,y]]]]),
   F2 = sofs:family_intersection(F1),
   sofs:to_external(F2).
   [{a,[2,3]},{b,[x,y]]

family_intersection(Family1, Family2) -> Family3

Types:
- Family1 = Family2 = Family3 = family()

If Family1 and Family2 are families [page 310], then Family3 is the family such that the index set is the intersection of Family1's and Family2's index sets, and Family3[i] is the intersection of Family1[i] and Family2[i].

1> F1 = sofs:family([[a,[1,2]],[b,[]],[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 310] then Family2 is the family with the same index set as Family1 such that Family2[i] is the result of calling SetFun with Family1[i] as argument.

1> F1 = sofs:from_term([[a,[[1,2],[2,3]]],[b,[]]]),
   F2 = sofs:family_projection([sofs, union], F1),
   sofs:to_external(F2).
   [[a,[1,2,3]],[b,[[]]]

family_range(Family1) -> Family2

Types:
- Family1 = Family2 = family()

If Family1 is a family [page 310] 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 309] of Family1[i].
family_specification(Fun, Family1) -> Family2
Types:
• Fun = spec_fun()
• Family1 = Family2 = family()
If Family1 is a family [page 310], then Family2 is the restriction [page 309] 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 311] 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 310] Family. For each pair (a,\{b[1],...,b[n]\}) of Family, the vertex a as well the edges (a,b[i]) for 1 <= i <= n are added to a newly created directed graph.
If no graph type is given, digraph:new/1 is used for creating the directed graph, otherwise the GraphType argument is passed on as second argument to digraph:new/2.
If 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 310], 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].

family_union(Family1) -> Family2

Types:
  * Family1 = Family2 = family()
If Family1 is a family [page 310] 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 309] of Family1[i].

```
1> F1 = sofs:from_term([[a,[[1,2],[2,3]],{b,[]]]]),
   F2 = sofs:family_union(F1),
   sofs:to_external(F2).
[a,[1,2,3]],[b,[]]
family_union(F) is equivalent to family_projection([sofs,union],F).
```

family_union(Family1, Family2) -> Family3

Types:
  * Family1 = Family2 = Family3 = family()
If Family1 and Family2 are families [page 310], then Family3 is the family such that the index set is the union of Family1’s and Family2’s index sets, and Family3[i] is the union of Family1[i] and Family2[i] if both maps i, Family1[i] or Family2[i] otherwise.

```
1> F1 = sofs:family([[a,[[1,2]],[b,[[3,4]],{c,[5,6]]]]]),
   F2 = sofs:family([[b,[[4,5]]],[c,[[7,8]],{d,[9,10]]]]),
   F3 = sofs:family_union(F1, F2),
   sofs:to_external(F3).
[a,[1,2]],{b,[3,4,5],[c,5,6,7,8],[d,[9,10]]]
```

field(BinRel) -> Set

Types:
  * BinRel = binary_relation()
  * Set = set()
Returns the field [page 309] of the binary relation BinRel.

```
1> R = sofs:relation([[a],[1,b],[2,b],[2,c]]),
   S = sofs:field(R),
   sofs:to_external(S).
[1,2,a,b,c]
field(R) is equivalent to union(domain(R), range(R)).
```

from_external(ExternalSet, Type) -> AnySet

Types:
  * ExternalSet = external_set()
  * AnySet = anyset()
  * Type = type()
Creates a set from the external set [page 311] ExternalSet and the type [page 311] Type. It is assumed that Type is a valid type [page 311] of ExternalSet.

```
from_sets(ListOfSets) -> Set
```
Types:
- Set = set()
-ListOfSets = [anyset()]

Returns the unordered set [page 311] containing the sets of the list ListOfSets.

```prolog
1> S1 = sofs:relation([[a,1],[b,2]]),
   S2 = sofs:relation([[x,3],[y,4]]),
   S = sofs:from_sets([S1,S2]),
   sofs:to_external(S).
[['[a,1],[b,2]],[[x,3],[y,4]]]
```

`from_sets(TupleOfSets) -> Ordset`

Types:
- Ordset = ordset()
- TupleOfSets = tuple-of(anyset())

Returns the ordered set [page 311] containing the sets of the non-empty tuple TupleOfSets.

`from_term(Term [, Type]) -> AnySet`

Types:
- AnySet = anyset()
- Term = term()
- Type = type()

Creates an element of Sets [page 311] by traversing the term Term, sorting lists, removing duplicates and deriving or verifying a valid type [page 311] for the so obtained external set. An explicitly given type [page 311] 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:

```prolog
1> S = sofs:from_term([["foo"],[1,1]],["foo",[2,2]],[[atom,[atom]]]),
   sofs:to_external(S).
[{{"foo"},[1]},{"foo",[2]}]
```

`from_term` can be used for creating atomic or ordered sets. The only purpose of such a set is that of later building unordered sets since all functions in this module that do anything operate on unordered sets. Creating unordered sets from a collection of ordered sets may be the way to go if the ordered sets are big and one does not want to waste heap by rebuilding the elements of the unordered set. An example showing that a set can be built “layer by layer”:

```prolog
1> A = sofs:from_term(a),
   S = sofs:set([1,2,3]),
   P1 = sofs:from_sets([A,S]),
   P2 = sofs:from_term([b,[6,5,4]]),
   Ss = sofs:from_sets([P1,P2]),
   sofs:to_external(Ss).
[['[a,1,2,3]],[b,[4,5,6]]]
```
Other functions that create sets are `from_external/2` and `from_sets/1`. Special cases of `from_term/2` are `a_function/1,2`, `empty_set/0`, `family/1,2`, `relation/1,2`, and `set/1,2`.

\[ \text{image}(\text{BinRel}, \text{Set1}) \rightarrow \text{Set2} \]

Types:
- \( \text{BinRel} = \text{binary} \_	ext{relation}() \)
- \( \text{Set1} = \text{Set2} = \text{set}() \)

Returns the image [page 309] of the set \( \text{Set1} \) under the binary relation \( \text{BinRel} \).

1> \( \text{R} = \text{sofs:relation}([\{1,a\},\{2,b\},\{2,c\},\{3,d\}]), \)
   \( \text{S1} = \text{sofs:set}([1,2]), \)
   \( \text{S2} = \text{sofs:image}(\text{R}, \text{S1}), \)
   \( \text{sofs:to_external}(\text{S2}). \)

\[ \text{intersection}(\text{SetOfSets}) \rightarrow \text{Set} \]

Types:
- \( \text{Set} = \text{set}() \)
- \( \text{SetOfSets} = \text{set of sets}() \)

Returns the intersection [page 309] of the set of sets \( \text{SetOfSets} \).

Intersecting an empty set of sets exits the process with a `badarg` message.

\[ \text{intersection}(\text{Set1}, \text{Set2}) \rightarrow \text{Set3} \]

Types:
- \( \text{Set1} = \text{Set2} = \text{Set3} = \text{set}() \)

Returns the intersection [page 309] of \( \text{Set1} \) and \( \text{Set2} \).

\[ \text{intersection_of_family}(\text{Family}) \rightarrow \text{Set} \]

Types:
- \( \text{Family} = \text{family}() \)
- \( \text{Set} = \text{set}() \)

Returns the intersection of the family [page 310] \( \text{Family} \).

Intersecting an empty family exits the process with a `badarg` message.

1> \( \text{F} = \text{sofs:family}([\{a,\{0,2,4\}\},\{b,\{0,1,2\}\},\{c,\{2,3\}\}]), \)
   \( \text{S} = \text{sofs:intersection_of_family}(\text{F}), \)
   \( \text{sofs:to_external}(\text{S}). \)

\[ \text{inverse}(\text{Function1}) \rightarrow \text{Function2} \]

Types:
- \( \text{Function1} = \text{Function2} = \text{function}() \)

Returns the inverse [page 310] of the function \( \text{Function1} \).
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 309] 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 310] 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 309], 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 309], false otherwise.

is_set(AnySet) -> Bool
Types:
• AnySet = anyset()
  • Bool = bool()
Returns true if AnySet is an unordered set [page 311], 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 311], 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 309] of Set2, false otherwise.

is_type(Term) -> Bool
Types:
  • Bool = bool()
  • Term = term()
Returns true if the term Term is a type [page 311].

join(Relation1, I, Relation2, J) -> Relation3
Types:
  • Relation1 = Relation2 = Relation3 = relation()
  • I = J = integer() > 0
Returns the natural join [page 310] of the relations Relation1 and Relation2 on
coordinates I and J.

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 310] of the
ordered set \{R[i],...,R[n]\} and BinRel1.
Ri = sofs:relation([[a,1],[b,2],[c,3]]),
R = sofs:relation([[a,b],[b,c],[c,a]]),
MP = sofs:multiple_relative_product([Ri, Ri], R),
sofs:to_external(sofs:range(MP)).

[1,2],[2,3],[3,1]]

no_elements(ASet) -> NoElements
Types:
• A Set = set() | ordset()
• NoElements = integer() >= 0
Returns the number of elements of the ordered or unordered set A Set.

partition(SetOfSets) -> Partition
Types:
• SetOfSets = set_of_sets()
• Partition = set()
Returns the partition [page 310] 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 310] 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 310] 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.

partition(SetFun1, Set1, Set2) -> {Set3, Set4}
Types:
• SetFun1 = set_fun()
• Set1 = Set2 = Set3 = Set4 = set()
Returns a pair of sets that, regarded as constituting a set, forms a partition [page 310] of
Set1. If the result of applying SetFun1 to an element of Set2 yields an element in Set1,
the element belongs to Set3, otherwise the element belongs to Set4.
1> R1 = sofs:relation([[1,a],[2,b],[3,c]]),
S = sofs:set([2,4,6]),
[R2,R3] = sofs:partition(1, R1, S),
{sofs:to_external(R2),sofs:to_external(R3)}.
[[2,b]],[[1,a],[3,c]]
partition(F,S1,S2) is equivalent to {restriction(F,S1,S2),
drestriction(F,S1,S2)}.

partition_family(SetFun, Set) -> Family
Types:
• Family = family()
• SetFun = set_fun()
• Set = set()

Returns the family [page 310] Family where the indexed set is a partition [page 310] of
Set such that two elements are considered equal if the results of applying SetFun are the
same value i. This i is the index that Family maps onto the equivalence class [page 310].

1> S = sofs:relation([[a,a,a,a],[a,a,b,b],[a,b,b,b]]),
SetFun = {external, fun(A,C) -> [A,C] end},
F = sofs:partition_family(SetFun, S),
sofs:to_external(F).
[[[a,a],[a,a,a,a]],[[a,b],[[a,a,b,b],[a,b,b,b]]]]

product(TupleOfSets) -> Relation
Types:
• Relation = relation()
• TupleOfSets = tuple_of(set())

If (x[1],...,x[n]) is an element of the n-ary relation Relation, then x[i] is drawn from
element i of TupleOfSets

1> S1 = sofs:set([a,b]),
S2 = sofs:set([1,2]),
S3 = sofs:set([x,y]),
P3 = sofs:product([S1,S2,S3]),
sofs:to_external(P3).
[[[a,1],[a,1,y],[a,2,x],[a,2,y],[b,1,x],[b,1,y],[b,2,x],[b,2,y]]]

product(Set1, Set2) -> BinRel
Types:
• BinRel = binary_relation()
• Set1 = Set2 = set()

Returns the Cartesian product [page 309] of Set1 and Set2.

1> S1 = sofs:set([1,2]),
S2 = sofs:set([a,b]),
R = sofs:product(S1, S2),
sofs:to_external(R).
[[1,a],[1,b],[2,a],[2,b]]
product(S1, S2) is equivalent to product([S1, S2]).

projection(SetFun, Set1) -> Set2

Types:
- SetFun = set fun()
- Set1 = Set2 = set()

Returns the set created by substituting each element of Set1 by the result of applying SetFun to the element.

If SetFun is a number i ≥ 1 and Set1 is a relation, then the returned set is the projection [page 310] of Set1 onto coordinate i.

1> S1 = sofs:from_term([[1,a],[2,b],[3,a]]),
   S2 = sofs:projection(2, S1),
   sofs:to_external(S2).
   [a,b]

range(BinRel) -> Set

Types:
- BinRel = binary relation()
- Set = set()

Returns the range [page 309] of the binary relation BinRel.

1> R = sofs:relation([[1,a],[1,b],[2,b],[2,c]]),
   S = sofs:range(R),
   sofs:to_external(S).
   [a,b,c]

relation(Tuples [, Type]) -> Relation

Types:
- N = integer()
- Type = N | type()
- Relation = relation()
- Tuples = [tuple()]

Creates a relation [page 309]. relation(R, T) is equivalent to from_term(R, T), if T is a type [page 311] and the result is a relation. If Type is an integer N, then [[atom,...,atom]], where the size of the tuple is N, is used as type of the relation. If no type is explicitly given, the size of the first tuple of Tuples is used if there is such a tuple. relation([]) is equivalent to relation([],2).

relation_to_family(BinRel) -> Family

Types:
- Family = family()
- BinRel = binary relation()

Returns the family [page 310] Family such that the index set is equal to the domain [page 309] of the binary relation BinRel, and Family[i] is the image [page 309] of the set of i under BinRel.
1 > R = sofs:relation([[b,1],[c,2],[c,3]]),
   F = sofs:relation_to_family(R),
   sofs:to_external(F).
   [[b,[1]],[c,[2,3]]]

relative_product(TupleOfBinRels [, BinRel1]) -> BinRel2

Types:
- TupleOfBinRels = tuple-of(BinRel)
- BinRel = BinRel1 = BinRel2 = binary_relation()

If TupleOfBinRels is a non-empty tuple [R[1],...,R[n]] of binary relations and BinRel1
is a binary relation, then BinRel2 is the relative product [page 310] of the ordered set
(R[1],...,R[n]) and BinRel1.

If BinRel1 is omitted, the relation of equality between the elements of the Cartesian
product [page 310] of the ranges of R[1], 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 309] of the binary relations BinRel1 and BinRel2.

relative_product1(BinRel1, BinRel2) -> BinRel3

Types:
- BinRel1 = BinRel2 = BinRel3 = binary_relation()

Returns the relative product [page 309] of the converse [page 309] 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 309] of the binary relation BinRel1 to Set.
sofs STDLIB Reference Manual

```prolog
1> R1 = sofs:relation([[1,a],[2,b],[3,c]]),
   S = sofs:set([1,2,4]),
   R2 = sofs:restriction(R1, S),
   sofs:to_external(R2).
   [{1,a},{2,b}]
```

restriction(SetFun, Set1, Set2) -> Set3

Types:
- SetFun = set_fun()
- Set1 = Set2 = Set3 = set()

Returns a subset of Set1 containing those elements that yield an element in Set2 as the result of applying SetFun.

```prolog
1> S1 = sofs:relation([[1,a],[2,b],[3,c]]),
   S2 = sofs:set([b,c,d]),
   S3 = sofs:restriction(2, S1, S2),
   sofs:to_external(S3).
   [{2,b},{3,c}]
```

set(Terms [, Type]) -> Set

Types:
- Set = set()
- Terms = [term()]
- Type = type()

Creates an unordered set [page 311]. set(L,T) is equivalent to from_term(L,T), if the result is an unordered set. If no type [page 311] is explicitly given, [atom] is used as type of the set.

```prolog
1> R1 = sofs:relation([[a,1],[b,2]]),
   R2 = sofs:relation([[x,1],[x,2],[y,3]]),
   S1 = sofs:from_sets([R1,R2]),
   S2 = sofs:specification([sofs,is_a_function], S1),
   sofs:to_external(S2).
   [[[a,1],[b,2]]]
```

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 311] of each element, otherwise Fun is applied to each element.

```prolog
1> R1 = sofs:relation([[a,1],[b,2]]),
   R2 = sofs:relation([[x,1],[x,2],[y,3]]),
   S1 = sofs:from_sets([R1,R2]),
   S2 = sofs:specification([sofs,is_a_function], S1),
   sofs:to_external(S2).
   [[[a,1],[b,2]]]
```

strict_relation(BinRel1) -> BinRel2

Types:
- BinRel1 = BinRel2 = binary_relation()

Returns the strict relation [page 310] corresponding to the binary relation BinRel1.
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.

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 309] 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 309] (or the Boolean sum) of Set1 and Set2.
S1 = sofs:set([1, 2, 3]),
S2 = sofs:set([2, 3, 4]),
P = sofs:symdiff(S1, S2),
sofs:to_external(P).

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 311] 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 311] of an atomic, ordered or unordered set.

union(SetOfSets) -> Set

Types:
  • Set = set()
  • SetOfSets = set_of_sets()
Returns the union [page 309] of the set of sets SetOfSets.

union(Set1, Set2) -> Set3

Types:
  • Set1 = Set2 = Set3 = set()
Returns the union [page 309] of Set1 and Set2.
union_of_family(Family) -> Set

Types:
  • Family = family()
  • Set = set()

Returns the union of the family [page 310] Family.

```erlang
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 310] W corresponding to the binary relation BinRel1. Let F be the field [page 309] 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.

```erlang
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 87], digraph(3) [page 92], orddict(3) [page 249], ordsets(3) [page 250], sets(3) [page 293]
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:substr(" Hello Hello World World ", "Hello World").
3

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 defgghix 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()
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 Character.

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 Character.

For example:

> string:right("Hello", 10, $).
".....Hello"

centre(String, Number) -> Centered
centre(String, Number, Character) -> Centered

Types:
- String = Centered = string()
- Character = char
- Number = integer()

Returns a string, where String is centred in the string and surrounded by blanks or characters. The resulting string will have the length Number.

sub_string(String, Start) -> SubString
sub_string(String, Start, Stop) -> SubString

Types:
- String = SubString = string()
- Start = Stop = integer()

Returns a substring of String, starting at the position Start to the end of the string, or to 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:

```erlang
> {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:
\[\{I_1, I_2\} = \text{string:to_integer}("33+22"),\]
\[\{I_2, []\} = \text{string:to_integer}(I_1),\]
\[I_1 - I_2.\]
\[11\]
\[\text{string:to_integer}("0.5").\]
\[\{0, \".5\"\}\]
\[\text{string:to_integer}("x=2").\]
\[\{\text{error}, \text{no_integer}\}\]

```
to_lower(String) -> Result
    to_lower(Char) -> CharResult
    to_upper(String) -> Result
    to_upper(Char) -> CharResult
```

Types:
- String = Result = string()
- Char = CharResult = integer()

The given string or character is case-converted. Note that the supported character set is ISO/IEC 8859-1 (a.k.a. Latin 1), all values outside this set is unchanged.

Notes

Some of the general string functions may seem to overlap each other. The reason for this is that this string package is the combination of two earlier packages and all the functions of both packages have been retained.

The regular expression functions have been moved to their own module regexp (see regexp(3) [page 288]). 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 MaxR and MaxT. If more than MaxR restarts occur within MaxT seconds, the supervisor terminates all child processes and then itself.

This is the type definition of a child specification:

```haskell
child_spec() = {Id,StartFunc,Restart,Shutdown,Type,Modules}
Id = term()
StartFunc = {M,F,A}
M = F = atom()
A = [term()]
Restart = permanent | transient | temporary
Shutdown = brutal_kill | int()>=0 | infinity
Type = worker | supervisor
Modules = [Module] | dynamic
Module = atom()
```

- **Id** is a name that is used to identify the child specification internally by the supervisor.
- **StartFunc** defines the function call used to start the child process. It should be a module-function-arguments tuple \( \{M,F,A\} \) used as \( \text{apply}(M,F,A) \).

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 \{ignore\} if the child process for some reason cannot be started, in which case the child specification will be kept by the supervisor but the non-existing child process will be ignored.

If something goes wrong, the function may also return an error tuple \{error,Error\}.

Note that the start_link functions of the different behaviour modules fulfill the above requirements.

- **Restart** defines when a terminated child process should be restarted. A permanent child process should always be restarted, a temporary child process should never be restarted and a transient child process should be restarted only if it terminates abnormally, i.e. with another exit reason than normal.
- **Shutdown** defines how a child process should be terminated. brutal_kill means the child process will be unconditionally terminated using \( \text{exit}(\text{Child},\text{kill}) \). 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, **Shutdown** should be set to infinity to give the subtree ample time to shutdown.

Important note on simple-one-for-one supervisors: The dynamically created child processes of a simple-one-for-one supervisor are not explicitly killed, regardless of
shutdown strategy, but are expected to terminate when the supervisor does (that is, when an exit signal from the parent process is received). Note that all child processes implemented using the standard OTP behavior modules automatically adhere to the shutdown protocol.

- **Type** specifies if the child process is a supervisor or a worker.
- **Modules** is used by the release handler during code replacement to determine which processes are using a certain module. As a rule of thumb, **Modules** should be a list with one element `[Module]`, where **Module** is the callback module, if the child process is a supervisor, `gen_server` or `gen_fsm`. If the child process is an event manager (`gen_event`) with a dynamic set of callback modules, **Modules** should be dynamic. See OTP Design Principles for more information about release handling.
- Internally, the supervisor also keeps track of the pid **Child** of the child process, or `undefined` if no pid exists.

### Exports

```
start_link(Module, Args) -> Result
start_link(SupName, Module, Args) -> Result
```

Types:
- **SupName** = `{local, Name} | {global, Name}`
- **Name** = `atom()`
- **Module** = `atom()`
- **Args** = `term()`
- **Result** = `{ok, Pid} | ignore | {error, Error}`
- **Pid** = `pid()`
- **Error** = `{already_started, Pid} | shutdown | term()`

Creates a supervisor process as part of a supervision tree. The function will, among other things, ensure that the supervisor is linked to the calling process (its supervisor). The created supervisor process calls `Module:init/1` to find out about restart strategy, maximum restart frequency and child processes. To ensure a synchronized start-up procedure, `start_link/2,3` does not return until `Module:init/1` has returned and all child processes have been started.

If **SupName**=`{local, Name}` the supervisor is registered locally as **Name** using `register/2`. If **SupName**=`{global, Name}` the supervisor is registered globally as **Name** using `global:register_name/2`. If no name is provided, the supervisor is not registered.

**Module** is the name of the callback module.

**Args** is an arbitrary term which is passed as the argument to `Module:init/1`.

If the supervisor and its child processes are successfully created (i.e., if all child process start functions return `{ok, Child}, {ok, Child, Info}, or `ignore`) the function returns `{ok, Pid}`, where **Pid** is the pid of the supervisor. If there already exists a process with the specified **SupName** the function returns `{error, {already_started, Pid}}`, where **Pid** is the pid of that process.

If `Module:init/1` returns `ignore`, this function returns `ignore` as well and the supervisor terminates with reason `normal`. If `Module:init/1` fails or returns an
incorrect value, this function returns \{error,Term\} where Term is a term with information about the error, and the supervisor terminates with reason Term.

If any child process start function fails or returns an error tuple or an erroneous value, the function returns \{error,shutdown\} and the supervisor terminates all started child processes and then itself with reason shutdown.

\texttt{start\_child(SupRef, ChildSpec) -> Result}  

Types:
- \texttt{SupRef = Name \mid \{Name,Node\} \mid \{global,Name\} \mid pid()}  
- \texttt{Name = Node = atom()}  
- \texttt{ChildSpec = child\_spec() \mid [term()]}  
- \texttt{Result = \{ok,Child\} \mid \{ok,Child,Info\} \mid \{error,Error\}}  
- \texttt{Child = pid() \mid undefined}  
- \texttt{Info = term()}  
- \texttt{Error = already\_present \mid \{already\_started,Child\} \mid term()}  

Dynamically adds a child specification to the supervisor \texttt{SupRef} which starts the corresponding child process.

\texttt{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.

\texttt{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:\texttt{init/1} will be used and \texttt{ChildSpec} should instead be an arbitrary list of terms \texttt{List}. The child process will then be started by appending \texttt{List} to the existing start function arguments, i.e. by calling \texttt{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 \texttt{Id}, \texttt{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.

\texttt{terminate\_child(SupRef, Id) -> Result}  

Types:
- \texttt{SupRef = Name \mid \{Name,Node\} \mid \{global,Name\} \mid pid()}
Tells the supervisor \texttt{SupRef} to terminate the child process corresponding to the child specification identified by \texttt{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 \texttt{restart_child/2}. Use \texttt{delete_child/2} to remove the child specification.

See \texttt{start_child/2} for a description of \texttt{SupRef}.

If successful, the function returns \texttt{ok}. If there is no child specification with the specified \texttt{Id}, the function returns \{\texttt{error,not_found}\}.

\begin{verbatim}
delete_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
\end{verbatim}

Tells the supervisor \texttt{SupRef} to delete the child specification identified by \texttt{Id}. The corresponding child process must not be running, use \texttt{terminate_child/2} to terminate it.

See \texttt{start_child/2} for a description of \texttt{SupRef}.

If successful, the function returns \texttt{ok}. If the child specification identified by \texttt{Id} exists but the corresponding child process is running, the function returns \{\texttt{error,running}\}. If the child specification identified by \texttt{Id} does not exist, the function returns \{\texttt{error,not_found}\}.

\begin{verbatim}
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()
\end{verbatim}

Tells the supervisor \texttt{SupRef} to restart a child process corresponding to the child specification identified by \texttt{Id}. The child specification must exist and the corresponding child process must not be running.

See \texttt{start_child/2} for a description of \texttt{SupRef}.

If the child specification identified by \texttt{Id} does not exist, the function returns \{\texttt{error,not_found}\}. If the child specification exists but the corresponding process is already running, the function returns \{\texttt{error,running}\}. If the child process start function returns \{\texttt{ok,Child}\} or \{\texttt{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 `/CU ok,undefined/CV`.

If the child process start function returns an error tuple or an erroneous value, or if it fails, the function returns `/CU error,Error/CV` where `Error` is a term containing information about the error.

`which_children(SupRef) -> [{Id,Child,Type,Modules}]`

Types:
- `SupRef = Name | {Name,Node} | {global,Name} | pid()`
- `Name = Node = atom()`
- `Id = term() | undefined`
- `Child = pid() | undefined`
- `Type = worker | supervisor`
- `Modules = [Module] | dynamic`
- `Module = atom()`

Returns a list with information about all child specifications and child processes belonging to the supervisor `SupRef`.

See `start_child/2` for a description of `SupRef`.

The information given for each child specification/process is:

- `Id` - as defined in the child specification or `undefined` in the case of a `simple_one_for_one` supervisor.
- `Child` - the pid of the corresponding child process, or `undefined` if there is no such process.
- `Type` - as defined in the child specification.
- `Modules` - as defined in the child specification.

`check_childspecs([ChildSpec]) -> Result`

Types:
- `ChildSpec = child_spec()`
- `Result = ok | {error,Error}`
- `Error = term()`

This function takes a list of child specification as argument and returns `ok` if all of them are syntactically correct, or `{error,Error}` otherwise.

**CALLBACK FUNCTIONS**

The following functions should be exported from a `supervisor` callback module.
Exports

Module:init(Args) -> Result

Types:
- Args = term()
- Result = {ok,{{RestartStrategy,MaxR,MaxT},[ChildSpec]}} | ignore
- RestartStrategy = one_for_all | one_for_one | rest_for_one | simple_one_for_one
- MaxR = MaxT = int() = 0
- ChildSpec = child_spec()

Whenever a supervisor is started using supervisor:start_link/2,3, this function is called by the new process to find out about restart strategy, maximum restart frequency and child specifications.

Args is the Args argument provided to the start function.

RestartStrategy is the restart strategy and MaxR and MaxT defines the maximum restart frequency of the supervisor. [ChildSpec] is a list of valid child specifications defining which child processes the supervisor should start and monitor. See the discussion about Supervision Principles above.

Note that when the restart strategy is simple_one_for_one, the list of child specifications must be a list with one child specification only. (The Id is ignored). No child process is then started during the initialization phase, but all children are assumed to be started dynamically using supervisor:start_child/2.

The function may also return ignore.

SEE ALSO

gen_event(3) [page 173], gen_fsm(3) [page 183], gen_server(3) [page 194], sys(3) [page 348]
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

\[ \text{start_link(Module, Args) -> Result} \]
\[ \text{start_link(SupBridgeName, Module, Args) -> Result} \]

Types:
- \( \text{SupBridgeName} = \{\text{local,Name}\} \mid \{\text{global,Name}\} \)
- \( \text{Name} = \text{atom}() \)
- \( \text{Module} = \text{atom}() \)
- \( \text{Args} = \text{term}() \)
- \( \text{Result} = \{\text{ok,Pid}\} \mid \text{ignore} \mid \{\text{error,Error}\} \)
- \( \text{Pid} = \text{pid}() \)
- \( \text{Error} = \{\text{already_started,Pid}\} \mid \text{term}() \)

Creates a supervisor_bridge process, linked to the calling process, which calls Module:init/1 to start the subsystem. To ensure a synchronized start-up procedure, this function does not return until Module:init/1 has returned.

If SupBridgeName=\{local,Name\} the supervisor_bridge is registered locally as Name using register/2. If SupBridgeName=\{global,Name\} the supervisor_bridge is registered globally as Name using global:register_name/2. If no name is provided, the supervisor_bridge is not registered. If there already exists a process with the specified SupBridgeName the function returns \{\text{error,\{already_started,Pid\}}\}, where Pid is the pid of that process.

Module is the name of the callback module.

Args is an arbitrary term which is passed as the argument to Module:init/1.

If the supervisor_bridge and the subsystem are successfully started the function returns \{\text{ok,Pid}\}, where 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

**Module:init(Args) -> Result**

Types:
- Args = term()
- Result = `ok`, Pid, State | `ignore` | `{error,Error}`
- Pid = pid()
- State = term()
- Error = term()

Whenever a supervisor_bridge is started using `supervisor_bridge:start_link/2,3`, this function is called by the new process to start the subsystem and initialize. `Args` is the `Args` argument provided to the start function.

The function should return `{ok, Pid, State}` where `Pid` is the pid of the main process in the subsystem and `State` is any term.

If later `Pid` terminates with a reason `Reason`, the supervisor_bridge will terminate with reason `Reason` as well. If later the supervisor_bridge is stopped by its supervisor with reason `Reason`, it will call `Module:terminate(Reason, State)` to terminate.

If something goes wrong during the initialization the function should return `{error, Error}` where `Error` is any term, or `ignore`.

**Module:terminate(Reason, State)**

Types:
- Reason = shutdown | term()
- State = term()

This function is called by the supervisor_bridge when it is about to terminate. It should be the opposite of `Module:init/1` and stop the subsystem and do any necessary cleaning up. The return value is ignored.

`Reason` is `shutdown` if the supervisor_bridge is terminated by its supervisor. If the supervisor_bridge terminates because a a linked process (apart from the main process of the subsystem) has terminated with reason `Term`, `Reason` will be `Term`.

`State` is taken from the return value of `Module:init/1`. 
SEE ALSO

supervisor(3) [page 338], sys(3) [page 348]
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 a shut-down request from its parent, the supervisor. The message `{EXIT, Parent, Reason}` from the parent is an order to terminate. The process must terminate when this message is received, normally with the same Reason as Parent.
There is one more message which the process must understand if the modules used to implement the process change dynamically during runtime. An example of such a process is the gen_event processes. This message is \{get_modules, From\}. The reply to this message is From \{modules, Modules\}, where Modules is a list of the currently active modules in the process. This message is used by the release handler to find which processes execute a certain module. The process may at a later time be suspended and ordered to perform a code change for one of its modules.

System Events

When debugging a process with the functions of this module, the process generates system_events which are then treated in the debug function. For example, trace formats the system events to the tty.

There are three predefined system events which are used when a process receives or sends a message. The process can also define its own system events. It is always up to the process itself to format these events.

Exports

log(Name, Flag)
log(Name, Flag, Timeout) -> ok | \{ok, \{system_event()\}\}

Types:
- Flag = true | \{true, N\} | false | get | print
- N = integer() > 0

Turns the logging of system events On or Off. If On, a maximum of N events are kept in the debug structure (the default is 10). If Flag is get, a list of all logged events is returned. If Flag is print, the logged events are printed to standard io. The events are formatted with a function that is defined by the process that generated the event (with a call to sys:handle_debug/4).

log_to_file(Name, Flag)
log_to_file(Name, Flag, Timeout) -> ok | \{error, open_file\}

Types:
- Flag = FileName | false
- FileName = string()

Enables or disables the logging of all system events in textual format to the file. The events are formatted with a function that is defined by the process that generated the event (with a call to sys:handle_debug/4).

statistics(Name, Flag)
statistics(Name, Flag, Timeout) -> ok | \{ok, Statistics\}

Types:
- Flag = true | false | get
• Statistics = [{start_time, {Date1, Time1}}, {current_time, {Date, Time2}},
               {reductions, integer()}, {messages_in, integer()}, {messages_out, integer()}]
• Date1 = Date2 = {Year, Month, Day}
• Time1 = Time2 = {Hour, Min, Sec}

Enables or disables the collection of statistics. If Flag is get, the statistical collection is
returned.

trace(Name, Flag)
trace(Name, Flag, Timeout) -> void()

Types:
• Flag = boolean()

Prints all system events on standard io. The events are formatted with a function that
is defined by the process that generated the event (with a call to sys:handle debug/4).

no_debug(Name)
no_debug(Name, Timeout) -> void()

Turns off all debugging for the process. This includes functions that have been installed
explicitly with the install function, for example triggers.

suspend(Name)
suspend(Name, Timeout) -> void()

Suspends the process. When the process is suspended, it will only respond to other
system messages, but not other messages.

resume(Name)
resume(Name, Timeout) -> void()

Resumes a suspended process.

change_code(Name, Module, OldVsn, Extra)
change_code(Name, Module, OldVsn, Extra, Timeout) -> ok | {error, Reason}

Types:
• OldVsn = undefined | term()
• Module = atom()
• Extra = term()

Tells the process to change code. The process must be suspended to handle this
message. The Extra argument is reserved for each process to use as its own. The
function Mod:system code change/4 is called. OldVsn is the old version of the Module.

get_status(Name)
get_status(Name, Timeout) -> {status, Pid, {module, Mod}, [PDict, SysState, Parent,
Dbg, Misc]}
install(Name,\{Func,FuncState\})
install(Name,\{Func,FuncState\},Timeout)

Types:
- Func = dbg\_fun()
- dbg\_fun() = fun(FuncState, Event, ProcState) \rightarrow\ done \mid\ 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) \rightarrow\ 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 \texttt{Module:system\textunderscore continue(Parent, NDebug, Misc)} where the process continues the execution, or \texttt{Module:system\textunderscore terminate(Reason, Parent, Debug, Misc)} if the process should terminate. The Module must export \texttt{system\textunderscore continue/3}, \texttt{system\textunderscore terminate/4}, and \texttt{system\textunderscore code\textunderscore 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 \texttt{Module:system\textunderscore continue/3} or \texttt{Module:system\textunderscore terminate/4}.

\begin{verbatim}
print\_log(Debug) -\(\rightarrow\) void()

Types:
\begin{itemize}
  \item \texttt{Debug = \{dbg\ opt\()\}
\end{itemize}
Prints the logged system events in the debug structure using \texttt{FormFunc} as defined when the event was generated by a call to \texttt{handle\_debug/4}.

\texttt{Mod:system\_continue(Parent, Debug, Misc)}

Types:
\begin{itemize}
  \item \texttt{Parent = pid()}\n  \item \texttt{Debug = \{dbg\ opt\()\}
  \item \texttt{Misc = term()}\n\end{itemize}
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{Mod:system\_terminate(Reason, Parent, Debug, Misc)}

Types:
\begin{itemize}
  \item \texttt{Reason = term()}\n  \item \texttt{Parent = pid()}\n  \item \texttt{Debug = \{dbg\ opt\()\}
  \item \texttt{Misc = term()}\n\end{itemize}
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{Mod:system\_code\_change(Misc, Module, OldVsn, Extra)} -\(\rightarrow\) \{ok, NMisc\}

Types:
\begin{itemize}
  \item \texttt{Misc = term()}\n  \item \texttt{OldVsn = undefined | term()}\n  \item \texttt{Module = atom()}\n  \item \texttt{Extra = term()}\n  \item \texttt{NMisc = term()}\n\end{itemize}

\end{verbatim}
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 \texttt{TRef} below. By using \texttt{cancel/1}, the returned reference can be used to cancel any requested action. A \texttt{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

\begin{verbatim}
start() -> ok

Starts the timer server. Normally, the server does not need to be started explicitly. It is started dynamically if it is needed. This is useful during development, but in a target system the server should be started explicitly. Use configuration parameters for \texttt{kernel} for this.

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

Types:
\begin{itemize}
\item Time = integer() in milliseconds
\item Module = Function = atom()
\item Arguments = [term()]
\end{itemize}

Evaluates \texttt{apply(M, F, A)} \texttt{after} \texttt{Time} amount of time has elapsed. Returns \{ok, TRef\}, or \{error, Reason\}.

send_after(Time, Pid, Message) -> \{ok, TRef\} \| \{error,Reason\}

Types:
\begin{itemize}
\item Time = integer() in milliseconds
\item Pid = \texttt{pid()} \| atom()
\item Message = \texttt{term()}
\item Result = \{ok, TRef\} \| \{error, Reason\}
\end{itemize}

send_after/3 Evaluates \texttt{Pid} ! \texttt{Message} \texttt{after} \texttt{Time} amount of time has elapsed. (\texttt{Pid} can also be an atom of a registered name.) Returns \{ok, TRef\}, or \{error, Reason\}.
\end{verbatim}
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}
k ill_after(Time, Pid) -> {ok, TRef} | {error, Reason2}
k ill_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).
k ill_after/2  Same as exit_after(Time, Pid, kill).
k ill_after/1  Same as exit_after(Time, self(), kill).

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

Types:
- Time = integer() in milliseconds
- Module = Function = atom()
- Arguments = [term()]

Evaluates apply(Module, Function, Arguments) repeatedly at intervals of Time. Returns {ok, TRef}, or {error, Reason}.

send_interval(Time, Pid, Message) -> {ok, TRef} | {error, Reason}
send_interval(Time, Message) -> {ok, TRef} | {error, Reason}

Types:
- Time = integer() in milliseconds
- Pid = pid() | atom()
- Message = term()
- Reason = term()

send_interval/3  Evaluates Pid ! Message repeatedly after Time amount of time has elapsed. (Pid can also be an atom of a registered name.) Returns {ok, TRef} or {error, Reason}.
send_interval/2  Same as send_interval(Time, self(), Message).

cancel(TRef) -> {ok, cancel} | {error, Reason}

Cancels a previously requested timeout. TRef is a unique timer reference returned by the timer function in question. Returns {ok, cancel}, or {error, Reason} when TRef is not a timer reference.

sleep(Time) -> ok
Types:
- Time = integer() in milliseconds
SUSPENDS THE PROCESS CALLING THIS FUNCTION FOR TIME AMOUNT OF MILLISECONDS AND THEN RETURNS OK. NATURALLY, THIS FUNCTION DOES NOT RETURN IMMEDIATELY.

tc(Module, Function, Arguments) -> {Time, Value}
Types:
- Module = Function = atom()
- Arguments = [term()]
- Time = integer() in microseconds
- Value = term()
Evaluates apply(Module, Function, Arguments) and measures the elapsed real time. Returns {Time, Value}, where Time is the elapsed real time in microseconds, and Value is what is returned from the apply.

now_diff(T2, T1) -> Tdiff
Types:
- T1 = T2 = {MegaSecs, Secs, MicroSecs}
- Tdiff = MegaSecs = Secs = MicroSecs = integer()
Calculates the time difference Tdiff = T2 - T1 in microseconds, where T1 and T2 probably are timestamp tuples returned from erlang:now/0.

seconds(Seconds) -> Milliseconds
Returns the number of milliseconds in Seconds.

minutes(Minutes) -> Milliseconds
Return the number of milliseconds in Minutes.

hours(Hours) -> Milliseconds
Returns the number of milliseconds in Hours.

hms(Hours, Minutes, Seconds) -> Milliseconds
Returns the number of milliseconds in Hours + Minutes + Seconds.
Examples

This example illustrates how to print out “Hello World!” in 5 seconds:

```erl
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.

```erl
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 and REG_BINARY which is a binary. Other types can be read, and will be returned as binaries.

There is also a "default" value, which has the empty string as name. It is read and written with the atom default instead of the name.
Some registry values are stored as strings with references to environment variables, e.g. 
"%SystemRoot%\Windows". SystemRoot is an environment variable, and should be 
replaced with its value. A function expand/1 is provided, so that environment variables 
surrounded in % can be expanded to their values.

For additional information on the Windows registry consult the Win32 Programmer's Reference.

Exports

change_key(RegHandle, Key) -> ReturnValue

Types:
- RegHandle = term()
- Key = string()

Changes the current key to another key. Works like cd. The key can be specified as a 
relative path or as an absolute path, starting with \\

change_key_create(RegHandle, Key) -> ReturnValue

Types:
- RegHandle = term()
- Key = string()

Creates a key, or just changes to it, if it is already there. Works like a combination of 
mkdir and cd. Calls the Win32 API function RegCreateKeyEx().
The registry must have been opened in write-mode.

close(RegHandle) -> ReturnValue

Types:
- RegHandle = term()

Closes the registry. After that, the RegHandle cannot be used.

current_key(RegHandle) -> ReturnValue

Types:
- RegHandle = term()
- Return Value = {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.

```erlang
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.
```

```erlang
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.
```

```erlang
format_error(ErrorId) -> ErrorString
Types:
  • ErrorId = atom()
  • ErrorString = string()
Convert an POSIX errorcode to a string (by calling erl_posix_msg:message).
```

```erlang
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.
```

```erlang
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)
The `zip` module archives and extract files to and from a zip archive. The zip format is specified by the “ZIP Appnote.txt” file available on PKWare’s website www.pkware.com. The zip module supports zip archive versions up to 6.1. However, password-protection and Zip64 is not supported.

By convention, the name of a zip file should end in “.zip”. To abide to the convention, you’ll need to add “.zip” yourself to the name.

Zip archives are created with the `zip/2` [page 364] or the `zip/3` [page 364] function. (They are also available as `create`, to resemble the `erl_tar` module.)

To extract files from a zip archive, use the `unzip/1` [page 365] or the `unzip/2` [page 365] function. (They are also available as `extract`.)

To return a list of the files in a zip archive, use the `list_dir/1` [page 366] or the `list_dir/2` [page 366] function. (They are also available as `table`.)

To print a list of files to the Erlang shell, use either the `t/1` [page 366] or `tt/1` [page 366] function.

In some cases, it is desirable to open a zip archive, and to unzip files from it file by file, without having to reopen the archive. The functions `zip_open` [page 367], `zip_get` [page 367], `zip_list_dir` [page 367] and `zip_close` [page 367] do this.

**LIMITATIONS**

Zip64 archives are not currently supported.

Password-protected and encrypted archives are not currently supported

Only the DEFLATE (zlib-compression) and the STORE (uncompressed data) zip methods are supported.

The size of the archive is limited to 2 G-byte (32 bits).

Comments for individual files is not supported when creating zip archives. The zip archive comment for the whole zip archive is supported.

There is currently no support for altering an existing zip archive. To add or remove a file from an archive, the whole archive must be recreated.
**DATA TYPES**

**zip_file()**

The record `zip_file` contains the following fields:

- `name` = `string()` the name of the file
- `info` = `file_info()` file info as in `[file:read_file_info/1]`
- `comment` = `string()` the comment for the file in the zip archive
- `offset` = `integer()` the offset of the file in the zip archive (used internally)
- `comp_size` = `integer()` the compressed size of the file (the uncompressed size is found in `info`)

**zip_comment**

The record `zip_comment` just contains the archive comment for a zip archive

- `comment` = `string()` the comment for the zip archive

**Exports**

```erlang
zip(Name, FileList) -> RetValue
zip(Name, FileList, Options) -> RetValue
create(Name, FileList) -> RetValue
create(Name, FileList, Options) -> RetValue
```

Types:

- `Name` = `filename()`
- `FileList` = `[FileSpec]`
- `FileSpec` = `filename()` | `{filename(), binary()}`
- `Options` = `[Option]`
- `Option` = `memory` | `cooked` | `verbose` | `{comment, Comment}` | `{cwd, CWD}`
- `Comment` = `CWD` = `string()`
- `RetValue` = `{ok, Name}` | `{ok, {Name, binary()}}` | `{error, Reason}`
- `Reason` = `term()`

The `zip` function creates a zip archive containing the files specified in `FileList`. As synonyms, the functions `create/2` and `create/3` are provided, to make it resemble the `erl_tar` module.

The file-list is a list of files, with paths relative to the current directory, they will be stored with this path in the archive. Files may also be specified with data in binaries, to create an archive directly from data.

Files will be compressed using the DEFLATE compression, as described in the `Appnote.txt` file. However, files will be stored without compression if they already are compressed. The `zip/2` and `zip/3` checks the file extension to see whether the file should be stored without compression. Files with the following extensions are not compressed: `.Z`, `.zip`, `.zoo`, `.arc`, `.lzh`, `.arj`.

The following options are available:
cooked By default, the open/2 function will open the zip file in raw mode, which is faster but does not allow a remote (erlang) file server to be used. Adding cooked to the mode list will override the default and open the zip file without the raw option. The same goes for the files added.

verbose Print an informational message about each file being added.

memory The output will not be to a file, but instead as a tuple \{FileName, binary()\}. The binary will be a full zip archive with header, and can be extracted with for instance unzip/2.

\{comment, Comment\} Add a comment to the zip-archive.

\{cwd, CWD\} Use the given directory as current directory, it will be prepended to file names when adding them, although it will not be in the zip-archive. (Acting like a file:set_cwd/1, but without changing the global cwd property.)

unzip(Archive) -> RetValue
unzip(Archive, Options) -> RetValue
extract(Archive) -> RetValue
extract(Archive, Options) -> RetValue

Types:
- Name = filename() | binary()
- Options = [Option]
- Option = \{files, FileList\} | keep_old_files | verbose | memory | \{file_filter, FileFilter\} | \{cwd, CWD\}
- FileList = [filename()]
- FileFilter = fun(ZipFile) -> true | false
- CWD = string()
- ZipFile = zip_file()
- RetValue = ok | \{error, Reason\} | \{error, \{Name, Reason\}\}
- Reason = term()

The unzip/1 function extracts all files from a zip archive. The unzip/2 function provides options to extract some files, and more.

If the Archive argument is given as a binary, the contents of the binary is assumed to be a zip archive, otherwise it should be a filename.

The following options are available:

- \{files, FileList\} By default, all files will be extracted from the zip archive. With the \{files, Files\} option, the unzip/2 function will only extract the files whose names are included in FileList.

cooked By default, the open/2 function will open the zip file in raw mode, which is faster but does not allow a remote (erlang) file server to be used. Adding cooked to the mode list will override the default and open zip file without the raw option. The same goes for the files extracted.

keep_old_files By default, all existing files with the same name as file in the zip archive will be overwritten. With the keep_old_files option, the unzip/2 function will not overwrite any existing files. Not that even with the memory option given, which means that no files will be overwritten, files existing will be excluded from the result.

verbose Print an informational message as each file is being extracted.
Instead of extracting to the current directory, the `memory` option will give the result as a list of tuples `{Filename, Binary}`, where `Binary` is a binary containing the extracted data of the file named `Filename` in the zip archive.

Use the given directory as current directory, it will be prepended to file names when extracting them from the zip-archive. (Acting like a `files:dir_cwd/1`, but without changing the global `cwd` property.)

```erlang
list_dir(Archive) -> RetValue
list_dir(Archive, Options)
table(Archive) -> RetValue
table(Archive, Options)
```

**Types:**
- `Archive = filename() | binary()`
- `RetVal = {ok, [Comment, Files]} | {error, Reason}`
- `Comment = zip_comment()`
- `Files = [zip_file()]`
- `Options = [Option]`
- `Option = cooked`
- `Reason = term()`

The `list_dir/1` function retrieves the names of all files in the zip archive `Archive`. The `list_dir/2` function provides options.

As synonyms, the functions `table/2` and `table/3` are provided, to make it resemble the `erl_tar` module.

The result value is the tuple `{ok, List}`, where `List` contains the zip archive comment as the first element.

The following options are available:

- `cooked` By default, the `open/2` function will open the zip file in `raw` mode, which is faster but does not allow a remote (erlang) file server to be used. Adding `cooked` to the mode list will override the default and open zip file without the `raw` option.

```erlang
t(Archive)
```

**Types:**
- `Archive = filename() | binary() | ZipHandle`
- `ZipHandle = pid()`

The `t/1` function prints the names of all files in the zip archive `Archive` to the Erlang shell. (Similar to "tar ".)

```erlang
tt(Archive)
```

**Types:**
- `Name = filename() | binary()`

The `tt/1` function prints names and information about all files in the zip archive `Archive` to the Erlang shell. (Similar to "tar tv ".)

```erlang
zip_open(Archive) -> {ok, ZipHandle} | {error, Reason}
```
zip_open(Archive, Options) -> {ok, ZipHandle} | {error, Reason}

Types:
- Name = filename() | binary()
- Options = [Option]
- Options = cooked | memory | {cwd, CWD}
- CWD = string()
- ZipHandle = pid()

The `zip_open` function opens a zip archive, and reads and saves its directory. This means that subsequently reading files from the archive will be faster than unzipping files one at a time with `unzip`.

The archive must be closed with `zip_close/1`.

zip_list_dir(ZipHandle) -> Result | {error, Reason}

Types:
- Result = [ZipComment, ZipFile..]
- ZipComment = #zip_comment{}
- ZipFile = #zip_file{}
- ZipHandle = pid()

The `zip_list_dir/1` function returns the file list of an open zip archive.

zip_get(ZipHandle) -> {ok, Result} | {error, Reason}
zip_get(FileName, ZipHandle) -> {ok, Result} | {error, Reason}

Types:
- Name = filename() | binary() | ZipHandle
- ZipHandle = pid()

The `zip_get` function extracts one or all files from an open archive. The files will be unzipped to memory or to file, depending on the options given to the `zip_open` function when the archive was opened.

zip_close(ZipHandle) -> ok | {error, einval}

Types:
- ZipHandle = pid()

The `zip_close/1` function closes a zip archive, previously opened with `zip_open`. All resources are closed, and the handle should not be used after closing.
Index of Modules and Functions

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

a_function/2
    sofs , 313
abcast/2
    gen_server , 198
abcast/3
    gen_server , 198
absname/1
    filename , 156
absname/2
    filename , 157
absname_join/2
    filename , 157
abstract/1
    erl_parse , 115
acos/1
    math , 236
acosh/1
    math , 236
add/2
    gb_sets , 163
add/3
    erl_tar , 123
add/4
    erl_tar , 123
add_binding/3
    erl_eval , 106
add_edge/3
    digraph , 92
add_edge/4
    digraph , 92
add_edge/5
    digraph , 92
add_element/2
    gb_sets , 163
    sets , 294
    add_handler/3
        gen_event , 174
    add_sup_handler/3
        gen_event , 175
    add_vertex/1
        digraph , 93
    add_vertex/2
        digraph , 93
    add_vertex/3
        digraph , 93
all/0
    dets , 72
    ets , 129
all/2
    lists , 230
any/2
    lists , 230
append/1
    lists , 220
    qlc , 272
append/2
    lists , 220
    qlc , 272
append/3
    dict , 87
append_list/3
    dict , 87
append_values/2
    proplists , 260
apply_after/4
    timer , 355
apply_interval/4
timer, 356

arith_op/2
  erl_internal, 110

asin/1
  math, 236

asinh/1
  math, 236

atan/1
  math, 236

atan2/2
  math, 236

atanh/1
  math, 236

attach/1
  pool, 253

attribute/1
  erl_pp, 117

attribute/2
  erl_pp, 117

balance/1
  gb_sets, 163
  gb_trees, 168

base64
  decode/1, 53
  decode_to_string/1, 53
  encode/1, 53
  encode_to_string/1, 53
  mime_decode/1, 53
  mime_decode_string/1, 53

basename/1
  filename, 157

basename/2
  filename, 157

bchunk/2
  dets, 72

beam_lib
  chunks/2, 57
  chunks/3, 57
  clear_crypto_key_fun/0, 61
  cmp/2, 59
  cmp_dirs/2, 59
  crypto_key_fun/1, 60
  diff_dirs/2, 59
  format_error/1, 60
  info/1, 58

md5/1, 58
strip/1, 59
strip_files/1, 59
strip_release/1, 60
version/1, 57

bif/2
  erl_internal, 110

binding/2
  erl_eval, 106

bindings/1
  erl_eval, 106

bool_op/2
  erl_internal, 110

bt/1
  c, 62

  c
    bt/1, 62
    c/1, 62
    c/2, 62
    cd/1, 62
    flush/0, 63
    help/0, 63
    i/0, 63
    i/3, 63
    l/1, 63
    lc/1, 63
    ls/0, 63
    ls/1, 63
    m/0, 64
    m/1, 64
    memory/0, 64
    memory/1, 64
    nc/1, 64
    nc/2, 64
    ni/0, 63
    nl/1, 64
    nregs/0, 65
    pid/3, 64
    pwd/0, 65
    q/0, 65
    regs/0, 65
    xm/1, 65
    y/1, 65
    y/2, 65

370  STDLIB
calendar
  date_to_gregorian_days/1, 67
  date_to_gregorian_days/3, 67
  datetime_to_gregorian_days/2, 67
  day_of_the_week/1, 67
  day_of_the_week/3, 67
  gregorian_days_to_date/1, 67
  gregorian_seconds_to_gregorian/1, 67
  gregorian_seconds_to_datetime/1, 67
  is_leap_year/1, 67
  last_day_of_the_month/2, 67
  local_time/0, 68
  local_time_to_universal_time/2, 68
  local_time_to_universal_time_dst/2, 68
  now_to_datetime/1, 69
  now_to_local_time/1, 68
  now_to_universal_time/1, 69
  seconds_to_daytime/1, 69
  seconds_to_time/1, 69
  time_difference/2, 69
  time_to_seconds/1, 69
  universal_time/0, 69
  universal_time_to_local_time/2, 70
  valid_date/1, 70
  valid_date/3, 70

call/2
  gen_server, 196

call/3
  gen_event, 176
  gen_server, 196

call/4
  gen_event, 176

cancel/1
  timer, 356

cancel_timer/1
  gen_fsm, 188

canonical_relation/1
  sofs, 313

cast/2
  gen_server, 198

cd/1
  c, 62

centre/2
  string, 335

centre/3
  string, 335

change_code/4
  sys, 350

change_code/5
  sys, 350

change_key/2
  win32reg, 360

change_key_create/2
  win32reg, 360

char_list/1
  io_lib, 217

chars/2
  string, 333

chars/3
  string, 333

check/1
  file_sorter, 152

check/2
  file_sorter, 152

check_childspecs/1
  supervisor, 343

chr/2
  string, 332

chunks/2
  beam_lib, 57

chunks/3
  beam_lib, 57

clear_crypto_key_fun/0
  beam_lib, 61

close/1
  dets, 73
  epp, 103
  erl_tar, 123
  win32reg, 360

cmp/2
  beam_lib, 59

cmp_dirs/2
  beam_lib, 59

comp_op/2
  erl_internal, 111

compact/1
  proplists, 260

components/1
  digraph_utils, 100

composite/2
  sofs, 313
```
concat/1
lists, 220
concat/2
string, 332
condensation/1
digraph_utils, 100
cons/2
queue, 282
constant_function/2
sofs, 313
converse/1
sofs, 314
copies/2
string, 334
cos/1
math, 236
cosh/1
math, 236
create/1
pg, 251
create/2
erl_tar, 123
pg, 251
zip, 364
create/3
erl_tar, 124
zip, 364
crypto_key_fun/1
beam_lib, 60
cspan/2
string, 333
current_key/1
win32reg, 360
cursor/2
qic, 272
cyclic_strong_components/1
digraph_utils, 100
daeh/1
queue, 282
date_to_gregorian_days/1
calendar, 67
date_to_gregorian_days/2
calendar, 67
date_to_gregorian_days/3
calendar, 67
datetime_to_gregorian_seconds/2
calendar, 67
day_of_the_week/1
calendar, 67
day_of_the_week/3
calendar, 67
debug_options/1
sys, 352
decode/1
base64, 53
decode_to_string/1
base64, 53
deep_char_list/1
io_lib, 217
del_binding/2
erl_eval, 106
del_edge/2
digraph, 93
del_edges/2
digraph, 93
del_element/2
gb_sets, 163
sets, 294
del_path/3
digraph, 93
del_vertex/2
digraph, 94
del_vertices/2
digraph, 94
delete/1
digraph, 94
ets, 129
delete/2
dets, 73
ets, 129
gb_sets, 163
gb_trees, 168
lists, 221
proplists, 260
delete_all_objects/1
dets, 73
ets, 129
delete_any/2
gb_sets, 163
gb_trees, 169
```
delete_child/2
  supervisor, 342
delete_cursor/1
  qlc , 273
delete_handler/3
gen_event, 177
delete_key/1
  win32reg , 360
delete_object/2
dets, 73
  ets, 129
delete_value/2
  win32reg, 361
dets
  all/0, 72
  bchunk/2, 72
  close/1, 73
  delete/2, 73
  delete_all_objects/1, 73
  delete_object/2, 73
  first/1, 73
  fold1/3, 74
  foldr/3, 74
  from_ets/2, 74
  info/1, 74
  info/2, 75
  init_table/3, 75
  insert/2, 76
  insert_new/2, 76
  is_compatible_bchunk_format/2, 77
  is_dets_file/1, 77
  lookup/2, 77
  match/1, 77
  match/2, 78
  match/3, 78
  match_delete/2, 78
  match_object/1, 79
  match_object/2, 79
  match_object/3, 79
  member/2, 80
  next/2, 80
  open_file/1, 80
  open_file/2, 80
  pid2name/1, 82
  repair_continuation/2, 82
  safe_fixtable/2, 82
  select/1, 83
  select/2, 83
  select/3, 83
  select_delete/2, 84
  slot/2, 84
  sync/1, 84
  table/2, 85
  to_ets/2, 85
  traverse/2, 86
  update_counter/3, 86
dict
  append/3, 87
  append_list/3, 87
  erase/2, 87
  fetch/2, 87
  fetch_keys/1, 88
  filter/2, 88
  find/2, 88
  fold/3, 88
  from_list/1, 88
  is_key/2, 88
  map/2, 89
  merge/3, 89
  new/0, 89
  store/3, 89
  to_list/1, 89
  update/3, 90
  update/4, 90
  update_counter/3, 90
diff_dirs/2
  beam_lib, 59
difference/2
gb_sets, 163
  sofs, 314
digraph
  add_edge/3, 92
  add_edge/4, 92
  add_edge/5, 92
  add_vertex/1, 93
  add_vertex/2, 93
  add_vertex/3, 93
  del_edge/2, 93
  del_edges/2, 93
  del_path/3, 93
  del_vertex/2, 94
  del_vertices/2, 94
  delete/1, 94
  edge/2, 94
  edges/1, 94
  edges/2, 94
  get_cycle/2, 95
  get_path/3, 95
  get_short_cycle/2, 95
  get_short_path/3, 95
  in_degree/2, 96
in_edges/2, 96
in_neighbours/2, 96
info/1, 96
new/0, 97
new/1, 97
no_edges/1, 97
no_vertices/1, 97
out_degree/2, 97
out_edges/2, 97
out_neighbours/2, 97
vertex/2, 98
vertices/1, 98
digraph_to_family/2
  sofs, 314
digraph_utils
  components/1, 100
  condensation/1, 100
  cyclic_strong_components/1, 100
  is_acyclic/1, 100
  loop_vertices/1, 100
  postorder/1, 101
  preorder/1, 101
  reachable/2, 101
  reachable_neighbours/2, 101
  reaching/2, 101
  reaching_neighbours/2, 101
  strong_components/1, 102
  subgraph/3, 102
  toposort/1, 102
dirname/1
  filename, 158
domain/1
  sofs, 314
drestriction/2
  sofs, 314
drestriction/3
  sofs, 315
dropwhile/2
  lists, 231
duplicate/2
  lists, 221
e/2
  qlc, 273
edge/2
  digraph, 94
edges/1
  digraph, 94
dges/2
  digraph, 94
empty/0
  gb_sets, 164
  gb_trees, 169
empty_set/0
  sofs, 315
encode/1
  base64, 53
encode_to_string/1
  base64, 53
ensure_dir/1
  filelib, 153
enter/3
  gb_trees, 169
enter_loop/3
  gen_server, 199
enter_loop/4
  gen_fsm, 188
  gen_server, 199
enter_loop/5
  gen_fsm, 188
  gen_server, 199
enter_loop/6
  gen_fsm, 188
epp
  close/1, 103
  open/2, 103
  open/3, 103
  parse_erl_form/1, 103
  parse_file/3, 103
equal/2
  string, 332
erase/2
  dict, 87
erf/1
  math, 237
erfc/1
  math, 237
erl_eval
  add_binding/3, 106
  binding/2, 106
  bindings/1, 106
  del_binding/2, 106
  expr/2, 105
expr/3, 105
expr/4, 105
expr_list/2, 106
expr_list/3, 106
expr_list/4, 106
exprs/2, 105
exprs/3, 105
exprs/4, 105
new_bindings/0, 106

erl_expand_records
  module/2, 108

erl_id_trans
  parse_transform/2, 109

erl_internal
  arith_op/2, 110
  bif/2, 110
  bool_op/2, 110
  comp_op/2, 111
  guard_bif/2, 110
  list_op/2, 111
  op_type/2, 111
  send_op/2, 111
  type_test/2, 110

erl_lint
  format_error/1, 113
  is_guard_test/1, 113
  module/1, 112
  module/2, 112
  module/3, 112

erl_parse
  abstract/1, 115
  format_error/1, 115
  normalise/1, 115
  parse_expr/1, 114
  parse_form/1, 114
  parse_term/1, 114
  tokens/1, 115
  tokens/2, 115

erl_pp
  attribute/1, 117
  attribute/2, 117
  expr/1, 118
  expr/2, 118
  expr/3, 118
  expr/4, 118
  exprs/1, 118
  exprs/2, 118
  exprs/3, 118
  form/1, 117
  form/2, 117
  function/1, 117
  function/2, 117
  guard/1, 117
  guard/2, 117

erl_scan
  format_error/1, 121
  reserved_word/1, 121
  string/1, 120
  string/2, 120
  tokens/3, 120

erl_tar
  add/3, 123
  add/4, 123
  close/1, 123
  create/2, 123
  create/3, 124
  extract/1, 124
  extract/2, 124
  format_error/1, 125
  open/2, 125
  t/1, 126
  table/1, 126
  table/2, 126
  tt/1, 126

error_message/2
  lib , 218

esend/2
  pg , 252

et
  all/0, 129
  delete/1, 129
  delete/2, 129
  delete_all_objects/1, 129
  delete_object/2, 129
  file2tab/1, 129
  first/1, 130
  fixtable/2, 130
  foldl/3, 130
  foldr/3, 130
  from_dets/2, 131
  fun2ms/1, 131
  i/0, 132
  i/1, 132
  info/1, 132
  info/2, 133
  init_table/2, 133
  insert/2, 134
  insert_new/2, 134
  is_compiled_ms/1, 134
  last/1, 135
lookup/2, 135
lookup_element/3, 135
match/1, 137
match/2, 136
match/3, 136
match_delete/2, 137
match_object/1, 137
match_object/2, 137
match_object/3, 137
match_spec Compile/1, 138
match_spec_run/2, 138
member/2, 139
new/2, 139
next/2, 140
prev/2, 140
rename/2, 140
repair_continuation/2, 140
safe_fixtable/2, 141
select/1, 144
select/2, 142
select/3, 144
select_count/2, 144
select_delete/2, 144
slot/2, 145
tab2file/2, 145
tab2list/1, 145
table/2, 145
to_dets/2, 147
update_counter/3, 147
update_counter/4, 147
update_counter/6, 147
eval/2
qlc, 273
exit_after/2
timer, 356
exit_after/3
timer, 356
exp/1
math, 236
expand/1
win32reg, 361
expand/2
proplists, 260
expr/1
erl_pp, 118
expr/2
erl_eval, 105
erl_pp, 118
eval
expr/3
erl_eval, 105
erl_pp, 118
expr/4
erl_eval, 105
erl_pp, 118
expr_list/2
erl_eval, 106
expr_list/3
erl_eval, 106
expr_list/4
erl_eval, 106
exps/1
erl_pp, 118
exps/2
erl_eval, 105
erl_pp, 118
exps/3
erl_eval, 105
erl_pp, 118
exps/4
erl_eval, 105
extension/1
filename, 158
extension/3
sofs, 315
extract/1
erl_tar, 124
zip, 365
extract/2
erl_tar, 124
zip, 365
family/2
sofs, 316
family_difference/2
sofs, 316
family_domain/1
sofs, 316
family_field/1
sofs, 316
family_intersection/1
sofs, 317
family_intersection/2
sofs, 317
family_projection/2
  sofs, 317
family_range/1
  sofs, 317
family_specification/2
  sofs, 318
family_to_digraph/2
  sofs, 318
family_to_relation/1
  sofs, 318
family_union/1
  sofs, 319
family_union/2
  sofs, 319
fetch/2
  dict, 87
fetch_keys/1
  dict, 88
field/1
  sofs, 319
file2tab/1
  ets, 129
file_size/1
  filelib, 153
file_sorter
  check/1, 152
  check/2, 152
  keycheck/2, 152
  keycheck/3, 152
  keymerge/3, 152
  keymerge/4, 152
  keysort/2, 151
  keysort/3, 151
  keysort/4, 151
  merge/2, 151
  merge/3, 151
  sort/1, 151
  sort/2, 151
  sort/3, 151
filelib
  ensure_dir/1, 153
  file_size/1, 153
  fold_files/5, 153
  is_dir/1, 153
  is_file/1, 153
  is_regular/1, 154
  last_modified/1, 154
  wildcard/1, 154
  wildcard/2, 155
filename
  absname/1, 156
  absname/2, 157
  absname_join/2, 157
  basename/1, 157
  basename/2, 157
  dirname/1, 158
  extension/1, 158
  find_src/1, 160
  find_src/2, 160
  flatten/1, 158
  join/1, 159
  join/2, 159
  nativename/1, 159
  pathtype/1, 159
  rootname/1, 160
  rootname/2, 160
  split/1, 160
filter/2
  dict, 88
  gb_sets, 164
  lists, 231
  sets, 295
find/2
  dict, 88
find_src/1
  filename, 160
find_src/2
  filename, 160
first/1
  dets, 73
  ets, 130
first_match/2
  regexp, 288
fixtable/2
  ets, 130
flatlength/1
  lists, 221
flatmap/2
  lists, 231
flatten/1
  filename, 158
  lists, 221
flatten/2
  lists, 221
flush/0
  c, 63
flush_receive/0
  lib, 218
fold/3
  dict, 88
  gb_sets, 164
  sets, 295
fold/4
  qic, 273
fold_files/5
  filelib, 153
foldl/3
  dets, 74
  ets, 130
  lists, 231
foldr/3
  dets, 74
  ets, 130
  lists, 232
foreach/2
  lists, 232
form/1
  erl_pp, 117
form/2
  erl_pp, 117
format/1
  io, 205
  proc_lib, 258
format/2
  io_lib, 215
format/3
  io, 205
format_error/1
  beam_lib, 60
  erl_bint, 113
  erl_parse, 115
  erl_scan, 121
  erl_tar, 125
  ms_transform, 248
  qic, 273
  regexp, 290
  win32reg, 361
fread/2
  io_lib, 215
fread/3
  io, 205
from_dets/2
  ets, 131
from_ets/2
  dets, 74
from_external/2
  sofs, 319
from_list/1
  dict, 88
  gb_sets, 164
  queue, 282
  sets, 293
from_orddict/1
  gb_trees, 169
from_ordset/1
  gb_sets, 164
from_sets/1
  sofs, 319, 320
from_term/2
  sofs, 320
fun2ms/1
  ets, 131
function/1
  erl_pp, 117
function/2
  erl_pp, 117
fwrite/1
  io, 205
fwrite/2
  io_lib, 215
fwrite/3
  io, 205
gb_sets
  add/2, 163
  add_element/2, 163
  balance/1, 163
  del_element/2, 163
  delete/2, 163
  delete_any/2, 163
  difference/2, 163
  empty/0, 164
  filter/2, 164
  fold/3, 164
  from_list/1, 164
Module: handle_cast/2, 201
Module: handle_info/2, 201
Module: init/1, 200
Module: terminate/2, 201
multi_call/2, 197
multi_call/3, 197
multi_call/4, 197
reply/2, 198
start/3, 195
start/4, 195
start_link/3, 194
start_link/4, 194
get/2
  gb_trees, 169
get_all_values/2
  proplists, 261
get_bool/2
  proplists, 261
get_chars/3
  io, 203
get_cycle/2
digraph, 95
get_debug/3
  sys, 352
get_keys/1
  proplists, 261
get_line/2
  io, 204
get_node/0
  pool, 254
get_nodes/0
  pool, 254
get_path/3
digraph, 95
get_short_cycle/2
digraph, 95
get_short_path/3
digraph, 95
get_status/1
  sys, 350
get_status/2
  sys, 350
get_value/2
  proplists, 262
get_value/3
  proplists, 262
gregoriant_days_to_date/1
calendar, 67
gregoriant_seconds_to_datetime/1
calendar, 67
gsub/3
  regexp, 289
guard/1
erl_pp, 117
guard/2
erl_pp, 117
guard_bif/2
erl_internal, 110
handle_debug/1
  sys, 352
handle_system_msg/6
  sys, 352
head/1
  queue, 282
help/0
c, 63
hibernate/3
  proc_lib, 259
history/1
  shell, 304
hms/3
timer, 357
hours/1
timer, 357
i/0
c, 63
  ets, 132
i/1
  ets, 132
i/3
c, 63
image/2
  sofs, 321
in/2
  queue, 282
in_degree/2
digraph, 96
install/4
    sys, 351
intersection/1
    gb_sets, 165
    sets, 294
    sofs, 321
intersection/2
    gb_sets, 165
    sets, 294
    sofs, 321
intersection_of_family/1
    sofs, 321
inverse/1
    sofs, 321
inverse_image/2
    sofs, 322
io
    format/1, 205
    format/3, 205
    fread/3, 209
    fwrite/1, 205
    fwrite/3, 205
    get_chars/3, 203
    get_line/2, 204
    nl/1, 203
    parse_erl_exprs/1, 212
    parse_erl_exprs/3, 212
    parse_erl_form/1, 212
    parse_erl_form/3, 212
    put_chars/2, 203
    read/2, 205
    read/3, 205
    scan_erl_exprs/1, 211
    scan_erl_exprs/3, 211
    scan_erl_form/1, 211
    scan_erl_form/3, 211
    setopts/2, 204
    write/2, 204
io_lib
    char_list/1, 217
    deep_char_list/1, 217
    format/2, 215
    fread/2, 215
    fread/3, 215
    fwrite/2, 215
    indentation/2, 216
    nl/0, 214
    print/1, 214
    print/4, 214
is_a_function/1
sofs, 322

is_acyclic/1
digraph_utils, 100

is_compatible_bchunk_format/2
dets, 77

is_compiled_ms/1
ets, 134

is_defined/2
gb_trees, 170
proplists, 262

is_dets_file/1
dets, 77

is_dir/1
filelib, 153

is_disjoint/2
sofs, 322

is_element/2
gb_sets, 165
sets, 293

is_empty/1
gb_sets, 165
gb_trees, 170
queue, 283

is_empty_set/1
sofs, 322

is_equal/2
sofs, 322

is_file/1
filelib, 153

is_guard_test/1
erl_int, 113

is_key/2
dict, 88

is_leap_year/1
calendar, 67

is_member/2
gb_sets, 165

is_regular/1
filelib, 154

is_set/1
gb_sets, 165
sofs, 322

is_sofs_set/1
sofs, 323

is_subset/2
gb_sets, 165
sofs, 323

is_type/1
sofs, 323

iterator/1
gb_sets, 165
gb_trees, 170

join/1
filename, 159

join/2
filename, 159
pg, 251
queue, 283

join/4
sofs, 323

keycheck/2
file_sorter, 152

keycheck/3
file_sorter, 152

keydelete/3
lists, 221

keymap/3
lists, 232

keymember/3
lists, 222

keymerge/3
file_sorter, 152
lists, 222

keymerge/4
file_sorter, 152

keyreplace/4
lists, 222

keys/1
gb_trees, 170
keysearch/3
  lists, 222
keysort/2
  file_sorter, 151
  lists, 222
keysort/3
  file_sorter, 151
  q/c, 275
keysort/4
  file_sorter, 151
kill_after/1
  timer, 356
kill_after/2
  timer, 356
1/1
  c, 63
lait/1
  queue, 283
largest/1
  gb_sets, 166
  gb_trees, 170
last/1
  ets, 135
  lists, 223
  queue, 283
last_day_of_the_month/2
  calendar, 67
last_modified/1
  filelib, 154
lc/1
  c, 63
left/2
  string, 335
left/3
  string, 335
len/1
  queue, 283
  string, 332
lib
  error_message/2, 218
  flush_receive/0, 218
  nonl/1, 218
  proname/0, 218
send/2, 218
sendv/2, 219
list_dir/1
  zip, 366
list_dir/2
  zip, 366
list_op/2
  erl_internal, 111
lists
  all/2, 230
  any/2, 230
  append/1, 220
  append/2, 220
  concat/1, 220
  delete/2, 221
  dropwhile/2, 231
  duplicate/2, 221
  filter/2, 231
  flatlength/1, 221
  flatmap/2, 231
  flatten/1, 221
  flatten/2, 221
  foldl/3, 231
  foldr/3, 232
  foreach/2, 232
  keydelete/3, 221
  keymap/3, 232
  keymember/3, 222
  keymerge/3, 222
  keyreplace/4, 222
  keysearch/3, 222
  keysort/2, 222
  last/1, 223
  map/2, 233
  mapfoldl/3, 233
  mapfoldr/3, 233
  max/1, 223
  member/2, 223
  merge/1, 223
  merge/2, 223
  merge/3, 223
  merge3/3, 224
  min/1, 224
  nth/2, 224
  nthtail/2, 224
  partition/2, 233
  prefix/2, 224
  reverse/1, 225
  reverse/2, 225
  seq/2, 225
  seq/3, 225

STDLIB 383
sort/1, 225
sort/2, 225
split/2, 226
splitwith/2, 234
sublist/2, 226
sublist/3, 226
subtract/2, 226
suffix/2, 227
sum/1, 227
takewhile/2, 234
ukeymerge/3, 227
ukeysort/2, 227
umerge/1, 227
umerge/2, 227
umerge/3, 228
umerge3/3, 228
unzip/1, 228
unzip3/1, 228
usort/1, 228
usort/2, 229
zip/2, 229
zip3/3, 229
zipwith/3, 229
zipwith/3, 230

local_time/0
calendar ,68

local_time_to_universal_time/2
calendar ,68

local_time_to_universal_time_dst/2
calendar ,68

log/1
math ,236

log/2
sys ,349

log/3
sys ,349

log10/1
math ,236

log_mf_h
init/3, 235
init/4, 235

log_to_file/2
sys ,349

log_to_file/3
sys ,349

lookup/2
dets ,77
ets ,135

gb_trees ,169
proplists ,262

lookup_all/2
proplists ,262

lookup_element/3
ets ,135

loop_vertices/1
digraph_utils ,100

ls/0
c ,63

ls/1
c ,63

m/0
c ,64

m/1
c ,64

map/2
dict ,89
lists ,233

mapfoldl/3
lists ,233

mapfoldr/3
lists ,233

match/1
dets ,77
ets ,137

match/2
dets ,78
ets ,136
regexp ,288

match/3
dets ,78
ets ,136

match_delete/2
dets ,78
ets ,137

match_object/1
dets ,79
ets ,137

match_object/2
dets ,79
ets ,137

match_object/3
dets ,79
ets, 137
match_spec_compile/1
ets, 138
match_spec_run/2
ets, 138
matches/2
regexp, 288
math
acos/1, 236
acosh/1, 236
asin/1, 236
asinh/1, 236
atan/1, 236
atan2/2, 236
atanh/1, 236
cos/1, 236
cosh/1, 236
erf/1, 237
erfc/1, 237
exp/1, 236
log/1, 236
log10/1, 236
pi/0, 236
pow/2, 236
sin/1, 236
sinh/1, 236
sqrt/1, 236	
tan/1, 236
tanh/1, 236
max/1
lists, 223
md5/1
beam_lib, 58
member/2
dets, 80
ets, 139
lists, 223
members/1
pg, 252
memory/0
c, 64
memory/1
c, 64
merge/1
lists, 223
merge/2
file_sorter, 151
lists, 223
merge/3
dict, 89
file_sorter, 151
lists, 223
merge3/3
lists, 224
mime_decode/1
base64, 53
mime_decode_string/1
base64, 53
min/1
lists, 224
minutes/1
timer, 357
Mod:system_code_change/4
sys, 353
Mod:system_continue/3
sys, 353
Mod:system_terminate/4
sys, 353
module/1
erl_lint, 112
module/2
erl_expand_records, 108
erl_lint, 112
module/3
erl_lint, 112
Module:code_change/3
gen_event, 181
gen_server, 202
Module:code_change/4
gen_fsm, 193
Module:handle_call/2
gen_event, 180
Module:handle_call/3
gen_server, 200
Module:handle_cast/2
gen_server, 201
Module:handle_event/2
gen_event, 179
Module:handle_event/3
gen_fsm, 190
start/1, 253
start/2, 253
stop/0, 254

postorder/1
digraph_utils, 101

pow/2
  math, 236

prefix/2
  lists, 224

preorder/1
digraph_utils, 101

prev/2
  ets, 140

print/1
  io_lib, 214

print/4
  io_lib, 214

print_log/1
  sys, 353

printable_list/1
  io_lib, 217

proc_lib
  format/1, 258
  hibernate/3, 259
  init_ack/1, 257
  init_ack/2, 257
  initial_call/1, 258
  spawn/1, 255
  spawn/2, 255
  spawn/3, 255
  spawn/4, 255
  spawn_link/1, 255
  spawn_link/2, 255
  spawn_link/3, 255
  spawn_link/4, 256
  spawn_opt/2, 256
  spawn_opt/3, 256
  spawn_opt/4, 256
  spawn_opt/5, 256
  start/3, 256
  start/4, 256
  start/5, 256
  start_link/3, 256
  start_link/4, 256
  start_link/5, 256
  translate_initial_call/1, 258

product/1
  sofs, 325

product/2
  sofs, 325

proggname/0
  lib, 218

projection/2
  sofs, 326

property/1
  proplists, 263

property/2
  proplists, 263

proplists
  append_values/2, 260
  compact/1, 260
  delete/2, 260
  expand/2, 260
  get_all_values/2, 261
  get_bool/2, 261
  get_keys/1, 261
  get_value/2, 262
  get_value/3, 262
  is_defined/2, 262
  lookup/2, 262
  lookup_all/2, 262
  normalize/2, 262
  property/1, 263
  property/2, 263
  split/2, 263
  substitute_aliases/2, 264
  substitute_negations/2, 264
  unfold/1, 264

pseudo/1
  slave, 308

pseudo/2
  slave, 308

pspawn/3
  pool, 254

pspawn_link/3
  pool, 254

put_chars/2
  io, 203

pwd/0
  c, 65

q/0
  c, 65

q/2
  qlc, 275
q/c
  append/1, 272
  append/2, 272
  cursor/2, 272
  delete_cursor/1, 273
  e/2, 273
  eval/2, 273
  fold/4, 273
  format_error/1, 273
  info/2, 274
  keysort/3, 275
  next_answers/2, 275
  q/2, 275
  sort/2, 278
  string_to_handle/3, 278
  table/2, 279
queue
  cons/2, 282
  daeh/1, 282
  from_list/1, 282
  head/1, 282
  in/2, 282
  in_r/2, 283
  init/1, 283
  is_empty/1, 283
  join/2, 283
  lait/1, 283
  last/1, 283
  len/1, 283
  new/0, 284
  out/1, 284
  out_r/1, 284
  reverse/1, 284
  snoc/2, 284
  split/2, 284
  tail/1, 284
  to_list/1, 285
random
  seed/0, 286
  seed/3, 286
  seed0/0, 286
  uniform/0, 286
  uniform/1, 286
  uniform_s/1, 287
  uniform_s/2, 287
range/1
  sofs, 326
rchr/2
  string, 332
reachable/2

digraph_utils, 101
  reachable_neighbours/2
digraph_utils, 101
reaching/2
digraph_utils, 101
reaching_neighbours/2
digraph_utils, 101
read/2
  io, 205
read/3
  io, 205
regexp
  first_match/2, 288
  format_error/1, 290
  gsub/3, 289
  match/2, 288
  matches/2, 288
  parse/1, 290
  sh_to_awk/1, 290
  split/2, 289
  sub/3, 289
regs/0
  c, 65
relation/2
  sofs, 326
relation_to_family/1
  sofs, 326
relative_product/2
  sofs, 327
relative_product1/2
  sofs, 327
relay/1
  slave, 308
remove/2
  sys, 351
remove/3
  sys, 351
rename/2
  ets, 140
repair_continuation/2
  dets, 82
  ets, 140
reply/2
  gen_fsm, 187
  gen_server, 198
seq/2
  lists, 225
seq/3
  lists, 225
set/2
  sofs, 328
set_value/3
  win32reg, 361
setopts/2
  io, 204
sets
  add_element/2, 294
del_element/2, 294
filter/2, 295
do{, 295
from_list/1, 293
intersection/1, 294
intersection/2, 294
is_element/2, 293
is_set/1, 293
is_subset/2, 295
new/0, 293
size/1, 293
subtract/2, 294
to_list/1, 293
union/1, 294
union/2, 294
sh_to_awk/1
  regexp, 290
shell
  history/1, 304
results/1, 304
start_restricted/1, 304
stop_restricted/0, 304
sin/1
  math, 236
singleton/1
gb_sets, 166
sinh/1
  math, 236
size/1
gb_sets, 166
gb_trees, 171
sets, 293
slave
  pseudo/1, 308
  pseudo/2, 308
relay/1, 308
start/1, 306
start/2, 306
start/3, 306
start_link/1, 307
start_link/2, 307
start_link/3, 307
stop/1, 308
sleep/1
timer, 356
slot/2
dets, 84
ets, 145
smallest/1
gb_sets, 166
gb_trees, 171
snoc/2
  queue, 284
sof
  a_function/2, 313
canonical_relation/1, 313
composite/2, 313
constant_function/2, 313
converse/1, 314
difference/2, 314
digraph_to_family/2, 314
domain/1, 314
drestriction/2, 314
drestriction/3, 315
empty_set/0, 315
extension/3, 315
family/2, 316
family_difference/2, 316
family_domain/1, 316
family_field/1, 316
family_intersection/1, 317
family_intersection/2, 317
family_projection/2, 317
family_range/1, 317
family_specification/2, 318
family_to_digraph/2, 318
family_to_relation/1, 318
family_union/1, 319
family_union/2, 319
field/1, 319
from_external/2, 319
from_sets/1, 319, 320
from_term/2, 320
image/2, 321
intersection/1, 321

STDLIB

391
intersection/2, 321
intersection_of_family/1, 321
inverse/1, 321
inverse_image/2, 322
is_a_function/1, 322
is_disjoint/2, 322
is_empty_set/1, 322
is_equal/2, 322
is_set/1, 322
is_sofs_set/1, 323
is_subset/2, 323
is_type/1, 323
join/4, 323
multiple_relative_product/2, 323
no_elements/1, 324
partition/1, 324
partition/2, 324
partition/3, 324
partition_family/2, 325
product/1, 325
product/2, 325
projection/2, 326
range/1, 326
relation/2, 326
relation_to_family/1, 326
relative_product/2, 327
relative_product1/2, 327
restriction/2, 327
restriction/3, 328
set/2, 328
specification/2, 328
strict_relation/1, 328
substitution/2, 329
symdiff/2, 329
symmetric_partition/2, 330
to_external/1, 330
to_sets/1, 330
type/1, 330
union/1, 330
union/2, 330
union_of_family/1, 331
weak_relation/1, 331

sort/1
  file_sorter, 151
  lists, 225

sort/2
  file_sorter, 151
  lists, 225
  qlc, 278

sort/3
  file_sorter, 151

span/2
  string, 333

spawn/1
  proc_lib, 255

spawn/2
  proc_lib, 255

spawn/3
  proc_lib, 255

spawn/4
  proc_lib, 255

spawn_link/1
  proc_lib, 255

spawn_link/2
  proc_lib, 255

spawn_link/3
  proc_lib, 255

spawn_link/4
  proc_lib, 256

spawn_opt/2
  proc_lib, 256

spawn_opt/3
  proc_lib, 256

spawn_opt/4
  proc_lib, 256

spawn_opt/5
  proc_lib, 256

specification/2
  sofs, 328

split/1
  filename, 160

split/2
  lists, 226
  proplists, 263
  queue, 284
  regexp, 289

splitwith/2
  lists, 234

sqrt/1
  math, 236

start/0
  gen_event, 174
  timer, 355

start/1
  gen_event, 174
pool, 253
slave, 306
start/2
  pool, 253
  slave, 306
start/3
  gen_fsm, 185
  gen_server, 195
  proc_lib, 256
  slave, 306
start/4
  gen_fsm, 185
  gen_server, 195
  proc_lib, 256
start/5
  proc_lib, 256
start_child/2
  supervisor, 341
start_link/0
  gen_event, 174
start_link/1
  gen_event, 174
  slave, 307
start_link/2
  slave, 307
  supervisor, 340
  supervisor_bridge, 345
start_link/3
  gen_fsm, 184
  gen_server, 194
  proc_lib, 256
  slave, 307
  supervisor, 340
  supervisor_bridge, 345
start_link/4
  gen_fsm, 184
  gen_server, 194
  proc_lib, 256
start_link/5
  proc_lib, 256
start_restricted/1
  shell, 304
start_timer/2
  gen_fsm, 187
statistics/2
  sys, 349
stop/0
  pool, 254
stop/1
  gen_event, 179
  slave, 308
stop_restricted/0
  shell, 304
store/3
  dict, 89
str/2
  string, 332
strict_relation/1
  sofs, 328
string
  centre/2, 335
  centre/3, 335
  chars/2, 333
  chars/3, 333
  chr/2, 332
  concat/2, 332
  copies/2, 334
cspan/2, 333
equal/2, 332
left/2, 335
left/3, 335
len/1, 332
rchr/2, 332
right/2, 335
right/3, 335
rstr/2, 332
span/2, 333
str/2, 332
strip/1, 334
strip/2, 334
strip/3, 334
sub_string/2, 335
sub_string/3, 336
sub_word/2, 334
sub_word/3, 334
substr/2, 333
substr/3, 333
to_float/1, 336
to_integer/1, 336
to_lower/1, 337
to_upper/1, 337
words/1, 334

STDLIB
393
words/2, 334
string/1
erl_scan, 120
string/2
erl_scan, 120
string_to_handle/3
qic, 278
strip/1
beam_lib, 59
       string, 334
strip/2
       string, 334
strip/3
       string, 334
strip_files/1
       beam_lib, 59
strip_release/1
       beam_lib, 60
strong_components/1
digraph_utils, 102
sub/3
       regexp, 289
sub_keys/1
       win32reg, 362
sub_string/2
       string, 335
sub_string/3
       string, 336
sub_word/2
       string, 334
sub_word/3
       string, 334
subgraph/3
digraph_utils, 102
sublist/2
       lists, 226
sublist/3
       lists, 226
substitute_aliases/2
       proplists, 264
substitute_negations/2
       proplists, 264
substitution/2
       sofs, 329
substr/2
       string, 333
substr/3
       string, 333
subtract/2
       gb_sets, 164
       lists, 226
       sets, 294
suffix/2
       lists, 227
sum/1
       lists, 227
supervisor
       check_childspecs/1, 343
       delete_child/2, 342
       Module:init/1, 344
       restart_child/2, 342
       start_child/2, 341
       start_link/2, 340
       start_link/3, 340
       terminate_child/2, 341
       which_children/1, 343
supervisor_bridge
       Module:init/1, 346
       Module:terminate/2, 346
       start_link/2, 345
       start_link/3, 345
suspend/1
       sys, 350
suspend/2
       sys, 350
swap_handler/5
       gen_event, 177
swap_sup_handler/5
       gen_event, 178
symdiff/2
       sofs, 329
symmetric_partition/2
       sofs, 330
sync/1
       dets, 84
sync_notify/2
       gen_event, 176
sync_send_all_state_event/2
extract/1, 365
extract/2, 365
list_dir/1, 366
list_dir/2, 366
t/1, 366
table/1, 366
table/2, 366
tt/1, 366
unzip/1, 365
unzip/2, 365
zip/2, 364
zip/3, 364
zip_close/1, 367
zip_get/1, 367
zip_get/2, 367
zip_list_dir/1, 367
zip_open/1, 366
zip_open/2, 367
zip/2
lists, 229
zip, 364
zip/3
zip, 364
zip3/3
lists, 229
zip_close/1
zip, 367
zip_get/1
zip, 367
zip_get/2
zip, 367
zip_list_dir/1
zip, 367
zip_open/1
zip, 366
zip_open/2
zip, 367
zipwith/3
lists, 229
zipwith3/4
lists, 230