STDLIB

version 1.16
Typeset in \LaTeX\ from SGML source using the DocBuilder-0.9.8.5 Document System.
# Contents

1 **STDLIB User’s Guide**  

1.1 The Erlang I/O-protocol  
1.1.1 Protocol basics  
1.1.2 Output requests  
1.1.3 Input Requests  
1.1.4 I/O-server modes  
1.1.5 Multiple I/O requests  
1.1.6 Optional I/O-requests  
1.1.7 Unimplemented request types:  
1.1.8 An annotated and working example io_server:  

1.2 Using Unicode in Erlang  
1.2.1 What Unicode is  
1.2.2 Standard Unicode representation in Erlang  
1.2.3 Basic language support for Unicode  
1.2.4 The interactive shell  
1.2.5 Unicode-aware modules  
1.2.6 Unicode recipes  

STDLIB
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>STDLIB</td>
<td>79</td>
</tr>
<tr>
<td>2.2</td>
<td>array</td>
<td>80</td>
</tr>
<tr>
<td>2.3</td>
<td>base64</td>
<td>86</td>
</tr>
<tr>
<td>2.4</td>
<td>beam_lib</td>
<td>87</td>
</tr>
<tr>
<td>2.5</td>
<td>c</td>
<td>95</td>
</tr>
<tr>
<td>2.6</td>
<td>calendar</td>
<td>99</td>
</tr>
<tr>
<td>2.7</td>
<td>dets</td>
<td>104</td>
</tr>
<tr>
<td>2.8</td>
<td>dict</td>
<td>120</td>
</tr>
<tr>
<td>2.9</td>
<td>digraph</td>
<td>125</td>
</tr>
<tr>
<td>2.10</td>
<td>digraph_utils</td>
<td>132</td>
</tr>
<tr>
<td>2.11</td>
<td>epp</td>
<td>137</td>
</tr>
<tr>
<td>2.12</td>
<td>erl_eval</td>
<td>139</td>
</tr>
<tr>
<td>2.13</td>
<td>erl_expand_records</td>
<td>142</td>
</tr>
<tr>
<td>2.14</td>
<td>erl_id_trans</td>
<td>143</td>
</tr>
<tr>
<td>2.15</td>
<td>erl_internal</td>
<td>144</td>
</tr>
<tr>
<td>2.16</td>
<td>erl_lint</td>
<td>146</td>
</tr>
<tr>
<td>2.17</td>
<td>erl_parse</td>
<td>148</td>
</tr>
<tr>
<td>2.18</td>
<td>erl_pp</td>
<td>151</td>
</tr>
<tr>
<td>2.19</td>
<td>erl_scan</td>
<td>154</td>
</tr>
<tr>
<td>2.20</td>
<td>erl_tar</td>
<td>160</td>
</tr>
<tr>
<td>2.21</td>
<td>ets</td>
<td>166</td>
</tr>
<tr>
<td>2.22</td>
<td>file_sorter</td>
<td>190</td>
</tr>
<tr>
<td>2.23</td>
<td>filelib</td>
<td>195</td>
</tr>
<tr>
<td>2.24</td>
<td>filename</td>
<td>198</td>
</tr>
<tr>
<td>2.25</td>
<td>gb_sets</td>
<td>204</td>
</tr>
<tr>
<td>2.26</td>
<td>gb_trees</td>
<td>210</td>
</tr>
<tr>
<td>2.27</td>
<td>gen_event</td>
<td>215</td>
</tr>
<tr>
<td>2.28</td>
<td>gen_fsm</td>
<td>225</td>
</tr>
<tr>
<td>2.29</td>
<td>gen_server</td>
<td>236</td>
</tr>
<tr>
<td>2.30</td>
<td>io</td>
<td>246</td>
</tr>
<tr>
<td>2.31</td>
<td>io_lib</td>
<td>260</td>
</tr>
<tr>
<td>2.32</td>
<td>lib</td>
<td>264</td>
</tr>
<tr>
<td>2.33</td>
<td>lists</td>
<td>266</td>
</tr>
<tr>
<td>2.34</td>
<td>log-mf.h</td>
<td>282</td>
</tr>
<tr>
<td>2.35</td>
<td>math</td>
<td>283</td>
</tr>
<tr>
<td>2.36</td>
<td>ms_transform</td>
<td>285</td>
</tr>
<tr>
<td>2.37</td>
<td>orddict</td>
<td>296</td>
</tr>
<tr>
<td>2.38</td>
<td>ordsets</td>
<td>301</td>
</tr>
<tr>
<td>2.39</td>
<td>pg</td>
<td>304</td>
</tr>
</tbody>
</table>
2.40 pool .......................................................... 306
2.41 proplib .......................................................... 308
2.42 proplists ...................................................... 313
2.43 qlc ............................................................ 318
2.44 queue .......................................................... 337
2.45 random .......................................................... 343
2.46 re .............................................................. 345
2.47 regexp ........................................................... 392
2.48 sets .............................................................. 397
2.49 shell ............................................................ 400
2.50 shell_default ..................................................... 411
2.51 slave ............................................................ 412
2.52 sofs ............................................................. 415
2.53 string ............................................................ 438
2.54 supervisor ....................................................... 444
2.55 supervisor_bridge .............................................. 451
2.56 sys ............................................................. 454
2.57 timer ............................................................ 461
2.58 unicode ......................................................... 465
2.59 win32reg ......................................................... 469
2.60 zip ............................................................. 473

List of Figures 479
Chapter 1

STDLIB User’s Guide

The Erlang standard library STDLIB.

1.1 The Erlang I/O-protocol

The I/O-protocol in Erlang specifies a way for a client to communicate with an io_server and vice versa. The io_server is a process handling the requests and that performs the requested task on i.e. a device. The client is any Erlang process wishing to read or write data from/to the device.

The common I/O-protocol has been present in OTP since the beginning, but has been fairly undocumented and has also somewhat evolved over the years. In an addendum to Robert Virdings rationale the original I/O-protocol is described. This document describes the current I/O-protocol.

The original I/O-protocol was simple and flexible. Demands for spacial and execution time efficiency has triggered extensions to the protocol over the years, making the protocol larger and somewhat less easy to implement than the original. It can certainly be argued that the current protocol is too complex, but this text describes how it looks today, not how it should have looked.

The basic ideas from the original protocol still holds. The io_server and client communicate with one single, rather simplistic protocol and no server state is ever present in the client. Any io_server can be used together with any client code and client code need not be aware of the actual device the io_server communicates with.

1.1.1 Protocol basics

As described in Roberts paper, servers and clients communicate using io_request/io_reply tuples as follows:

{io_request, From, ReplyAs, Request}
{io_reply, ReplyAs, Reply}

The client sends an io_request to the io_server and the server eventually sends a corresponding reply.

- From is the pid() of the client, the process which the io_server sends the reply to.
・ReplyAs can be any datum and is simply returned in the corresponding io_reply. The io-module in the Erlang standard library simply uses the pid() of the io_server as the ReplyAs datum, but a more complicated client could have several outstanding io-requests to the same server and would then use i.e. a ref() or something else to differentiate among the incoming io_reply’s. The ReplyAs element should be considered opaque by the io_server. Note that the pid() of the server is not explicitly present in the io_reply. The reply can be sent from any process, not necessarily the actual io_server. The ReplyAs element is the only thing that connects one io_request with an io_reply.

・Request and Reply are described below.

When an io_server receives an io_request, it acts upon the actual Request part and eventually sends an io_reply with the corresponding Reply part.

### 1.1.2 Output requests

To output characters on a device, the following Requests exist:

{put_chars, Encoding, Characters}
{put_chars, Encoding, Module, Function, Args}

・Encoding is either ‘latin1’ or ‘unicode’, meaning that the characters are (in case of binaries) encoded as either UTF-8 or iso-latin-1 (pure bytes). A well behaved io_server should also return error if list elements contain integers > 255 when the Encoding is set to latin1. Note that this does not in any way tell how characters should be put on the actual device or how the io_server should handle them. Different io_servers may handle the characters however they want, this simply tells the io_server which format the data is expected to have. In the Module/Function/argument case, the Encoding tells which format the designated function produces. Note that byte-oriented data is simplest sent using latin1 Encoding.

・Characters are the data to be put on the device. If encoding is latin1, this is an iolist(). If encoding is unicode, this is an Erlang standard mixed unicode list (one integer in a list per character, characters in binaries represented as UTF-8).

・Module, Function, Args denotes a function which will be called to produce the data (like io_lib:format), Args is a list of arguments to the function. The function should produce data in the given Encoding. The io_server should call the function as apply(Mod, Func, Args) and will put the returned data on the device as if it was sent in a {put_chars, Encoding, Characters} request. If the function returns anything else than a binary or list or throws an exception, an error should be sent back to the client.

The server replies to the client with an io_reply where the Reply element is one of:

ok
{error, Error}

・Error describes the error to the client, which may do whatever it wants with it. The Erlang io-module typically returns it as is.

For backward compatibility the following Requests should also be handled by an io_server (these messages should not be present after R15B of OTP):

{put_chars, Characters}
{put_chars, Module, Function, Args}

These should behave as {put_chars, latin1, Characters} and {put_chars, latin1, Module, Function, Args} respectively.
1.1.3 Input Requests

To read characters from a device, the following Requests exist:

\( \text{get\_until(\text{Encoding}, \text{Prompt}, \text{Module}, \text{Function}, \text{ExtraArgs})} \)

- **Encoding** denotes how data is to be sent back to the client and what data is sent to the function denoted by Module/Function/Arity. If the function supplied returns data as a list, the data is converted to this encoding. If however the function supplied returns data in some other format, no conversion can be done and it’s up to the client supplied function to return data in a proper way. If Encoding is latin1, lists of integers 0..255 or binaries containing plain bytes are sent back to the client when possible, if Encoding is unicode, lists with integers in the whole unicode range or binaries encoded in UTF-8 are sent to the client. The user supplied function will always see lists of integers, never binaries, but the list may contain numbers > 255 if the Encoding is ‘unicode’.

- **Prompt** is a list of characters (not mixed, no binaries) or an atom() to be output as a prompt for input on the device. The Prompt is often ignored by the io_server and a Prompt set to " should always be ignored (and result in nothing being written to the device).

- **Module, Function, ExtraArgs** denotes a function and arguments to determine when enough data is written. The function should take two additional arguments, the last state, and a list of characters. The function should return one of:
  \( \text{done, Result, RestChars} \)
  \( \text{more, Continuation} \)

  The Result can be any Erlang term, but if it is a list(), the io_server may convert it to a binary() of appropriate format before returning it to the client, if the server is set in binary mode (see below).

  The function will be called with the data the io_server finds on it’s device, returning (done, Result, RestChars) when enough data is read (in which case Result is sent to the client and RestChars are kept in the io_server as a buffer for subsequent input) or (more, Continuation), indicating that more characters are needed to complete the request. The Continuation will be sent as the state in subsequent calls to the function when more characters are available. When no more characters are available, the function shall return (done, eof, Rest). The initial state is the empty list and the data when an end of file is reached on the device is the atom ‘eof’. An emulation of the get\_until request could be (inefficiently) implemented using the following functions:

```erlang
-module(demo).
-export([until_newline/3, get_line/1]).

until_newline(_, eof, _) ->
  \{done, eof, []\};
until_newline(ThisFar, CharList, MyStopCharacter) ->
  case lists:splitwith(fun(X) -> X =/= MyStopCharacter end, CharList) of
    \{L,[]} ->
      \{more, ThisFar++L\};
    \{L2,[MyStopCharacter|Rest]\} ->
      \{done, ThisFar++L2++[MyStopCharacter], Rest\}
  end.

get_line(IoServer) ->
  IoServer ! \{io_request, self(), IoServer, \{get\_until, unicode, '', ?MODULE, until_newline, ['$' \}]),
    receive
      \{io_reply, IoServer, Data} ->
        Data
    end.
Note especially that the last element in the Request tuple ([$]) is appended to the argument list when the function is called. The function should be called like apply(Module, Function, [State, Data | ExtraArgs]) by the io_server.

A defined number of characters is requested using this Request:
{get_chars, Encoding, Prompt, N}
- Encoding and Prompt as for get_until.
- N is the number of characters to be read from the device.

A single line (like in the example above) is requested with this Request:
{get_line, Encoding, Prompt}
- Encoding and prompt as above.

Obviously, get_chars and get_line could be implemented with the get_until request (and indeed was originally), but demands for efficiency has made these additions necessary.

The server replies to the client with an io_reply where the Reply element is one of:
- **Data**
- **eof**
- **error**, Error

- Data is the characters read, in either list or binary form (depending on the io_server mode, see below).
- Error describes the error to the client, which may do whatever it wants with it. The Erlang io-module typically returns it as is.
- eof is returned when input end is reached and no more data is available to the client process.

For backward compatibility the following Requests should also be handled by an io_server (these messages should not be present after R15B of OTP):
{get_until, Prompt, Module, Function, ExtraArgs}
{get_chars, Prompt, N}
{get_line, Prompt}
These should behave as {get_until, latin1, Prompt, Module, Function, ExtraArgs}, {get_chars, latin1, Prompt, N} and {get_line, latin1, Prompt} respectively.

### 1.1.4 I/O-server modes

Demands for efficiency when reading data from an io_server has not only lead to the addition of the get_line and get_chars requests, but has also added the concept of io_server options. No options are mandatory to implement, but all io_servers in the Erlang standard libraries honor the 'binary' option, which allows the Data in the io_reply to be binary instead of in list form when possible. If the data is sent as a binary, Unicode data will be sent in the standard Erlang unicode format, i.e. UTF-8 (note that the function in get_until still gets list data regardless of the io_server mode).

Note that i.e. the get_until request allows for a function with the data specified as always being a list. Also the return value data from such a function can be of any type (as is indeed the case when an io:fread request is sent to an io_server). The client has to be prepared for data received as answers to those requests to be in a variety of forms, but the server should convert the results to binaries whenever possible (i.e. when the function supplied to get_until actually returns a list). The example shown later in this text does just that.
1.1: The Erlang I/O-protocol

An I/O-server in binary mode will affect the data sent to the client, so that it has to be able to handle binary data. For convenience, it is possible to set and retrieve the modes of an io_server using the following I/O-requests:

```
(setopts, Opts)
```

- `Opts` is a list of options in the format recognized by proplists (and of course by the io_server itself).

As an example, the io_server for the interactive shell (in group.erl) understands the following options:

```
{binary, bool()} (or 'binary'/’list')
{echo, bool()}
{expand_fun, fun()}
{encoding, 'unicode'/’latin1'} (or 'unicode'/'latin1')
```

- of which the ‘binary’ and ‘encoding’ options are common for all io_servers in OTP, while ‘echo’ and ‘expand’ is valid only for this io_server. It’s worth noting that the ‘unicode’ option notifies how characters are actually put on the physical device, i.e. if the terminal per se is unicode aware, it does not affect how characters are sent in the I/O-protocol, where each request contains encoding information for the provided or returned data.

The server should send one of the following as Reply:

```
ok
{error, Error}
```

An error (preferably enotsup) is to be expected if the option is not supported by the io_server (like if an ‘echo’ option is sent in a setopt Request to a plain file).

To retrieve options, this message is used:

```
getopts
```

The ‘getopts’ message requests a complete list of all options supported by the io_server as well as their current values.

The server replies:

```
OptList
```

```
{error, Error}
```

- `OptList` is a list of tuples `{Option, Value}` where `Option` is always an atom.

### 1.1.5 Multiple I/O requests

The Request element can in itself contain several Requests by using the following format:

```
(requests, Requests)
```

- `Requests` is a list of valid Request tuples for the protocol, they shall be executed in the order in which they appear in the list and the execution should continue until one of the requests result in an error or the list is consumed. The result of the last request is sent back to the client.

The server can for a list of requests send any of the valid results in the reply:

```
ok
{ok, Data}
{ok, O ptions}
{error, Error}
```

- depending on the actual requests in the list.
1.1.6 Optional I/O-requests

The following I/O request is optional to implement and a client should be prepared for an error return:
{get_geometry, Geometry}
- Geometry is either the atom 'rows' or the atom 'columns'.

The server should send the Reply as:
{ok, N}
{error, Error}
- N is the number of character rows or columns the device has, if applicable to the device the io_server handles, otherwise {error, enotsup} is a good answer.

1.1.7 Unimplemented request types:

If an io_server encounters a request it does not recognize (i.e. the io_request tuple is in the expected format, but the actual Request is unknown), the server should send a valid reply with the error tuple:
{error, request}
This makes it possible to extend the protocol with optional messages and for the clients to be somewhat backwards compatible.

1.1.8 An annotated and working example io_server:

An io_server is any process capable of handling the protocol. There is no generic io_server behavior, but could well be. The framework is simple enough, a process handling incoming requests, usually both io_requests and other device-specific requests (for i.e. positioning, closing etc.).

Our example io_server stores characters in an ets table, making up a fairly crude ram-file (it is probably not useful, but working).

The module begins with the usual directives, a function to start the server and a main loop handling the requests:

-module(ets_io_server).
-export([start_link/0, init/0, loop/1, until_newline/3, until_enough/3]).
-define(CHARS_PER_REC, 10).
-record(state, {
    table, %
    position, % absolute
    mode % binary | list
}).

start_link() ->
    spawn_link(?MODULE, init, []).
init() ->
    Table = ets:new(noname,[ordered_set]),
    ?MODULE:loop(#state{table = Table, position = 0, mode=list}).
1.1: The Erlang I/O-protocol

loop(State) ->
  receive
    {io_request, From, ReplyAs, Request} ->
      case request(Request, State) of
        {Tag, Reply, NewState} when Tag == ok; Tag == error ->
          reply(From, ReplyAs, Reply),
          ?MODULE:loop(NewState);
        {stop, Reply, _NewState} ->
          reply(From, ReplyAs, Reply),
          exit(Reply)
      end;
      %% Private message
      {From, rewind} ->
        From ! {self(), ok},
        ?MODULE:loop(State#state{position = 0});
      Unknown ->
        ?MODULE:loop(State)
    end.
end.

The main loop receives messages from the client (which might be using the io-module to send
requests). For each request the function request/2 is called and a reply is eventually sent using the
reply/3 function.

The "private" message {From, rewind} results in the current position in the pseudo-file to be reset to 0
(the beginning of the "file"). This is a typical example of device-specific messages not being part of the
I/O-protocol. It is usually a bad idea to embed such private messages in io_request tuples, as that might
be confusing to the reader.

Let's look at the reply function first...

reply(From, ReplyAs, Reply) ->
  From ! {io_reply, ReplyAs, Reply}.

Simple enough, it sends the io_reply tuple back to the client, providing the ReplyAs element received in
the request along with the result of the request, as described above.

Now look at the different requests we need to handle. First the requests for writing characters:

request({put_chars, Encoding, Chars}, State) ->
  put_chars(unicode:characters_to_list(Chars, Encoding), State);
request({put_chars, Encoding, Module, Function, Args}, State) ->
  try
    request({put_chars, Encoding, apply(Module, Function, Args)}, State)
  catch
    _err ->
      {error, {error, Function}, State}
  end;

The Encoding tells us how the characters in the message are represented. We want to store the
characters as lists in the ets-table, so we convert them to lists using the unicode:characters_to_list/2
function. The conversion function conveniently accepts the encoding types unicode or latin1, so we can
use the Encoding parameter directly.
When Module, Function and Arguments are provided, we simply apply it and do the same thing with the result as if the data was provided directly.

Let's handle the requests for retrieving data too:

```erlang
request({get_until, Encoding, _Prompt, M, F, As}, State) ->
    get_until(Encoding, M, F, As, State);
request({get_chars, Encoding, _Prompt, N}, State) ->
    %% To simplify the code, get_chars is implemented using get_until
    get_until(Encoding, ?MODULE, until_enough, [N], State);
request({get_line, Encoding, _Prompt}, State) ->
    %% To simplify the code, get_line is implemented using get_until
    get_until(Encoding, ?MODULE, until_newline, ['$\n'], State);
```

Here we have cheated a little by more or less only implementing get_until and using internal helpers to implement get_chars and get_line. In production code, this might be too inefficient, but that of course depends on the frequency of the different requests. Before we start actually implementing the functions put_chars/2 and get_until/5, lets look into the few remaining requests:

```erlang
request({get_geomentry,}, State) ->
    {error, {error, enotsup}, State};
request({setopts, Opts}, State) ->
    setopts(Opts, State);
request(getopts, State) ->
    getopts(State);
request({requests, Reqs}, State) ->
    multi_request(Reqs, {ok, ok, State});
```

The get_geometry request has no meaning for this io_server, so the reply will be `{error, enotsup}`. The only option we handle is the binary/list option, which is done in separate functions.

The multi-request tag (requests) is handled in a separate loop function applying the requests in the list one after another, returning the last result.

What's left is to handle backward compatibility and the file-module (which uses the old requests until backward compatibility with pre-R13 nodes is no longer needed). Note that the io_server will not work with a simple file:write if these are not added:

```erlang
request({put_chars,Chars}, State) ->
    request({put_chars,latin1,Chars}, State);
request({put_chars,M,F,As}, State) ->
    request({put_chars,latin1,M,F,As}, State);
request({get_chars,Prompt,N}, State) ->
    request({get_chars,latin1,Prompt,N}, State);
request({get_line,Prompt}, State) ->
    request({get_line,latin1,Prompt}, State);
request({get_until, Prompt,M,F,As}, State) ->
    request({get_until,latin1,Prompt,M,F,As}, State);
```

Ok, what's left now is to return `{error, request}` if the request is not recognized:

```erlang
request(_Other, State) ->
    {error, {error, request}, State}.
```
Let’s move further and actually handle the different requests, first the fairly generic multi-request type:

```erlang
multi_request([R|Rs], {ok, _Res, State}) ->
multi_request(Rs, request(R, State));
multi_request([], Error) ->
  Error;
multi_request([], Result) ->
  Result.
```

We loop through the requests one at the time, stopping when we either encounter an error or the list is exhausted. The last return value is sent back to the client (it’s first returned to the main loop and then sent back by the function `io_reply`).

The getopt and setopt requests is also simple to handle, we just change or read our state record:

```erlang
setopts(Opts0, State) ->
  Opts = proplists:unfold(
    proplists:substitute_negations(
      [[list, binary] | Opts0]),
    case check_valid_opts(Opts) of
      true ->
        case proplists:get_value(binary, Opts) of
          true ->
            {ok, ok, State#state{mode=binary}};
          false ->
            {ok, ok, State}
        end;
      false ->
        {error, {error, enotsup}, State}
    end.
  end.
check_valid_opts([]) ->
  true;
check_valid_opts([{binary, Bool}|T]) when is_boolean(Bool) ->
  check_valid_opts(T);
check_valid_opts(_) ->
  false.
getopts(#state{mode=M} = S) ->
  {ok, [{binary, case M of
    binary ->
      true;
    _ ->
      false
  end}], S}.
```

As a convention, all io_servers handle both `{setopts, [binary]}`, `{setopts, [list]}` and `{setopts, [{binary, bool()}]}`, hence the trick with `proplists:substitute_negations/2` and `proplists:unfold/1`. If invalid options are sent to us, we send `{error, enotsup}` back to the client.
Chapter 1: STDLIB User's Guide

The getopt request should return a list of (Option, Value) tuples, which has the twofold function of providing both the current values and the available options of this io server. We have only one option, and hence return that.

So far our io_server has been fairly generic (except for the rewind request handled in the main loop and the creation of an ets table). Most io_servers contain code similar to what's above.

To make the example runnable, we now start implementing the actual reading and writing of the data to/from the ets-table. First the put_chars function:

```
put_chars(Chars, #state{table = T, position = P} = State) ->
  R = P div #CHARS_PER_REC,
  C = P rem #CHARS_PER_REC,
  [ apply_update(T,U) || U <- split_data(Chars, R, C) ],
  {ok, ok, State#state{position = (P + length(Chars))}}.
```

We already have the data as (Unicode) lists and therefore just split the list in runs of a predefined size and put each run in the table at the current position (and forward). The functions split_data/3 and apply_update/2 are implemented below.

Now we want to read data from the table. The get_until function reads data and applies the function until it says it's done. The result is sent back to the client:

```
get_until(Encoding, Mod, Func, As, #state{position = P, mode = M, table = T} = State) ->
  case get_loop(Mod,Func,As,T,P,[]) of
    {done,Data,_,NewP} when is_binary(Data); is_list(Data) ->
      if
        M == binary ->
          {ok, unicode:characters_to_binary(Data,unicode,Encoding),
           State#state{position = NewP}};
        true ->
          case check(Encoding, unicode:characters_to_list(Data, unicode)) of
            {error, _} = E ->
              {error, E, State};
            List ->
              {ok, List, State#state{position = NewP}}
          end
      end;
    {done,Data,_,NewP} ->
      {ok, Data, State#state{position = NewP}};
    Error ->
      {error, Error, State}
  end.
```

```
get_loop(M,F,A,T,P,C) ->
  {NewP,L} = get(P,T),
  case catch apply(M,F,[C,L|A]) of
    {done, List, Rest} ->
      {done, List, [], NewP - length(Rest)};
    {more, NewC} ->
```

STDLIB
get_loop(M,F,A,T,NewP,NewC);  
   _ ->  
      {error,F}  
   end.

Here we also handle the mode (binary or list) that can be set by the setopts request. By default, all OTP io_servers send data back to the client as lists, but switching mode to binary might increase efficiency if the server handles it in an appropriate way. The implementation of get_until is hard to get efficient as the supplied function is defined to take lists as arguments, but get_chars and get_line can be optimized for binary mode. This example does not optimize anything however. It is important though that the returned data is of the right type depending on the options set, so we convert the lists to binaries in the correct encoding if possible before returning. The function supplied in the get_until request may, as it's final result return anything, so only functions actually returning lists can get them converted to binaries. If the request contained the encoding tag unicode, the lists can contain all unicode codepoints and the binaries should be in UTF-8, if the encoding tag was latin1, the client should only get characters in the range 0..255. The function check/2 takes care of not returning arbitrary unicode codepoints in lists if the encoding was given as latin1. If the function did not return a list, the check cannot be performed and the result will be that of the supplied function untouched.

Now we are more or less done. We implement the utility functions below to actually manipulate the table:

check(unicode, List) ->  
   List;  
check(latin1, List) ->  
   try  
      [ throw(not_ascii) || X <- List,  
          X > 255 ],  
   List  
   catch  
      throw:_ ->  
         {error,{cannot_convert, unicode, latin1}}  
   end.

The function check takes care of providing an error tuple if unicode codepoints above 255 is to be returned if the client requested latin1.

The two functions until_newline/3 and until_enough/3 are helpers used together with the get_until function to implement get_chars and get_line (inefficiently):

until_newline([],eof,_MyStopCharacter) ->  
   {done,eof,[]};  
until_newline(ThisFar,eof,_MyStopCharacter) ->  
   {done,ThisFar,[]};  
until_newline(ThisFar,CharList,MyStopCharacter) ->  
   case lists:splitwith(fun(X) -> X =/= MyStopCharacter end, CharList) of  
      {L,[]} ->  
         {more,ThisFar++L};  
      {L2,[MyStopCharacter|Rest]} ->  
         {done,ThisFar++L2++[MyStopCharacter],Rest}  
   end.

until_enough([],eof,_N) ->  
   {done,eof,[]};
Chapter 1: STDLIB User’s Guide

until_enough(ThisFar, eof, _N) ->
    {done, ThisFar, []};
until_enough(ThisFar, CharList, N)
    when length(ThisFar) + length(CharList) >= N ->
        {Res, Rest} = my_split(N, ThisFar ++ CharList, []),
        {done, Res, Rest};
until_enough(ThisFar, CharList, _N) ->
    {more, ThisFar++CharList}.

As can be seen, the functions above are just the type of functions that should be provided in get_until
requests.

Now we only need to read and write the table in an appropriate way to complete the server:

get(P, Tab) ->
    R = P div ?CHARS_PER_REC,
    C = P rem ?CHARS_PER_REC,
    case ets:lookup(Tab,R) of
        [] ->
            {P, eof};
        [[R, List]] ->
            case my_split(C, List, []) of
                _, [] ->
                    {P+length(List), eof};
                _, Data ->
                    {P+length(Data), Data}
            end
    end.

my_split(0, Left, Acc) ->
    {lists:reverse(Acc), Left};
my_split(_, [], Acc) ->
    {lists:reverse(Acc), []};
my_split(N, [H|T], Acc) ->
    my_split(N-1, T, [H|Acc]).

split_data([], _, _) ->
    [];
split_data(Chars, Row, Col) ->
    {This, Left} = my_split(?CHARS_PER_REC - Col, Chars, []),
    [ {Row, Col, This} | split_data(Left, Row + 1, 0) ].

apply_update(Table, {Row, Col, List}) ->
    case ets:lookup(Table,Row) of
        [] ->
            ets:insert(Table, {Row, lists:duplicate(Col,0) ++ List});
        [Row, OldData] ->
            {Part1, _} = my_split(Col, OldData, []),
            {_, Part2} = my_split(Col+length(List), OldData, []),
            ets:insert(Table, {Row, Part1 ++ List ++ Part2})
    end.

The table is read or written in chunks of ?CHARS_PER_REC, overwriting when necessary. The
implementation is obviously not efficient, it is just working.
This concludes the example. It is fully runnable and you can read or write to the io_server by using i.e. the io_module or even the file module. It’s as simple as that to implement a fully fledged io_server in Erlang.

1.2 Using Unicode in Erlang

Implementing support for Unicode character sets is an ongoing process. The Erlang Enhancement Proposal (EEP) 10 outlines the basics of Unicode support and also specifies a default encoding in binaries that all Unicode-aware modules should handle in the future.

The functionality described in EEP10 is implemented in Erlang/OTP as of R13A, but that’s by no means the end of it. More functionality will be needed in the future and more OTP-libraries might need updating to cope with Unicode data. One example of future development is obvious when reading this manual, our documentation format is limited to the ISO-latin-1 character range, why no Unicode characters beyond that range will occur in this document.

This guide outlines the current Unicode support and gives a couple of recipes for working with Unicode data.

1.2.1 What Unicode is

Unicode is a standard defining codepoints (numbers) for all known, living or dead, scripts. In principle, every known symbol used in any language has a Unicode codepoint.

Unicode codepoints are defined and published by the Unicode Consortium, which is a non profit organization.

Support for Unicode is increasing throughout the world of computing, as the benefits of one common character set are overwhelming when programs are used in a global environment.

Along with the base of the standard, the codepoints for all the scripts, there are a couple of encoding standards available. Different operating systems and tools support different encodings. For example Linux and MacOS X has chosen the UTF-8 encoding, which is backwards compatible with 7-bit ASCII and therefore affects programs written in plain English the least. Windows on the other hand supports a limited version of UTF-16, namely all the code planes where the characters can be stored in one single 16-bit entity, which includes most living languages.

The most widely spread encodings are:

**UTF-8** Each character is stored in one to four bytes depending on codepoint. The encoding is backwards compatible with 7-bit ASCII as all 7-bit characters are stored in one single byte as is. The characters beyond codepoint 127 are stored in more bytes, letting the most significant bit in the first character indicate a multi-byte character. For details on the encoding, the RFC is publicly available.

**UTF-16** This encoding has many similarities to UTF-8, but the basic unit is a 16-bit number. This means that all characters occupy at least two bytes, some high numbers even four bytes. Some programs and operating systems claiming to use UTF-16 only allows for characters that can be stored in one 16-bit entity, which is usually sufficient to handle living languages. As the basic unit is more than one byte, byte-order issues occur, why UTF-16 exists in both a big-endian and little-endian variant.

**UTF-32** The most straight forward representation, each character is stored in one single 32-bit number. There is no need for escapes or any variable amount of entities for one character, all Unicode codepoints can be stored in one single 32-bit entity. As with UTF-16, there are byte-order issues, UTF-32 can be both big- and little-endian.
**UCS-4** Basically the same as UTF-32, but without some Unicode semantics, defined by IEEE and has little use as a separate encoding standard. For all normal (and possibly abnormal) usages, UTF-32 and UCS-4 are interchangeable.

Certain ranges of characters are left unused and certain ranges are even deemed invalid. The most notable invalid range is U+D800 - U+DFFF, as the UTF-16 encoding does not allow for encoding of these numbers. It can be speculated that the UTF-16 encoding standard was, from the beginning, expected to be able to hold all Unicode characters in one 16-bit entity, but then had to be extended, leaving a whole in the Unicode range to cope with backward compatibility.

Additionally, the codepoint U+FEFF is used for byte order marks (BOM’s) and use of that character is not encouraged in other contexts than that. It actually is valid though, as the character “ZWNBS” (Zero Width Non Breaking Space). BOM’s are used to identify encodings and byte order for programs where such parameters are not known in advance. Byte order marks are more seldom used than one could expect, put their use is becoming more widely spread as they provide the means for programs to make educated guesses about the Unicode format of a certain file.

### 1.2.2 Standard Unicode representation in Erlang

In Erlang, strings are actually lists of integers. A string is defined to be encoded in the ISO-latin-1 (ISO8859-1) character set, which is, codepoint by codepoint, a sub-range of the Unicode character set.

The standard list encoding for strings is therefore easily extendable to cope with the whole Unicode range: A Unicode string in Erlang is simply a list containing integers, each integer being a valid Unicode codepoint and representing one character in the Unicode character set.

Regular Erlang strings in ISO-latin-1 are a subset of there Unicode strings.

Binaries on the other hand are more troublesome. For performance reasons, programs often store textual data in binaries instead of lists, mainly because they are more compact (one byte per character instead of two words per character, as is the case with lists). Using erlang:list_to_binary/1, an regular Erlang string can be converted into a binary, effectively using the ISO-latin-1 encoding in the binary - one byte per character. This is very convenient for those regular Erlang strings, but cannot be done for Unicode lists.

As the UTF-8 encoding is widely spread and provides the most compact storage, it is selected as the standard encoding of Unicode characters in binaries for Erlang.

The standard binary encoding is used whenever a library function in Erlang should cope with Unicode data in binaries, but is of course not enforced when communicating externally. Functions and bit-syntax exist to encode and decode both UTF-8, UTF-16 and UTF-32 in binaries. Library functions dealing with binaries and Unicode in general, however, only deal with the default encoding.

Character data may be combined from several sources, sometimes available in a mix of strings and binaries. Erlang has for long had the concept of iodata or iolists, where binaries and lists can be combined to represent a sequence of bytes. In the same way, the Unicode aware modules often allow for combinations of binaries and lists where the binaries have characters encoded in UTF-8 and the lists contain such binaries or numbers representing Unicode codepoints:

```
unicode_binary() = binary() with characters encoded in UTF-8 coding standard
unicode_char() = integer() representing valid unicode codepoint

chardata() = charlist() | unicode_binary()

charlist() = [unicode_char() | unicode_binary() | charlist()]
```

A unicode_binary is allowed as the tail of the list.
The module `unicode` in `stdlib` even supports similar mixes with binaries containing other encodings than UTF-8, but that is a special case to allow for conversions to and from external data:

```erlang
external_unicode_binary() = binary() with characters coded in a user specified Unicode encoding ot
external_chardata() = external_charlist() | external_unicode_binary()
external_charlist() = [unicode_char() | external_unicode_binary() | external_charlist()]
```

an external_unicode_binary is allowed as the tail of the list

### 1.2.3 Basic language support for Unicode

First of all, Erlang is still defined to be written in the ISO-latin-1 character set. Functions have to be named in that character set, atoms are restricted to ISO-latin-1 and regular strings are still lists of characters 0..255 in the ISO-latin-1 encoding. This has not (yet) changed, but the language has been slightly extended to cope with Unicode characters and encodings.

#### Bit-syntax

The bit-syntax contains types for coping with binary data in the three main encodings. The types are named `utf8`, `utf16` and `utf32` respectively. The `utf16` and `utf32` types can be in a big- or little-endian variant:

```erlang
<<Ch/utf8,_/binary>> = Bin1,
<<Ch/utf16-little,_/binary>> = Bin2,
Bin3 = <<$H/utf32-little, $e/utf32-little, $l/utf32-little, $l/utf32-little, $o/utf32-little>>,
```

For convenience, literal strings can be encoded with a Unicode encoding in binaries using the following (or similar) syntax:

```erlang
Bin4 = <<"Hello"/utf16>>,
```

#### String- and character-literals

**Warning:**

The literal syntax described here may be subject to change in R13B, it has not yet passed the usual process for language changes approval.

It is convenient to be able to write a list of Unicode characters in the string syntax. However, the language specifies strings as being in the ISO-latin-1 character set which the compiler tool chain as well as many other tools expect.

Also the source code is (for now) still expected to be written using the ISO-latin-1 character set, why Unicode characters beyond that range cannot be entered in string literals.

To make it easier to enter Unicode characters in the shell, it allows strings with Unicode characters on input, immediately converting them to regular lists of integers. They will, by the evaluator etc be viewed as if they were input using the regular list syntax, which is - in the end - how the language actually treats them. They will in the same way not be output as strings by `io:write/2` or

---

**STDLIB**

15
io:format/3 unless the format string supplied to io:format uses the Unicode translation modifier (which we will talk about later).

For source code, there is an extension to the \OOO (backslash followed by three octal numbers) and \xHH (backslash followed by 'x', followed by two hexadecimal characters) syntax, namely \x{H ...} (a backslash followed by an ‘x’, followed by left curly bracket, any number of hexadecimal digits and a terminating right curly bracket). This allows for entering characters of any codepoint literally in a string. The string is immediately converted into a list by the scanner however, which is obvious when calling it directly:

1> erl:scan:string("\"X\".").
{ok,[{string,1,\"X\"},{dot,1}],1}
2> erl:scan:string("\"\x{400}\".").
{ok,[{"\",1},{integer,1,1024},{\",1},{dot,1}],1}

Character literals, or rather integers representing Unicode codepoints can be expressed in a similar way using $\x{H ...}:

4> \x{400}.
1024

This also is a translation by the scanner:

5> erl:scan:string("$Y.").
{ok,[{char,1,89},{dot,1}],1}
6> erl:scan:string("$\x{400}.").
{ok,[{integer,1,1024},{dot,1}],1}

In the shell, if using a Unicode input device, ‘$’ can be followed directly by a Unicode character producing an integer. In the following example, let’s imagine the character ‘c’ is actually a Cyrillic ‘s’ (looking fairly similar):

7> $c.
1089

The literal syntax allowing Unicode characters is to be viewed as “syntactic sugar”, but is, as such, fairly useful.

1.2.4 The interactive shell

The interactive Erlang shell, when started towards a terminal or started using the werl command on windows, can support Unicode input and output.

On Windows, proper operation requires that a suitable font is installed and selected for the Erlang application to use. If no suitable font is available on your system, try installing the DejaVu fonts (dejavu-fonts.org), which are freely available and then select that font in the Erlang shell application.

On Unix-like operating systems, the terminal should be able to handle UTF-8 on input and output (modern versions of X Term, KDE konsole and the Gnome terminal do for example) and your locale settings have to be proper. As an example, my LANG environment variable is set as this:

$ echo $LANG
en_JIS.UTF-8
Actually, most systems handle the LC_CTYPE variable before LANG, so if that is set, it has to be set to UTF-8:

$ echo $LC_CTYPE
en_US.UTF-8

The LANG or LC_CTYPE setting should be consistent with what the terminal is capable of; there is no portable way for Erlang to ask the actual terminal about its UTF-8 capacity, we have to rely on the language and character type settings.

To investigate what Erlang thinks about the terminal, the io:getopts() call can be used when the shell is started:

$ LC_CTYPE=en US.ISO-8859-1 erl
Erlang R13A (erts-5.7) [source] [64-bit] [smp:4:4] [rq:4] [async-threads:0] [kernel-poll:false]

Eshell V5.7  (abort with ^G)
1> lists:keyfind(encoding,1,io:getopts()).
   {encoding,lat1}
2> q()
ok

$ LC_CTYPE=en US.UTF-8 erl
Erlang R13A (erts-5.7) [source] [64-bit] [smp:4:4] [rq:4] [async-threads:0] [kernel-poll:false]

Eshell V5.7  (abort with ^G)
1> lists:keyfind(encoding,1,io:getopts()).
   {encoding,unicode}
2>

When (finally?) everything is in order with the locale settings, fonts and the terminal emulator, you probably also have discovered a way to input characters in the script you desire. For testing, the simplest way is to add some keyboard mappings for other languages, usually done with some applet in your desktop environment. In my KDE environment, I start the KDE Control Center (Personal Settings), select “Regional and Accessibility” and then “Keyboard Layout”. On Windows XP, I start Control Panel -> Regional and Language Options, select the Language tab and click the Details... button in the square named “Text services and input Languages”. Your environment probably provides similar means of changing the keyboard layout. Make sure you have a way to easily switch back and forth between keyboards if you are not used to this, entering commands using a Cyrillic character set is, as an example, not easily done in the Erlang shell.

Now you are set up for some Unicode input and output. The simplest thing to do is of course to enter a string in the shell:
While strings can be input as Unicode characters, the language elements are still limited to the ISO-latin-1 character set. Only character constants and strings are allowed to be beyond that range:

```
$ erl
Erlang R13A (erts-5.7) [source] [64-bit] [smp:4:4] [rq:4] [async-threads:0] [kernel-poll:false]

Eshell V5.7 (abort with ^G)
1> lists:keyfind(encoding,1,io:getopts()).
   {encoding,unicode}
2> "улица".
   [1091,1085,1080,1094,1086,1076,1077]
3> io:format(""ts"n",[v(2)])).
   unicode
   ok
4> []
```

Figure 1.1: Cyrillic characters in an Erlang shell

```
Eshell V5.7 (abort with ^G)
1> $£.
   958
2> 8uuycEe.
   * 1: illegal character
2>
```

Figure 1.2: Unicode characters in allowed and disallowed context

1.2.5 Unicode-aware modules

Most of the modules in Erlang/OTP are of course Unicode-unaware in the sense that they have no notion of Unicode and really shouldn’t have. Typically they handle non-textual or byte-oriented data (like gen_tcp etc).

Modules that actually handle textual data (like io_lib, string etc) are sometimes subject to conversion or extension to be able to handle Unicode characters.

Fortunately, most textual data has been stored in lists and range checking has been sparse, why modules like string works well for Unicode lists with little need for conversion or extension.

Some modules are however changed to be explicitly Unicode-aware. These modules include:

- `unicode` The module [unicode] is obviously Unicode-aware. It contains functions for conversion between different Unicode formats as well as some utilities for identifying byte order marks. Few programs handling Unicode data will survive without this module.

- `io` The [io] module has been extended along with the actual I/O-protocol to handle Unicode data. This means that several functions require binaries to be in UTF-8 and there are modifiers to formatting control sequences to allow for outputting of Unicode strings.
1.2: Using Unicode in Erlang

file, group and user I/O-servers throughout the system are able both to handle Unicode data and has options for converting data upon actual output or input to/from the device. As shown earlier, the [shell] has support for Unicode terminals and the [file] module allows for translation to and from various Unicode formats on disk.

The actual reading and writing of files with Unicode data is however not best done with the file module as its interface is byte oriented. A file opened with a Unicode encoding (like UTF-8), is then best read or written using the [io] module.

The [re] module allows for matching Unicode strings as a special option. As the library is actually centered on matching in binaries, the Unicode support is UTF-8-centered.

wx The [wx] graphical library has extensive support for Unicode text.

The module [string] works perfect for Unicode strings as well as for ISO-latin-1 strings with the exception of the language-dependent [to_upper] and [to_lower] functions, which are only correct for the ISO-latin-1 character set. Actually they can never function correctly for Unicode characters in their current form, there are language and locale issues as well as multi-character mappings to consider when conversion text between cases. Converting case in an international environment is a big subject not yet addressed in OTP.

1.2.6 Unicode recipes

When starting with Unicode, one often stumbles over some common issues. I try to outline some methods of dealing with Unicode data in this section.

Byte order marks

A common method of identifying encoding in text-files is to put a byte order mark (BOM) first in the file. The BOM is the codepoint 16#FEFF encoded in the same way as the rest of the file. If such a file is to be read, the first few bytes (depending on encoding) is not part of the actual text. This code outlines how to open a file which is believed to have a BOM and set the files encoding and position for further sequential reading (preferably using the [io] module). Note that error handling is omitted from the code:

```erlang
open_bom_file_for_reading(File) ->
    {ok,F} = file:open(File,[read,binary]),
    {ok,Bin} = file:read(F,4),
    {Type,Bytes} = unicode:bom_to_encoding(Bin),
    file:position(F,Bytes),
    io:setopts(F,[[encoding,Type]]),
    {ok,F}.
```

The `unicode:bom_to_encoding/1` function identifies the encoding from a binary of at least four bytes. It returns, along with an term suitable for setting the encoding of the file, the actual length of the BOM, so that the file position can be set accordingly. Note that `file:position` always works on byte-offsets, so that the actual byte-length of the BOM is needed.

To open a file for writing and putting the BOM first is even simpler:

```erlang
open_bom_file_for_writing(File,Encoding) ->
    {ok,F} = file:open(File,[write,binary]),
    ok = file:write(File,unicode:encoding_to_bom(Encoding)),
    io:setopts(F,[[encoding,Encoding]]),
    {ok,F}.
```
In both cases the file is then best processed using the io module, as the functions in io can handle codepoints beyond the ISO-latin-1 range.

Formatted input and output

When reading and writing to Unicode-aware entities, like the User or a file opened for Unicode translation, you will probably want to format text strings using the functions in [io] or [io_lib]. For backward compatibility reasons, these functions don’t accept just any list as a string, but require a special “translation modifier” when working with Unicode texts. The modifier is “t”. When applied to the “s” control character in a formatting string, it accepts all Unicode codepoints and expect binaries to be in UTF-8:

```
1> io:format(""-t"n",[[<<"utf8">>]]).
ok
2> io:format(""-s"n",[[<<"utf8">>]]).
ok
```

Obviously the second io:format gives undesired output because the UTF-8 binary is not in latin1. Because ISO-latin-1 is still the defined character set of Erlang, the non prefixed “s” control character expects ISO-latin-1 in binaries as well as lists.

As long as the data is always lists, the “t” modifier can be used for any string, but when binary data is involved, care must be taken to make the tight choice of formatting characters.

The function format in io_lib behaves similarly. This function is defined to return a deep list of characters and the output could easily be converted to binary data for outputting on a device of any kind by a simple erlang:list_to_binary. When the translation modifier is used, the list can however contain characters that cannot be stored in one byte. The call to erlang:list_to_binary will in that case fail. However, if the io_server you want to communicate with is Unicode-aware, the list returned can still be used directly:

```
Eshell V5.7 (abort with ^G)
1> io_lib:format(""-s"n",["8yεοέ"]).
[[952,957,953,968,959,948,949],"\n"]
2> io:put_chars(io_lib:format(""-s"n",["8yεοέ"])).
8yεοέ
ok
```

Figure 1.3: io_lib:format with Unicode translation

The Unicode string is returned as a Unicode list, why the return value of io_lib:format no longer qualifies as a regular Erlang string (the function io_lib:deep_char_list will, as an example, return false). The Unicode list is however valid input to the [io:put_chars] function, so data can be output on any Unicode capable device anyway. If the device is a terminal, characters will be output in the \x{H ...} format if encoding is latin1 otherwise in UTF-8 (for the non-interactive terminal - “oldshell” or “noshell”) or whatever is suitable to show the character properly (for an interactive terminal - the regular shell). The bottom line is that you can always send Unicode data to the standard io device. Files will however only accept Unicode codepoints beyond ISO-latin-1 if encoding is set to something else than latin1.
Heuristic identification of UTF-8

While it’s strongly encouraged that the actual encoding of characters in binary data is known prior to processing, that is not always possible. On a typical Linux system, there is a mix of UTF-8 and ISO-latin-1 text files and there are seldom any BOM’s in the files to identify them.

UTF-8 is designed in such a way that ISO-latin-1 characters with numbers beyond the 7-bit ASCII range are seldom considered valid when decoded as UTF-8. Therefore one can usually use heuristics to determine if a file is in UTF-8 or if it is encoded in ISO-latin-1 (one byte per character) encoding. The unicode module can be used to determine if data can be interpreted as UTF-8:

```erlang
heuristic_encoding_bin(Bin) when is_binary(Bin) ->
    case unicode:characters_to_binary(Bin,utf8,utf8) of
       Bin -> utf8;
       _ -> latin1
    end.
```

If one does not have a complete binary of the file content, one could instead chunk through the file and check part by part. The return-tuple `{incomplete,Decoded,Rest}` from `unicode:characters_to_binary/[1,2,3]` comes in handy. The incomplete rest from one chunk of data read from the file is prepended to the next chunk and we therefore circumvent the problem of character boundaries when reading chunks of bytes in UTF-8 encoding:

```erlang
heuristic_encoding_file(FileName) ->
    {ok,F} = file:open(FileName,[read,binary]),
    loop_through_file(F,<<>>,file:read(F,1024)).

loop_through_file(_,<<>>,eof) -> utf8;
loop_through_file(_,_,eof) -> latin1;
loop_through_file(F,Acc,{ok,Bin}) when is_binary(Bin) ->
    case unicode:characters_to_binary([Acc,Bin]) of
       {error,_,_} -> latin1;
       {incomplete,_,Rest} -> loop_through_file(F,Rest,file:read(F,1024));
       Res when is_binary(Res) -> loop_through_file(F,<<>>,file:read(F,1024))
    end.
```

Another option is to try to read the whole file in utf8 encoding and see if it fails. Here we need to read the file using `io:get_chars/3`, as we have to succeed in reading characters with a codepoint over 255:

```erlang
heuristic_encoding_file2(FileName) ->
    {ok,F} = file:open(FileName,[read,binary,{encoding,utf8}]),
    loop_through_file2(F,io:get_chars(F,'',1024)).

loop_through_file2(_,eof) -> utf8;
loop_through_file2(_,{error,_,Err}) ->
```

STDLIB

21
latin1;
loop_through_file2(F,Bin) when is_binary(Bin) ->
   loop_through_file2(F,io:get_chars(F, '',1024)).
Short Summaries

- Application **STDLIB** [page 79] – The STDLIB Application
- Erlang Module **array** [page 80] – Functional, extendible arrays.
- Erlang Module **base64** [page 86] – Implements base 64 encode and decode, see RFC 2045.
- Erlang Module **beam_lib** [page 87] – An Interface To the BEAM File Format
- Erlang Module **c** [page 95] – Command Interface Module
- Erlang Module **calendar** [page 99] – Local and universal time, day-of-the-week, date and time conversions
- Erlang Module **dets** [page 104] – A Disk Based Term Storage
- Erlang Module **dict** [page 120] – Key-Value Dictionary
- Erlang Module **digraph** [page 125] – Directed Graphs
- Erlang Module **digraph_utils** [page 132] – Algorithms for Directed Graphs
- Erlang Module **epp** [page 137] – An Erlang Code Preprocessor
- Erlang Module **erl_eval** [page 139] – The Erlang Meta Interpreter
- Erlang Module **erl_expand_records** [page 142] – Expands Records in a Module
- Erlang Module **erl_id_trans** [page 143] – An Identity Parse Transform
- Erlang Module **erl_internal** [page 144] – Internal Erlang Definitions
- Erlang Module **erl_lint** [page 146] – The Erlang Code Linter
- Erlang Module **erl_parse** [page 148] – The Erlang Parser
- Erlang Module **erl_pp** [page 151] – The Erlang Pretty Printer
- Erlang Module **erl_scan** [page 154] – The Erlang Token Scanner
- Erlang Module **erl_tar** [page 160] – Unix ‘tar’ utility for reading and writing tar archives
- Erlang Module **ets** [page 166] – Built-In Term Storage
- Erlang Module **file_sorter** [page 190] – File Sorter
- Erlang Module **filelib** [page 195] – File utilities, such as wildcard matching of filenames
- Erlang Module **filename** [page 198] – Filename Manipulation Functions
- Erlang Module **gb_sets** [page 204] – General Balanced Trees
- Erlang Module **gb_trees** [page 210] – General Balanced Trees
- Erlang Module **gen_event** [page 215] – Generic Event Handling Behaviour
• Erlang Module `gen_fsm` [page 225] – Generic Finite State Machine Behaviour
• Erlang Module `gen_server` [page 236] – Generic Server Behaviour
• Erlang Module `io` [page 246] – Standard I/O Server Interface Functions
• Erlang Module `io_lib` [page 260] – I/O Library Functions
• Erlang Module `lib` [page 264] – A number of useful library functions
• Erlang Module `lists` [page 266] – List Processing Functions
• Erlang Module `log_mf.h` [page 282] – An Event Handler which Logs Events to Disk
• Erlang Module `math` [page 283] – Mathematical Functions
• Erlang Module `ms_transform` [page 285] – Parse_transform that translates fun syntax into match specifications.
• Erlang Module `orddict` [page 296] – Key-Value Dictionary as Ordered List
• Erlang Module `ordsets` [page 301] – Functions for Manipulating Sets as Ordered Lists
• Erlang Module `pg` [page 304] – Distributed, Named Process Groups
• Erlang Module `pool` [page 306] – Load Distribution Facility
• Erlang Module `proc_lib` [page 308] – Functions for asynchronous and synchronous start of processes adhering to the OTP design principles.
• Erlang Module `proplists` [page 313] – Support functions for property lists
• Erlang Module `qlc` [page 318] – Query Interface to Mnesia, ETS, Dets, etc
• Erlang Module `queue` [page 337] – Abstract Data Type for FIFO Queues
• Erlang Module `random` [page 343] – Pseudo random number generation
• Erlang Module `re` [page 345] – Perl like regular expressions for Erlang
• Erlang Module `regexp` [page 392] – Regular Expression Functions for Strings
• Erlang Module `sets` [page 397] – Functions for Set Manipulation
• Erlang Module `shell` [page 400] – The Erlang Shell
• Erlang Module `shell_default` [page 411] – Customizing the Erlang Environment
• Erlang Module `slave` [page 412] – Functions to Starting and Controlling Slave Nodes
• Erlang Module `sofs` [page 415] – Functions for Manipulating Sets of Sets
• Erlang Module `string` [page 438] – String Processing Functions
• Erlang Module `supervisor` [page 444] – Generic Supervisor Behaviour
• Erlang Module `sys` [page 454] – A Functional Interface to System Messages
• Erlang Module `timer` [page 461] – Timer Functions
• Erlang Module `unicode` [page 465] – Functions for converting Unicode characters
• Erlang Module `win32reg` [page 469] – win32reg provides access to the registry on Windows

STDLIB

No functions are exported.
array

The following functions are exported:

- default(Array::array()) -> term()
  [page 81] Get the value used for uninitialized entries.
- fix(Array::array()) -> array()
  [page 81] Fix the size of the array.
- foldl(Function, InitialAcc::term(), Array::array()) -> term()
  [page 81] Fold the elements of the array using the given function and initial accumulator value.
- foldr(Function, InitialAcc::term(), Array::array()) -> term()
  [page 81] Fold the elements of the array right-to-left using the given function and initial accumulator value.
- from_list(List::list()) -> array()
  [page 81] Equivalent to from_list(List, undefined).
- from_list(List::list(), Default::term()) -> array()
  [page 81] Convert a list to an extendible array.
- from_orddict(Orddict::list()) -> array()
  [page 82] Equivalent to from_orddict(Orddict, undefined).
- from_orddict(List::list(), Default::term()) -> array()
  [page 82] Convert an ordered list of pairs \( \text{Index, Value} \) to a corresponding extendible array.
- get(I::integer(), Array::array()) -> term()
  [page 82] Get the value of entry \( I \).
- is_array(X::term()) -> bool()
  [page 82] Returns true if \( X \) appears to be an array, otherwise false.
- is_fix(Array::array()) -> bool()
  [page 82] Check if the array has fixed size.
- map(Function, Array::array()) -> array()
  [page 82] Map the given function onto each element of the array.
- new() -> array()
  [page 82] Create a new, extendible array with initial size zero.
- new(Options::term()) -> array()
  [page 82] Create a new array according to the given options.
- new(Size::integer(), Options::term()) -> array()
  [page 82] Create a new array according to the given size and options.
- relax(Array::array()) -> array()
  [page 83] Make the array resizable.
- reset(I::integer(), Array::array()) -> array()
  [page 83] Reset entry \( I \) to the default value for the array.
- resize(Array::array()) -> array()
  [page 84] Change the size of the array to that reported by sparse_size/1.
- resize(Size::integer(), Array::array()) -> array()
  [page 84] Change the size of the array.
- set(I::integer(), Value::term(), Array::array()) -> array()
  [page 84] Set entry \( I \) of the array to Value.
- `size(Array::array())` -> `integer()`  
  [page 84] Get the number of entries in the array.

- `sparse_foldl(Function, InitialAcc::term(), Array::array())` -> `term()`  
  [page 84] Fold the elements of the array using the given function and initial accumulator value, skipping default-valued entries.

- `sparse_foldr(Function, InitialAcc::term(), Array::array())` -> `term()`  
  [page 84] Fold the elements of the array right-to-left using the given function and initial accumulator value, skipping default-valued entries.

- `sparse_map(Function, Array::array())` -> `array()`  
  [page 85] Map the given function onto each element of the array, skipping default-valued entries.

- `sparse_size(A::array())` -> `integer()`  
  [page 85] Get the number of entries in the array up until the last non-default valued entry.

- `sparse_to_list(Array::array())` -> `list()`  
  [page 85] Converts the array to a list, skipping default-valued entries.

- `sparse_to_orddict(Array::array())` -> `[[Index::integer(), Value::term()]]`  
  [page 85] Convert the array to an ordered list of pairs \{Index, Value\}, skipping default-valued entries.

- `to_list(Array::array())` -> `list()`  
  [page 85] Converts the array to a list.

- `to_orddict(Array::array())` -> `[[Index::integer(), Value::term()]]`  
  [page 85] Convert the array to an ordered list of pairs \{Index, Value\}.

base64

The following functions are exported:

- `encode(Data)` -> `Base64`  
  [page 86] Encodes data into base64.

- `encode_to_string(Data)` -> `Base64String`  
  [page 86] Encodes data into base64.

- `decode(Base64)` -> `Data`  
  [page 86] Decodes a base64 encoded string to data.

- `decode_to_string(Base64)` -> `DataString`  
  [page 86] Decodes a base64 encoded string to data.

- `mime_decode(Base64)` -> `Data`  
  [page 86] Decodes a base64 encoded string to data.

- `mime_decode_to_string(Base64)` -> `DataString`  
  [page 86] 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 90] Read selected chunks from a BEAM file or binary
- `chunks(Beam, [ChunkRef], [Option]) -> {ok, {Module, [ChunkResult]}} | {error, beam_lib, Reason}`
  [page 90] Read selected chunks from a BEAM file or binary
- `version(Beam) -> {ok, {Module, [Version]}} | {error, beam_lib, Reason}`
  [page 90] Read the BEAM file's module version
- `md5(Beam) -> {ok, {Module, MD5}} | {error, beam_lib, Reason}`
  [page 91] Read the BEAM file's module version
- `info(Beam) -> [{Item, Info}] | {error, beam_lib, Reason1}`
  [page 91] Information about a BEAM file
- `cmp(Beam1, Beam2) -> ok | {error, beam_lib, Reason}`
  [page 92] Compare two BEAM files
- `cmp_dirs(Dir1, Dir2) -> {Only1, Only2, Different} | {error, beam_lib, Reason1}`
  [page 92] Compare the BEAM files in two directories
- `diff_dirs(Dir1, Dir2) -> ok | {error, beam_lib, Reason1}`
  [page 92] Compare the BEAM files in two directories
- `strip(Beam1) -> {ok, {Module, Beam2}} | {error, beam_lib, Reason1}`
  [page 92] Removes chunks not needed by the loader from a BEAM file
- `strip_files(Files) -> {ok, [{Module, Beam2}]} | {error, beam_lib, Reason1}`
  [page 92] Removes chunks not needed by the loader from BEAM files
- `strip_release(Dir) -> {ok, [{Module, Filename}]} | {error, beam_lib, Reason1}`
  [page 93] Removes chunks not needed by the loader from all BEAM files of a release
- `format_error(Reason) -> Chars`
  [page 93] Return an English description of a BEAM read error reply
- `crypto_key_fun(CryptoKeyFun) -> ok | {error, Reason}`
  [page 93] Register a fun that provides a crypto key
- `clear_crypto_key_fun() -> {ok, Result}`
  [page 94] Unregister the current crypto key fun

C

The following functions are exported:

- `bt(Pid) -> void()`
  [page 95] Stack backtrace for a process
- `c(File) -> {ok, Module} | error`
  [page 95] Compile and load code in a file
- `c(File, Options) -> {ok, Module} | error`
  [page 95] Compile and load code in a file
- `cd(Dir) -> void()`
  [page 95] Change working directory
- `flush() -> void()`
  [page 96] Flush any messages sent to the shell
- `help() -> void() [page 96]` Help information
- `i() -> void() [page 96]` Information about the system
- `ni() -> void() [page 96]` Information about the system
- `i(X, Y, Z) -> void() [page 96]` Information about pid <X,Y,Z>
- `l(Module) -> void() [page 96]` Load or reload module
- `lc(Files) -> ok [page 96]` Compile a list of files
- `ls() -> void() [page 96]` List files in the current directory
- `ls(Dir) -> void() [page 96]` List files in a directory
- `m() -> void() [page 96]` Which modules are loaded
- `m(Module) -> void() [page 97]` Information about a module
- `memory() -> [{Type, Size}] [page 97]` Memory allocation information
- `memory(Type) -> Size [page 97]` Memory allocation information
- `memory([Type]) -> [{Type, Size}] [page 97]` Memory allocation information
- `nc(File) -> {ok, Module} | error [page 97]` Compile and load code in a file on all nodes
- `nc(File, Options) -> {ok, Module} | error [page 97]` Compile and load code in a file on all nodes
- `nl(Module) -> void() [page 97]` Load module on all nodes
- `pid(X, Y, Z) -> pid() [page 97]` Convert X,Y,Z to a pid
- `pwd() -> void() [page 97]` Print working directory
- `q() -> void() [page 98]` Quit - shorthand for `init:stop()`
- `regs() -> void() [page 98]` Information about registered processes
- `nregs() -> void() [page 98]` Information about registered processes
- `xm(ModSpec) -> void() [page 98]` Cross reference check a module
- `y(File) -> YeccRet [page 98]` Generate an LALR-1 parser
- `y(File, Options) -> YeccRet [page 98]` Generate an LALR-1 parser
calendar

The following functions are exported:

- `date_to_gregorian_days(Date) -> Days`
  [page 100] Compute the number of days from year 0 up to the given date
- `date_to_gregorian_days(Year, Month, Day) -> Days`
  [page 100] Compute the number of days from year 0 up to the given date
- `datetime_to_gregorian_seconds([Date, Time]) -> Seconds`
  [page 100] Compute the number of seconds from year 0 up to the given date and time
- `day_of_the_week(Date) -> DayNumber`
  [page 100] Compute the day of the week
- `day_of_the_week(Year, Month, Day) -> DayNumber`
  [page 100] Compute the day of the week
- `gregorian_days_to_date(Days) -> Date`
  [page 100] Compute the date given the number of gregorian days
- `gregorian_seconds_to_datetime(Seconds) -> [Date, Time]`
  [page 100] Compute the date given the number of gregorian days
- `is_leap_year(Year) -> bool()`
  [page 100] Check if a year is a leap year
- `last_day_of_the_month(Year, Month) -> int()`
  [page 100] Compute the number of days in a month
- `local_time() -> [Date, Time]`
  [page 101] Compute local time
- `local_time_to_universal_time([Date1, Time1]) -> [Date2, Time2]`
  [page 101] Convert from local time to universal time (deprecated)
- `local_time_to_universal_time_dst([Date1, Time1]) -> [[Date, Time]]`
  [page 101] Convert from local time to universal time(s)
- `now_to_local_time(Now) -> [Date, Time]`
  [page 101] Convert now to local date and time
- `now_to_universal_time(Now) -> [Date, Time]`
  [page 102] Convert now to date and time
- `now_to_datetime(Now) -> [Date, Time]`
  [page 102] Convert now to date and time
- `seconds_to_daystime(Seconds) -> [Days, Time]`
  [page 102] Compute days and time from seconds
- `seconds_to_time(Seconds) -> Time`
  [page 102] Compute time from seconds
- `time_difference(T1, T2) -> [Days, Time]`
  [page 102] Compute the difference between two times (deprecated)
- `time_to_seconds(Time) -> Seconds`
  [page 102] Compute the number of seconds since midnight up to the given time
- `universal_time() -> [Date, Time]`
  [page 102] Compute universal time
- `universal_time_to_local_time([Date1, Time1]) -> [Date2, Time2]`
  [page 103] Convert from universal time to local time
• `valid_date(Date)` -> `bool()`  
  [page 103] Check if a date is valid

• `valid_date(Year, Month, Day)` -> `bool()`  
  [page 103] Check if a date is valid

dets

The following functions are exported:

• `all()` -> `[Name]`  
  [page 105] 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 105] Return a chunk of objects stored in a Dets table.

• `close(Name)` -> `ok | {error, Reason}`  
  [page 106] Close a Dets table.

• `delete(Name, Key)` -> `ok | {error, Reason}`  
  [page 106] Delete all objects with a given key from a Dets table.

• `delete_all_objects(Name)` -> `ok | {error, Reason}`  
  [page 106] Delete all objects from a Dets table.

• `delete_object(Name, Object)` -> `ok | {error, Reason}`  
  [page 106] Delete a given object from a Dets table.

• `first(Name)` -> `Key | '$end_of_table'`  
  [page 106] Return the first key stored in a Dets table.

• `foldl(Function, Acc0, Name)` -> `Acc1 | {error, Reason}`  
  [page 107] Fold a function over a Dets table.

• `foldr(Function, Acc0, Name)` -> `Acc1 | {error, Reason}`  
  [page 107] Fold a function over a Dets table.

• `from_ets(Name, EtsTab)` -> `ok | {error, Reason}`  
  [page 107] Replace the objects of a Dets table with the objects of an Ets table.

• `info(Name)` -> `InfoList | undefined`  
  [page 107] Return information about a Dets table.

• `info(Name, Item)` -> `Value | undefined`  
  [page 108] Return the information associated with a given item for a Dets table.

• `init_table(Name, InitFun [, Options])` -> `ok | {error, Reason}`  
  [page 108] Replace all objects of a Dets table.

• `insert(Name, Objects)` -> `ok | {error, Reason}`  
  [page 109] Insert one or more objects into a Dets table.

• `insert_new(Name, Objects)` -> `Bool`  
  [page 109] Insert one or more objects into a Dets table.

• `is_compatible_bchunk_format(Name, BchunkFormat)` -> `Bool`  
  [page 110] Test compatibility of a table's chunk data.

• `is_dets_file(FileName)` -> `Bool | {error, Reason}`  
  [page 110] Test for a Dets table.

• `lookup(Name, Key)` -> `[Object] | {error, Reason}`  
  [page 110] Return all objects with a given key stored in a Dets table.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>match(Continuation) -&gt; ([Match], Continuation2)</td>
<td>Match a chunk of objects stored in a Dets table and return a list of variable bindings.</td>
</tr>
<tr>
<td>match(Name, Pattern) -&gt; [Match]</td>
<td>Match the objects stored in a Dets table and return a list of variable bindings.</td>
</tr>
<tr>
<td>match(Name, Pattern, N) -&gt; [[Match], Continuation]</td>
<td>Match the first chunk of objects stored in a Dets table and return a list of variable bindings.</td>
</tr>
<tr>
<td>match_delete(Name, Pattern) -&gt; ok</td>
<td>Delete all objects that match a given pattern from a Dets table.</td>
</tr>
<tr>
<td>match_object(Continuation) -&gt; ([Object], Continuation2)</td>
<td>Match a chunk of objects stored in a Dets table and return a list of objects.</td>
</tr>
<tr>
<td>match_object(Name, Pattern) -&gt; [Object]</td>
<td>Match the objects stored in a Dets table and return a list of objects.</td>
</tr>
<tr>
<td>match_object(Name, Pattern, N) -&gt; [[Object], Continuation]</td>
<td>Match the first chunk of objects stored in a Dets table and return a list of objects.</td>
</tr>
<tr>
<td>member(Name, Key) -&gt; Bool</td>
<td>Test for occurrence of a key in a Dets table.</td>
</tr>
<tr>
<td>next(Name, Key1) -&gt; Key2</td>
<td>Return the next key in a Dets table.</td>
</tr>
<tr>
<td>open_file(Filename) -&gt; {ok, Reference}</td>
<td>Open an existing Dets table.</td>
</tr>
<tr>
<td>open_file(Name, Args) -&gt; {ok, Name}</td>
<td>Open a Dets table.</td>
</tr>
<tr>
<td>pid2name(Pid) -&gt; {ok, Name}</td>
<td>Return the name of the Dets table handled by a pid.</td>
</tr>
<tr>
<td>repair_continuation(Continuation, MatchSpec) -&gt; Continuation2</td>
<td>Repair a continuation from select/1 or select/3.</td>
</tr>
<tr>
<td>safe_fixtable(Name, Fix)</td>
<td>Fix a Dets table for safe traversal.</td>
</tr>
<tr>
<td>select(Continuation) -&gt; {Selection, Continuation2}</td>
<td>Apply a match specification to some objects stored in a Dets table.</td>
</tr>
<tr>
<td>select(Name, MatchSpec) -&gt; Selection</td>
<td>Apply a match specification to all objects stored in a Dets table.</td>
</tr>
<tr>
<td>select(Name, MatchSpec, N) -&gt; {Selection, Continuation}</td>
<td>Apply a match specification to the first chunk of objects stored in a Dets table.</td>
</tr>
<tr>
<td>select_delete(Name, MatchSpec) -&gt; N</td>
<td>Delete all objects that match a given pattern from a Dets table.</td>
</tr>
</tbody>
</table>
- `slot(Name, I) -> 'end_of_table' | [Object] | {error, Reason}`
  [page 117] Return the list of objects associated with a slot of a Dets table.
- `sync(Name) -> ok | {error, Reason}`
  [page 117] Ensure that all updates made to a Dets table are written to disk.
- `table(Name [, , Options]) -> QueryHandle`
  [page 118] Return a QLC query handle.
- `to_ets(Name, EtsTab) -> EtsTab | {error, Reason}`
  [page 118] Insert all objects of a Dets table into an Ets table.
- `traverse(Name, Fun) -> Return | {error, Reason}`
  [page 119] Apply a function to all or some objects stored in a Dets table.
- `update_counter(Name, Key, Increment) -> Result`
  [page 119] Update a counter object stored in a Dets table.

dict

The following functions are exported:

- `append(Key, Value, Dict1) -> Dict2`
  [page 120] Append a value to keys in a dictionary
- `append_list(Key, ValList, Dict1) -> Dict2`
  [page 120] Append new values to keys in a dictionary
- `erase(Key, Dict1) -> Dict2`
  [page 120] Erase a key from a dictionary
- `fetch(Key, Dict) -> Value`
  [page 121] Look-up values in a dictionary
- `fetch_keys(Dict) -> Keys`
  [page 121] Return all keys in a dictionary
- `filter(Pred, Dict1) -> Dict2`
  [page 121] Choose elements which satisfy a predicate
- `find(Key, Dict) -> {ok, Value} | error`
  [page 121] Search for a key in a dictionary
- `fold(Fun, Acc0, Dict) -> Acc1`
  [page 121] Fold a function over a dictionary
- `from_list(List) -> Dict`
  [page 121] Convert a list of pairs to a dictionary
- `is_key(Key, Dict) -> bool()`
  [page 122] Test if a key is in a dictionary
- `map(Fun, Dict1) -> Dict2`
  [page 122] Map a function over a dictionary
- `merge(Fun, Dict1, Dict2) -> Dict3`
  [page 122] Merge two dictionaries
- `new() -> dictionary()`
  [page 122] Create a dictionary
- `size(Dict) -> int()`
  [page 122] Return the number of elements in a dictionary
- `store(Key, Value, Dict1) -> Dict2`
  [page 122] Store a value in a dictionary
• to_list(Dict) -> List
  [page 123] Convert a dictionary to a list of pairs
• update(Key, Fun, Dict1) -> Dict2
  [page 123] Update a value in a dictionary
• update(Key, Fun, Initial, Dict1) -> Dict2
  [page 123] Update a value in a dictionary
• update_counter(Key, Increment, Dict1) -> Dict2
  [page 123] Increment a value in a dictionary

digraph

The following functions are exported:

• add_edge(G, E, V1, V2, Label) -> edge() | {error, Reason}
  [page 125] Add an edge to a digraph.
• add_edge(G, V1, V2, Label) -> edge() | {error, Reason}
  [page 125] Add an edge to a digraph.
• add_edge(G, V1, V2) -> edge() | {error, Reason}
  [page 125] Add an edge to a digraph.
• add_vertex(G, V, Label) -> vertex()
  [page 126] Add or modify a vertex of a digraph.
• add_vertex(G, V) -> vertex()
  [page 126] Add or modify a vertex of a digraph.
• add_vertex(G) -> vertex()
  [page 126] Add or modify a vertex of a digraph.
• del_edge(G, E) -> true
  [page 126] Delete an edge from a digraph.
• del_edges(G, Edges) -> true
  [page 126] Delete edges from a digraph.
• del_path(G, V1, V2) -> true
  [page 126] Delete paths from a digraph.
• del_vertex(G, V) -> true
  [page 127] Delete a vertex from a digraph.
• del_vertices(G, Vertices) -> true
  [page 127] Delete vertices from a digraph.
• delete(G) -> true
  [page 127] Delete a digraph.
• edge(G, E) -> {E, V1, V2, Label} | false
  [page 127] Return the vertices and the label of an edge of a digraph.
• edges(G) -> Edges
  [page 127] Return all edges of a digraph.
• edges(G, V) -> Edges
  [page 127] Return the edges emanating from or incident on a vertex of a digraph.
• get_cycle(G, V) -> Vertices | false
  [page 128] Find one cycle in a digraph.
• get_path(G, V1, V2) -> Vertices | false
  [page 128] Find one path in a digraph.
• `get_short_cycle(G, V)` - Vertices | false
  [page 128] Find one short cycle in a digraph.

• `get_short_path(G, V1, V2)` - Vertices | false
  [page 128] Find one short path in a digraph.

• `in_degree(G, V)` - integer()
  [page 129] Return the in-degree of a vertex of a digraph.

• `in_edges(G, V)` - Edges
  [page 129] Return all edges incident on a vertex of a digraph.

• `in_neighbours(G, V)` - Vertices
  [page 129] Return all in-neighbours of a vertex of a digraph.

• `info(G)` - InfoList
  [page 129] Return information about a digraph.

• `new()` -> digraph()
  [page 130] Return a protected empty digraph, where cycles are allowed.

• `new(Type)` -> digraph()
  [page 130] Create a new empty digraph.

• `no_edges(G)` -> integer() >= 0
  [page 130] Return the number of edges of the a digraph.

• `no_vertices(G)` -> integer() >= 0
  [page 130] Return the number of vertices of a digraph.

• `out_degree(G, V)` -> integer()
  [page 130] Return the out-degree of a vertex of a digraph.

• `out_edges(G, V)` - Edges
  [page 130] Return all edges emanating from a vertex of a digraph.

• `out_neighbours(G, V)` - Vertices
  [page 130] Return all out-neighbours of a vertex of a digraph.

• `vertex(G, V)` - {V, Label} | false
  [page 131] Return the label of a vertex of a digraph.

• `vertices(G)` - Vertices
  [page 131] Return all vertices of a digraph.

digraph_utils

The following functions are exported:

• `arborescence_root(Digraph)` - no | {yes, Root}
  [page 133] Check if a digraph is an arborescence.

• `components(Digraph)` - [Component]
  [page 133] Return the components of a digraph.

• `condensation(Digraph)` -> CondensedDigraph
  [page 133] Return a condensed graph of a digraph.

• `cyclic_strong_components(Digraph)` - [StrongComponent]
  [page 133] Return the cyclic strong components of a digraph.

• `is_acyclic(Digraph)` -> bool()
  [page 133] Check if a digraph is acyclic.

• `is_arborescence(Digraph)` -> bool()
  [page 134] Check if a digraph is an arborescence.
• `is_tree(Digraph) -> bool()`  
  [page 134] Check if a digraph is a tree.

• `loop_vertices(Digraph) -> Vertices`  
  [page 134] Return the vertices of a digraph included in some loop.

• `postorder(Digraph) -> Vertices`  
  [page 134] Return the vertices of a digraph in post-order.

• `preorder(Digraph) -> Vertices`  
  [page 134] Return the vertices of a digraph in pre-order.

• `reachable(Vertices, Digraph) -> Vertices`  
  [page 134] Return the vertices reachable from some vertices of a digraph.

• `reachable_neighbours(Vertices, Digraph) -> Vertices`  
  [page 135] Return the neighbours reachable from some vertices of a digraph.

• `reaching(Vertices, Digraph) -> Vertices`  
  [page 135] Return the vertices that reach some vertices of a digraph.

• `reaching_neighbours(Vertices, Digraph) -> Vertices`  
  [page 135] Return the neighbours that reach some vertices of a digraph.

• `strong_components(Digraph) -> [StrongComponent]`  
  [page 135] Return the strong components of a digraph.

• `subgraph(Digraph, Vertices [, Options]) -> Subgraph`  
  [page 135] Return a subgraph of a digraph.

• `topsort(Digraph) -> Vertices | false`  
  [page 136] Return a topological sorting of the vertices of a digraph.

---

**epp**

The following functions are exported:

• `open(FileName, IncludePath) -> {ok, Epp} | {error, ErrorDescriptor}`  
  [page 137] Open a file for preprocessing

• `open(FileName, IncludePath, PredefMacros) -> {ok, Epp} | {error, ErrorDescriptor}`  
  [page 137] Open a file for preprocessing

• `close(Epp) -> ok`  
  [page 137] Close the preprocessing of the file associated with Epp

• `parse_erl_form(Epp) -> {ok, AbsForm} | {eof, Line} | {error, ErrorInfo}`  
  [page 137] Return the next Erlang form from the opened Erlang source file

• `parse_file(FileName, IncludePath, PredefMacro) -> {ok, [Form]} | {error, OpenError}`  
  [page 137] Preprocesse and parse an Erlang source file

---

**erl_eval**

The following functions are exported:

• `exprs(Expressions, Bindings) -> {value, Value, NewBindings}`  
  [page 139] Evaluate expressions

---

STDLIB Reference Manual 35
• `exprs(Expressions, Bindings, LocalFunctionHandler) -> {value, Value, NewBindings}`
  [page 139] Evaluate expressions

• `exprs(Expressions, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {value, Value, NewBindings}`
  [page 139] Evaluate expressions

• `expr(Expression, Bindings) -> {value, Value, NewBindings}`
  [page 139] Evaluate expression

• `expr(Expression, Bindings, LocalFunctionHandler) -> {value, Value, NewBindings}`
  [page 139] Evaluate expression

• `expr(Expression, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {value, Value, NewBindings}`
  [page 139] Evaluate expression

• `expr_list(ExpressionList, Bindings) -> {ValueList, NewBindings}`
  [page 140] Evaluate a list of expressions

• `expr_list(ExpressionList, Bindings, LocalFunctionHandler) -> {ValueList, NewBindings}`
  [page 140] Evaluate a list of expressions

• `expr_list(ExpressionList, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {ValueList, NewBindings}`
  [page 140] Evaluate a list of expressions

• `new_bindings() -> BindingStruct`
  [page 140] Return a bindings structure

• `bindings(BindingStruct) -> Bindings`
  [page 140] Return bindings

• `binding(Name, BindingStruct) -> Binding`
  [page 140] Return bindings

• `add_binding(Name, Value, Bindings) -> BindingStruct`
  [page 140] Add a binding

• `del_binding(Name, Bindings) -> BindingStruct`
  [page 140] Delete a binding

**erl_expand_records**

The following functions are exported:

• `module(AbsForms, CompileOptions) -> AbsForms`
  [page 142] Expand all records in a module

**erl_id_trans**

The following functions are exported:

• `parse_transform(Forms, Options) -> Forms`
  [page 143] Transform Erlang forms
erl\_internal

The following functions are exported:

- `bif(Name, Arity) -> bool()` [page 144] Test for an Erlang BIF
- `guard\_bif(Name, Arity) -> bool()` [page 144] Test for an Erlang BIF allowed in guards
- `type\_test(Name, Arity) -> bool()` [page 144] Test for a valid type test
- `arith\_op(OpName, Arity) -> bool()` [page 144] Test for an arithmetic operator
- `bool\_op(OpName, Arity) -> bool()` [page 144] Test for a Boolean operator
- `comp\_op(OpName, Arity) -> bool()` [page 145] Test for a comparison operator
- `list\_op(OpName, Arity) -> bool()` [page 145] Test for a list operator
- `send\_op(OpName, Arity) -> bool()` [page 145] Test for a send operator
- `op\_type(OpName, Arity) -> Type` [page 145] Return operator type

erl\_lint

The following functions are exported:

- `module(AbsForms) -> {ok, Warnings} | {error, Errors, Warnings}` [page 146] Check a module for errors
- `module(AbsForms, FileName) -> {ok, Warnings} | {error, Errors, Warnings}` [page 146] Check a module for errors
- `module(AbsForms, FileName, CompileOptions) -> {ok, Warnings} | {error, Errors, Warnings}` [page 146] Check a module for errors
- `is\_guard\_test(Expr) -> bool()` [page 147] Test for a guard test
- `format\_error(Descriptor) -> Chars` [page 147] Format an error descriptor

erl\_parse

The following functions are exported:

- `parse\_form(Tokens) -> {ok, AbsForm} | {error, ErrorInfo}` [page 148] Parse an Erlang form
- `parse\_exprs(Tokens) -> {ok, Exprs\_list} | {error, ErrorInfo}` [page 148] Parse Erlang expressions
• `parse_term(Tokens) -> {ok, Term} | {error, ErrorInfo}`  
  [page 148] Parse an Erlang term
• `format_error(ErrorDescriptor) -> Chars`  
  [page 149] Format an error descriptor
• `tokens(AbsTerm) -> Tokens`  
  [page 149] Generate a list of tokens for an expression
• `tokens(AbsTerm, MoreTokens) -> Tokens`  
  [page 149] Generate a list of tokens for an expression
• `normalise(AbsTerm) -> Data`  
  [page 149] Convert abstract form to an Erlang term
• `abstract(Data) -> AbsTerm`  
  [page 149] Convert an Erlang term into an abstract form

**erl_pp**

The following functions are exported:

• `form(Form) -> DeepCharList`  
  [page 151] Pretty print a form
• `form(Form, HookFunction) -> DeepCharList`  
  [page 151] Pretty print a form
• `attribute(Attribute) -> DeepCharList`  
  [page 151] Pretty print an attribute
• `attribute(Attribute, HookFunction) -> DeepCharList`  
  [page 151] Pretty print an attribute
• `function(Function) -> DeepCharList`  
  [page 151] Pretty print a function
• `function(Function, HookFunction) -> DeepCharList`  
  [page 151] Pretty print a function
• `guard(Guard) -> DeepCharList`  
  [page 151] Pretty print a guard
• `guard(Guard, HookFunction) -> DeepCharList`  
  [page 151] Pretty print a guard
• `exprs(Expressions) -> DeepCharList`  
  [page 152] Pretty print `Expressions`
• `exprs(Expressions, HookFunction) -> DeepCharList`  
  [page 152] Pretty print `Expressions`
• `exprs(Expressions, Indent, HookFunction) -> DeepCharList`  
  [page 152] Pretty print `Expressions`
• `expr(Expression) -> DeepCharList`  
  [page 152] Pretty print one `Expression`
• `expr(Expression, HookFunction) -> DeepCharList`  
  [page 152] Pretty print one `Expression`
• `expr(Expression, Indent, HookFunction) -> DeepCharList`  
  [page 152] Pretty print one `Expression`
• `expr(Expression, Indent, Precedence, HookFunction) -> DeepCharList`  
  [page 152] Pretty print one `Expression`
erl_scan

The following functions are exported:

- `string(String) -> Return`  
  [page 154] Scan a string and return the Erlang tokens
- `string(String, StartLocation) -> Return`  
  [page 154] Scan a string and return the Erlang tokens
- `string(String, StartLocation, Options) -> Return`  
  [page 154] Scan a string and return the Erlang tokens
- `tokens(Continuation, CharSpec, StartLocation) -> Return`  
  [page 155] Re-entrant scanner
- `tokens(Continuation, CharSpec, StartLocation, Options) -> Return`  
  [page 155] Re-entrant scanner
- `reserved_word(Atom) -> bool()`  
  [page 156] Test for a reserved word
- `token_info(Token) -> TokenInfo`  
  [page 156] Return information about a token
- `token_info(Token, TokenItemSpec) -> TokenInfo`  
  [page 156] Return information about a token
- `attributes_info(Attributes) -> AttributesInfo`  
  [page 157] Return information about token attributes
- `attributes_info(Attributes, AttributeItemSpec) -> AttributesInfo`  
  [page 157] Return information about token attributes
- `set_attribute(AttributeItem, Attributes, SetAttributeFun) -> AttributesInfo`  
  [page 158] Set a token attribute value
- `format_error(ErrorDescriptor) -> string()`  
  [page 158] Format an error descriptor

erl_tar

The following functions are exported:

- `add(TarDescriptor, Filename, Options) -> RetValue`  
  [page 161] Add a file to an open tar file
- `add(TarDescriptor, FilenameOrBin, NameInArchive, Options) -> RetValue`  
  [page 161] Add a file to an open tar file
- `close(TarDescriptor)`  
  [page 161] Close an open tar file
- `create(Name, FileList) -> RetValue`  
  [page 161] Create a tar archive
- `create(Name, FileList, OptionList)`  
  [page 162] Create a tar archive with options
- `extract(Name) -> RetValue`  
  [page 162] Extract all files from a tar file
• extract(Name, OptionList)
  [page 162] Extract files from a tar file
• format_error(Reason) -> string()
  [page 163] Convert error term to a readable string
• open(Name, OpenModeList) -> RetValue
  [page 163] Open a tar file for writing.
• table(Name) -> RetValue
  [page 164] Retrieve the name of all files in a tar file
• table(Name, Options)
  [page 164] Retrieve name and information of all files in a tar file
• t(Name)
  [page 164] Print the name of each file in a tar file
• tt(Name)
  [page 165] Print name and information for each file in a tar file

ets

The following functions are exported:

• all() -> [Tab]
  [page 167] Return a list of all ETS tables
• delete(Tab) -> true
  [page 167] Delete an entire ETS table.
• delete(Tab, Key) -> true
  [page 167] Delete all objects with a given key from an ETS table.
• delete_all_objects(Tab) -> true
  [page 167] Delete all objects in an ETS table.
• delete_object(Tab, Object) -> true
  [page 167] Deletes a specific from an ETS table.
• file2tab(Filename) -> {ok,Tab} | {error,Reason}
  [page 168] Read an ETS table from a file.
• file2tab(Filename,Options) -> {ok,Tab} | {error,Reason}
  [page 168] Read an ETS table from a file.
• first(Tab) -> Key | '$end_of_table'
  [page 168] Return the first key in an ETS table.
• foldl(Function, Acc0, Tab) -> Acc1
  [page 169] Fold a function over an ETS table
• foldr(Function, Acc0, Tab) -> Acc1
  [page 169] Fold a function over an ETS table
• from_dets(Tab, DetsTab) -> true
  [page 169] Fill an ETS table with objects from a Dets table.
• fun2ms(LiteralFun) -> MatchSpec
  [page 169] Pseudo function that transforms fun syntax to a match spec.
• i() -> ok
  [page 171] Display information about all ETS tables on tty.
• i(Tab) -> ok
  [page 171] Browse an ETS table on tty.
- `info(Tab) -> [[Item, Value]] | undefined` [page 171] Return information about an ETS table.
- `info(Tab, Item) -> Value | undefined` [page 171] Return the information associated with given item for an ETS table.
- `init_table(Name, InitFun) -> true` [page 172] Replace all objects of an ETS table.
- `insert(Tab, ObjectOrObjects) -> true` [page 172] Insert an object into an ETS table.
- `insert_new(Tab, ObjectOrObjects) -> bool()` [page 173] Insert an object into an ETS table if the key is not already present.
- `is_compiled_ms(Term) -> bool()` [page 173] Checks if an Erlang term is the result of ets:match_spec_compile.
- `last(Tab) -> Key | '$end_of_table'` [page 173] Return the last key in an ETS table of type ordered_set.
- `lookup(Tab, Key) -> [Object]` [page 174] Return all objects with a given key in an ETS table.
- `lookup_element(Tab, Key, Pos) -> Elem` [page 174] Return the Pos:th element of all objects with a given key in an ETS table.
- `match(Tab, Pattern) -> [Match]` [page 174] Match the objects in an ETS table against a pattern.
- `match(Tab, Pattern, Limit) -> [{[Match],Continuation}] | '$end_of_table'` [page 175] Match the objects in an ETS table against a pattern and returns part of the answers.
- `match(Continuation) -> [{[Match],Continuation}] | '$end_of_table'` [page 175] Continues matching objects in an ETS table.
- `match_delete(Tab, Pattern) -> true` [page 175] Delete all objects which match a given pattern from an ETS table.
- `match_object(Tab, Pattern) -> [Object]` [page 176] Match the objects in an ETS table against a pattern.
- `match_object(Tab, Pattern, Limit) -> {[Match],Continuation}] | '$end_of_table'` [page 176] Match the objects in an ETS table against a pattern and returns part of the answers.
- `match_object(Continuation) -> {[Match],Continuation}] | '$end_of_table'` [page 176] Continues matching objects in an ETS table.
- `match_spec_compile(MatchSpec) -> CompiledMatchSpec` [page 176] Compiles a match specification into its internal representation.
- `match_spec_run(List,CompiledMatchSpec) -> list()` [page 177] Performs matching, using a compiled match_spec, on a list of tuples.
- `member(Tab, Key) -> true | false` [page 177] Tests for occurrence of a key in an ETS table.
- `new(Name, Options) -> tid() | atom()` [page 178] Create a new ETS table.
- `next(Tab, Key1) -> Key2 | '$end_of_table'`  
  [page 178] Return the next key in an ETS table.

- `prev(Tab, Key1) -> Key2 | '$end_of_table'`  
  [page 179] Return the previous key in an ETS table of typeordered_set.

- `rename(Tab, Name) -> Name`  
  [page 179] Rename a named ETS table.

- `repair_continuation(Continuation, MatchSpec) -> Continuation`  
  [page 179] Repair a continuation from ets:select/1 or ets:select/3 that has passed through external representation.

- `safe_fixtable(Tab, true|false) -> true`  
  [page 180] Fix an ETS table for safe traversal.

- `select(Tab, MatchSpec) -> [Match]`  
  [page 181] Match the objects in an ETS table against a match_spec.

- `select(Tab, MatchSpec, Limit) -> [[Match],Continuation] | '$end_of_table'`  
  [page 182] 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 183] Continue matching objects in an ETS table.

- `select_delete(Tab, MatchSpec) -> NumDeleted`  
  [page 183] 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 183] 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 184] Return all objects in a given slot of an ETS table.

- `tab2file(Tab, Filename) -> ok | {error,Reason}`  
  [page 184] Dump an ETS table to a file.

- `tab2list(Tab) -> [Object]`  
  [page 185] Return a list of all objects in an ETS table.

- `tabfile_info(Filename) -> {ok, TableInfo} | {error, Reason}`  
  [page 185] Return a list of all objects in an ETS table.

- `table(Tab [, Options]) -> QueryHandle`  
  [page 186] Return a QLC query handle.

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

- `to_dets(Tab, DetsTab) -> Tab`  
  [page 187] Fill a Dets table with objects from an ETS table.

- `update_counter(Tab, Key, UpdateOp) -> Result`  
  [page 188] Update a counter object in an ETS table.

- `update_counter(Tab, Key, [UpdateOp]) -> [Result]`  
  [page 188] Update a counter object in an ETS table.

- `update_counter(Tab, Key, Incr) -> Result`  
  [page 188] Update a counter object in an ETS table.
• `update_element(Tab, Key, {Pos,Value}) -> true | false`
  [page 188] Updates the Pos:th element of the object with a given key in an ETS table.

• `update_element(Tab, Key, [{Pos,Value}]) -> true | false`
  [page 188] Updates the Pos:th element of the object with a given key in an ETS table.

file_sorter

The following functions are exported:

• `sort(FileName) -> Reply`
  [page 193] Sort terms on files.

• `sort(Input, Output) -> Reply`
  [page 193] Sort terms on files.

• `sort(Input, Output, Options) -> Reply`
  [page 193] Sort terms on files.

• `keysort(KeyPos, FileName) -> Reply`
  [page 193] Sort terms on files by key.

• `keysort(KeyPos, Input, Output) -> Reply`
  [page 193] Sort terms on files by key.

• `keysort(KeyPos, Input, Output, Options) -> Reply`
  [page 193] Sort terms on files by key.

• `merge(FileNames, Output) -> Reply`
  [page 193] Merge terms on files.

• `merge(FileNames, Output, Options) -> Reply`
  [page 193] Merge terms on files.

• `keymerge(KeyPos, FileNames, Output) -> Reply`
  [page 194] Merge terms on files by key.

• `keymerge(KeyPos, FileNames, Output, Options) -> Reply`
  [page 194] Merge terms on files by key.

• `check(FileName) -> Reply`
  [page 194] Check whether terms on files are sorted.

• `check(FileNames, Options) -> Reply`
  [page 194] Check whether terms on files are sorted.

• `keycheck(KeyPos, FileName) -> CheckReply`
  [page 194] Check whether terms on files are sorted by key.

• `keycheck(KeyPos, FileNames, Options) -> Reply`
  [page 194] Check whether terms on files are sorted by key.

filelib

The following functions are exported:

• `ensure_dir(Name) -> ok | {error, Reason}`
  [page 195] Ensure that all parent directories for a file or directory exist.

• `file_size(Filename) -> integer()`
• fold_files(Dir, RegExp, Recursive, Fun, AccIn) -> AccOut
  [page 195] Fold over all files matching a regular expression.
• is_dir(Name) -> true | false
  [page 196] Test whether Name refer to a directory or not
• is_file(Name) -> true | false
  [page 196] Test whether Name refer to a file or directory.
• is_regular(Name) -> true | false
  [page 196] Test whether Name refer to a (regular) file.
• last_modified(Name) -> \{\{Year, Month, Day\}, \{Hour, Min, Sec\}\} | 0
  [page 196] Return the local date and time when a file was last modified.
• wildcard(Wildcard) -> list()
• wildcard(Wildcard, Cwd) -> list()
  [page 197] Match filenames using Unix-style wildcards starting at a specified directory.

filename

The following functions are exported:

• absname(Filename) -> string()
  [page 198] Convert a filename to an absolute name, relative the working directory
• absname(Filename, Dir) -> string()
  [page 199] Convert a filename to an absolute name, relative a specified directory
• absname_join(Dir, Filename) -> string()
  [page 199] Join an absolute directory with a relative filename
• basename(Filename) -> string()
  [page 199] Return the last component of a filename
• basename(Filename, Ext) -> string()
  [page 199] Return the last component of a filename, stripped of the specified extension
• dirname(Filename) -> string()
  [page 200] Return the directory part of a path name
• extension(Filename) -> string()
  [page 200] Return the file extension
• flatten(Filename) -> string()
  [page 200] Convert a filename to a flat string
• join(Components) -> string()
  [page 200] Join a list of filename components with directory separators
• join(Name1, Name2) -> string()
  [page 201] Join two filename components with directory separators
• nativename(Path) -> string()
  [page 201] Return the native form of a file path
• pathtype(Path) -> absolute | relative | volumerelative
  [page 201] Return the type of a path
• rootname(Filename) -> string()
  [page 201] Remove a filename extension
• rootname(Filename, Ext) -> string()
  [page 201] Remove a filename extension
• split(Filename) -> Components
  [page 202] Split a filename into its path components
• find_src(Beam) -> SourceFile, Options | error, [ErrorReason, Module]
  [page 202] Find the filename and compiler options for a module
• find_src(Beam, Rules) -> SourceFile, Options | error, [ErrorReason, Module]
  [page 202] Find the filename and compiler options for a module

gb_sets

The following functions are exported:
• add(Element, Set1) -> Set2
  [page 205] Add a (possibly existing) element to a gb_set
• add_element(Element, Set1) -> Set2
  [page 205] Add a (possibly existing) element to a gb_set
• balance(Set1) -> Set2
  [page 205] Rebalance tree representation of a gb_set
• delete(Element, Set1) -> Set2
  [page 205] Remove an element from a gb_set
• delete_any(Element, Set1) -> Set2
  [page 205] Remove a (possibly non-existing) element from a gb_set
• del_element(Element, Set1) -> Set2
  [page 205] Remove a (possibly non-existing) element from a gb_set
• difference(Set1, Set2) -> Set3
  [page 205] Return the difference of two gb_sets
• subtract(Set1, Set2) -> Set3
  [page 206] Return the difference of two gb_sets
• empty() -> Set
  [page 206] Return an empty gb_set
• new() -> Set
  [page 206] Return an empty gb_set
• filter(Pred, Set1) -> Set2
  [page 206] Filter gb_set elements
• fold(Function, Acc0, Set) -> Acc1
  [page 206] Fold over gb_set elements
• from_list(List) -> Set
  [page 206] Convert a list into a gb_set
• from_ordset(List) -> Set
  [page 206] Make a gb_set from an ordset list
• insert(Element, Set1) -> Set2
  [page 206] Add a new element to a gb_set
• intersection(Set1, Set2) -> Set3
  [page 207] Return the intersection of two gb_sets
• `intersection(SetList) -> Set`
  [page 207] Return the intersection of a list of gb_sets

• `is_disjoint(Set1, Set2) -> bool()`
  [page 207] Check whether two gb_sets are disjoint

• `is_empty(Set) -> bool()`
  [page 207] Test for empty gb_set

• `is_member(Element, Set) -> bool()`
  [page 207] Test for membership of a gb_set

• `is_element(Element, Set) -> bool()`
  [page 207] Test for membership of a gb_set

• `is_set(Term) -> bool()`
  [page 207] Test for a gb_set

• `is_subset(Set1, Set2) -> bool()`
  [page 207] Test for subset

• `iterator(Set) -> Iter`
  [page 208] Return an iterator for a gb_set

• `largest(Set) -> term()`
  [page 208] Return largest element

• `next(Iter1) -> {Element, Iter2} | none`
  [page 208] Traverse a gb_set with an iterator

• `singleton(Element) -> gb_set()`
  [page 208] Return a gb_set with one element

• `size(Set) -> int()`
  [page 208] Return the number of elements in a gb_set

• `smallest(Set) -> term()`
  [page 208] Return smallest element

• `take_largest(Set1) -> {Element, Set2}`
  [page 208] Extract largest element

• `take_smallest(Set1) -> {Element, Set2}`
  [page 209] Extract smallest element

• `to_list(Set) -> List`
  [page 209] Convert a gb_set into a list

• `union(Set1, Set2) -> Set3`
  [page 209] Return the union of two gb_sets

• `union(SetList) -> Set`
  [page 209] Return the union of a list of gb_sets

---

**gb_trees**

The following functions are exported:

• `balance(Tree1) -> Tree2`
  [page 210] Rebalance a tree

• `delete(Key, Tree1) -> Tree2`
  [page 210] Remove a node from a tree

• `delete_any(Key, Tree1) -> Tree2`
  [page 211] Remove a (possibly non-existing) node from a tree
empty() -> Tree
[page 211] Return an empty tree

eenter(Key, Val, Tree1) -> Tree2
[page 211] Insert or update key with value in a tree

from_oddict(List) -> Tree
[page 211] Make a tree from an orddict

get(Key, Tree) -> Val
[page 211] Look up a key in a tree, if present

lookup(Key, Tree) -> {value, Val} | none
[page 211] Look up a key in a tree

insert(Key, Val, Tree1) -> Tree2
[page 212] Insert a new key and value in a tree

is_defined(Key, Tree) -> bool()
[page 212] Test for membership of a tree

is_empty(Tree) -> bool()
[page 212] Test for empty tree

iterator(Tree) -> Iter
[page 212] Return an iterator for a tree

keys(Tree) -> [Key]
[page 212] Return a list of the keys in a tree

largest(Tree) -> {Key, Val}
[page 212] Return largest key and value

map(Function, Tree1) -> Tree2
[page 212] Return largest key and value

next(Iter1) -> {Key, Val, Iter2} | none
[page 213] Traverse a tree with an iterator

size(Tree) -> int()
[page 213] Return the number of nodes in a tree

smallest(Tree) -> {Key, Val}
[page 213] Return smallest key and value

take_largest(Tree1) -> {Key, Val, Tree2}
[page 213] Extract largest key and value

take_smallest(Tree1) -> {Key, Val, Tree2}
[page 213] Extract smallest key and value

to_list(Tree) -> [{Key, Val}]
[page 213] Convert a tree into a list

update(Key, Val, Tree1) -> Tree2
[page 214] Update a key to new value in a tree

values(Tree) -> [Val]
[page 214] Return a list of the values in a tree

gen_event

The following functions are exported:

start_link() -> Result
[page 216] Create a generic event manager process in a supervision tree.
The following functions are exported:

- `start_link(EventMgrName) -> Result`
  [page 216] Create a generic event manager process in a supervision tree.
- `start() -> Result`
  [page 216] Create a stand-alone event manager process.
- `start(EventMgrName) -> Result`
  [page 216] Create a stand-alone event manager process.
- `add_handler(EventMgrRef, Handler, Args) -> Result`
  [page 217] Add an event handler to a generic event manager.
- `add_sup_handler(EventMgrRef, Handler, Args) -> Result`
  [page 217] Add a supervised event handler to a generic event manager.
- `notify(EventMgrRef, Event) -> ok`
  [page 218] Notify an event manager about an event.
- `sync_notify(EventMgrRef, Event) -> ok`
  [page 218] Notify an event manager about an event.
- `call(EventMgrRef, Handler, Request) -> Result`
  [page 218] Make a synchronous call to a generic event manager.
- `call(EventMgrRef, Handler, Request, Timeout) -> Result`
  [page 218] Make a synchronous call to a generic event manager.
- `delete_handler(EventMgrRef, Handler, Args) -> Result`
  [page 219] Delete an event handler from a generic event manager.
- `swap_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result`
  [page 219] Replace an event handler in a generic event manager.
- `swap_sup_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result`
  [page 220] Replace an event handler in a generic event manager.
- `which_handlers(EventMgrRef) -> [Handler]`
  [page 220] Return all event handlers installed in a generic event manager.
- `stop(EventMgrRef) -> ok`
  [page 221] Terminate a generic event manager.
- `Module:init(InitArgs) -> {ok,State} | {ok,State,hibernate}`
  [page 221] Initialize an event handler.
- `Module:handle_event(Event, State) -> Result`
  [page 221] Handle an event.
- `Module:handle_call(Request, State) -> Result`
  [page 222] Handle a synchronous request.
- `Module:handle_info(Info, State) -> Result`
  [page 222] Handle an incoming message.
- `Module:terminate(Arg, State) -> term()`
  [page 223] Clean up before deletion.
- `Module:code_change(OldVsn, State, Extra) -> {ok, NewState}`
  [page 223] Update the internal state during upgrade/downgrade.

### gen_fsm

The following functions are exported:
- start_link(Module, Args, Options) -> Result
  [page 226] Create a gen_fsm process in a supervision tree.
- start_link(FsmName, Module, Args, Options) -> Result
  [page 226] Create a gen_fsm process in a supervision tree.
- start(Module, Args, Options) -> Result
  [page 227] Create a stand-alone gen_fsm process.
- start(FsmName, Module, Args, Options) -> Result
  [page 227] Create a stand-alone gen_fsm process.
- send_event(FsmRef, Event) -> ok
  [page 227] Send an event asynchronously to a generic FSM.
- send_all_state_event(FsmRef, Event) -> ok
  [page 228] Send an event asynchronously to a generic FSM.
- sync_send_event(FsmRef, Event) -> Reply
  [page 228] Send an event synchronously to a generic FSM.
- sync_send_event(FsmRef, Event, Timeout) -> Reply
  [page 228] Send an event synchronously to a generic FSM.
- sync_send_all_state_event(FsmRef, Event) -> Reply
  [page 228] Send an event synchronously to a generic FSM.
- sync_send_all_state_event(FsmRef, Event, Timeout) -> Reply
  [page 228] Send an event synchronously to a generic FSM.
- reply(Callers, Reply) -> true
  [page 229] Send a reply to a caller.
- send_event_after(Time, Event) -> Ref
  [page 229] Send a delayed event internally in a generic FSM.
- start_timer(Time, Msg) -> Ref
  [page 229] Send a timeout event internally in a generic FSM.
- cancel_timer(Ref) -> RemainingTime | false
  [page 230] Cancel an internal timer in a generic FSM.
- enter_loop(Module, Options, StateName, StateData)
  [page 230] Enter the gen_fsm receive loop
- enter_loop(Module, Options, StateName, StateData, FsmName)
  [page 230] Enter the gen_fsm receive loop
- enter_loop(Module, Options, StateName, StateData, Timeout)
  [page 230] Enter the gen_fsm receive loop
- enter_loop(Module, Options, StateName, StateData, FsmName, Timeout)
  [page 230] Enter the gen_fsm receive loop
- Module:init(Args) -> Result
  [page 231] Initialize process and internal state name and state data.
- Module:StateName(Event, StateData) -> Result
  [page 231] Handle an asynchronous event.
- Module:handle_event(Event, StateName, StateData) -> Result
  [page 232] Handle an asynchronous event.
- Module:handle_event(Event, From, StateData) -> Result
  [page 232] Handle a synchronous event.
- Module:handle_sync_event(Event, From, StateName, StateData) -> Result
  [page 233] Handle a synchronous event.
• Module:handle_info(Info, StateName, StateData) -> Result
  [page 234] Handle an incoming message.
• Module:terminate(Reason, StateName, StateData)
  [page 234] Clean up before termination.
• Module:code_change(OldVsn, StateName, StateData, Extra) -> {ok, NextStateName, NewStateData}
  [page 235] Update the internal state data during upgrade/downgrade.

**gen_server**

The following functions are exported:

• start_link(Module, Args, Options) -> Result
  [page 237] Create a gen_server process in a supervision tree.
• start_link(ServerName, Module, Args, Options) -> Result
  [page 237] Create a gen_server process in a supervision tree.
• start(Module, Args, Options) -> Result
  [page 238] Create a stand-alone gen_server process.
• start(ServerName, Module, Args, Options) -> Result
  [page 238] Create a stand-alone gen_server process.
• call(ServerRef, Request) -> Reply
  [page 238] Make a synchronous call to a generic server.
• call(ServerRef, Request, Timeout) -> Reply
  [page 238] Make a synchronous call to a generic server.
• multi_call(Name, Request) -> Result
  [page 239] Make a synchronous call to several generic servers.
• multi_call(Nodes, Name, Request) -> Result
  [page 239] Make a synchronous call to several generic servers.
• multi_call(Nodes, Name, Request, Timeout) -> Result
  [page 239] Make a synchronous call to several generic servers.
• cast(ServerRef, Request) -> ok
  [page 240] Send an asynchronous request to a generic server.
• abcast(Name, Request) -> abcast
  [page 240] Send an asynchronous request to several generic servers.
• abcast(Nodes, Name, Request) -> abcast
  [page 240] Send an asynchronous request to several generic servers.
• reply(Client, Reply) -> Result
  [page 240] Send a reply to a client.
• enter_loop(Module, Options, State)
  [page 241] Enter the gen_server receive loop.
• enter_loop(Module, Options, State, ServerName)
  [page 241] Enter the gen_server receive loop.
• enter_loop(Module, Options, State, Timeout)
  [page 241] Enter the gen_server receive loop.
• enter_loop(Module, Options, State, ServerName, Timeout)
  [page 241] Enter the gen_server receive loop.
- Module:init(Args) -> Result
  [page 242] Initialize process and internal state.
- Module:handle_call(Request, From, State) -> Result
  [page 242] Handle a synchronous request.
- Module:handle_cast(Request, State) -> Result
  [page 243] Handle an asynchronous request.
- Module:handle_info(Info, State) -> Result
  [page 243] Handle an incoming message.
- Module:terminate(Reason, State)
  [page 244] Clean up before termination.
- Module:code_change(OldVsn, State, Extra) -> {ok, NewState}
  [page 244] Update the internal state during upgrade/downgrade.

io

The following functions are exported:

- columns([IoDevice]) -> {ok, int()} | {error, enotsup}
  [page 247] Get the number of columns of a device
- put_chars([IoDevice,] IoData) -> ok
  [page 247] Write a list of characters
- nl([IoDevice]) -> ok
  [page 247] Write a newline
- get_chars([IoDevice,] Prompt, Count) -> Data | eof
  [page 247] Read a specified number of characters
- get_line([IoDevice,] Prompt) -> Data | eof | {error, Reason}
  [page 247] Read a line
- getopts([IoDevice]) -> Opt
  [page 248] Get the supported options and values from an I/O-server
- setopts([IoDevice,] Opt) -> ok | {error, Reason}
  [page 248] Set options
- write([IoDevice,] Term) -> ok
  [page 250] Write a term
- read([IoDevice,] Prompt) -> Result
  [page 250] Read a term
- read(IoDevice, Prompt, StartLine) -> Result
  [page 250] Read a term
- fwrite(Format) ->
  [page 251] Write formatted output
- fwrite([IoDevice,] Format, Data) -> ok
  [page 251] Write formatted output
- format(Format) ->
  [page 251] Write formatted output
- format([IoDevice,] Format, Data) -> ok
  [page 251] Write formatted output
- fread([IoDevice,] Prompt, Format) -> Result
  [page 255] Read formatted input
- `rows([IoDevice]) -> {ok,int()} | {error, enotsup}`
  [page 257] Get the number of rows of a device
- `scan ErlExprs(Prompt) ->`  
  [page 257] Read and tokenize Erlang expressions
- `scan ErlExprs([IoDevice,] Prompt, StartLine) -> Result`  
  [page 257] Read and tokenize Erlang expressions
- `scan ErlForm(Prompt) ->`  
  [page 257] Read and tokenize an Erlang form
- `scan ErlForm([IoDevice,] Prompt, StartLine) -> Result`  
  [page 257] Read and tokenize an Erlang form
- `parse ErlExprs(Prompt) ->`  
  [page 258] Read, tokenize and parse Erlang expressions
- `parse ErlExprs([IoDevice,] Prompt, StartLine) -> Result`  
  [page 258] Read, tokenize and parse Erlang expressions
- `parse ErlForm(Prompt) ->`  
  [page 258] Read, tokenize and parse an Erlang form
- `parse ErlForm([IoDevice,] Prompt, StartLine) -> Result`  
  [page 258] Read, tokenize and parse an Erlang form

**io_lib**

The following functions are exported:

- `nl() -> chars()`  
  [page 260] Write a newline
- `write(Term) ->`  
  [page 260] Write a term
- `write(Term, Depth) -> chars()`  
  [page 260] Write a term
- `print(Term) ->`  
  [page 260] Pretty print a term
- `print(Term, Column, LineLength, Depth) -> chars()`  
  [page 260] Pretty print a term
- `fwrite(Format, Data) ->`  
  [page 261] Write formatted output
- `format(Format, Data) -> chars() | UnicodeList`  
  [page 261] Write formatted output
- `fread(Format, String) -> Result`  
  [page 261] Read formatted input
- `fread(Continuation, String, Format) -> Return`  
  [page 262] Re-entrant formatted reader
- `write atom(Atom) -> chars()`  
  [page 262] Write an atom
- `write string(String) -> chars()`  
  [page 262] Write a string
- `write char(Integer) -> chars()`  
  [page 262] Write a character
- `indentation(String, StartIndent) -> int()
  [page 263] Indentation after printing string
- `char_list(Term) -> bool()
  [page 263] Test for a list of characters
- `deep_char_list(Term) -> bool()
  [page 263] Test for a deep list of characters
- `printable_list(Term) -> bool()
  [page 263] Test for a list of printable ISO-latin-1 characters

lib

The following functions are exported:
- `flush_receive() -> void()
  [page 264] Flush messages
- `error_message(Format, Args) -> ok
  [page 264] Print error message
- `progname() -> atom()
  [page 264] Return name of Erlang start script
- `nonl(String1) -> String2
  [page 264] Remove last newline
- `send(To, Msg)
  [page 264] Send a message
- `sendw(To, Msg)
  [page 265] Send a message and wait for an answer

lists

The following functions are exported:
- `all(Pred, List) -> bool()
  [page 266] Return true if all elements in the list satisfyPred
- `any(Pred, List) -> bool()
  [page 266] Return true if any of the elements in the list satisfiesPred
- `append(ListOfLists) -> List1
  [page 266] Append a list of lists
- `append(List1, List2) -> List3
  [page 267] Append two lists
- `concat(Things) -> string()
  [page 267] Concatenate a list of atoms
- `delete(Elem, List1) -> List2
  [page 267] Delete an element from a list
- `dropwhile(Pred, List1) -> List2
  [page 267] Drop elements from a list while a predicate is true
- `duplicate(N, Elem) -> List
  [page 267] Make N copies of element
- `filter(Pred, List1) -> List2
  [page 268] Choose elements which satisfy a predicate
- `flatlength(DeepList)` $\rightarrow$ `int()`
  [Page 268] Length of flattened deep list
- `flatMap(Fun, List1)` $\rightarrow$ `List2`
  [Page 268] Map and flatten in one pass
- `flatten(DeepList)` $\rightarrow$ `List`
  [Page 268] Flatten a deep list
- `flatten(DeepList, Tail)` $\rightarrow$ `List`
  [Page 268] Flatten a deep list
- `foldl(Fun, Acc0, List)` $\rightarrow$ `Acc1`
  [Page 269] Fold a function over a list
- `foldr(Fun, Acc0, List)` $\rightarrow$ `Acc1`
  [Page 269] Fold a function over a list
- `foreach(Fun, List)` $\rightarrow$ `void()`
  [Page 269] Apply a function to each element of a list
- `keydelete(Key, N, TupleList1)` $\rightarrow$ `TupleList2`
  [Page 269] Delete an element from a list of tuples
- `keyfind(Key, N, TupleList)` $\rightarrow$ `Tuple | false`
  [Page 270] Search for an element in a list of tuples
- `keymap(Fun, N, TupleList1)` $\rightarrow$ `TupleList2`
  [Page 270] Map a function over a list of tuples
- `keymember(Key, N, TupleList)` $\rightarrow$ `bool()`
  [Page 270] Test for membership of a list of tuples
- `keymerge(N, TupleList1, TupleList2)` $\rightarrow$ `TupleList3`
  [Page 270] Merge two key-sorted lists of tuples
- `keyreplace(Key, N, TupleList1, NewTuple)` $\rightarrow$ `TupleList2`
  [Page 271] Replace an element in a list of tuples
- `keysearch(Key, N, TupleList)` $\rightarrow$ `{value, Tuple} | false`
  [Page 271] Search for an element in a list of tuples
- `keysort(N, TupleList1)` $\rightarrow$ `TupleList2`
  [Page 271] Sort a list of tuples
- `keystore(Key, N, TupleList1, NewTuple)` $\rightarrow$ `TupleList2`
  [Page 271] Store an element in a list of tuples
- `keytake(Key, N, TupleList1)` $\rightarrow$ `{value, Tuple, TupleList2} | false`
  [Page 272] Extract an element from a list of tuples
- `last(List)` $\rightarrow$ `Last`
  [Page 272] Return last element in a list
- `map(Fun, List1)` $\rightarrow$ `List2`
  [Page 272] Map a function over a list
- `mapfoldl(Fun, Acc0, List)` $\rightarrow$ `{List2, Acc1}`
  [Page 272] Map and fold in one pass
- `mapfoldr(Fun, Acc0, List)` $\rightarrow$ `{List2, Acc1}`
  [Page 273] Map and fold in one pass
- `max(List)` $\rightarrow$ `Max`
  [Page 273] Return maximum element of a list
- `member(Elem, List)` $\rightarrow$ `bool()`
  [Page 273] Test for membership of a list
• merge(ListOfLists) -> List1  
  [page 273] Merge a list of sorted lists
• merge(List1, List2) -> List3  
  [page 273] Merge two sorted lists
• merge(List1, List2) -> List3  
  [page 273] Merge two sorted list
• merge(List1, List2, List3) -> List4  
  [page 274] Merge three sorted lists
• min(List) -> Min  
  [page 274] Return minimum element of a list
• nth(N, List) -> Elem  
  [page 274] Return the Nth element of a list
• nthtail(N, List1) -> Tail  
  [page 274] Return the Nth tail of a list
• partition(Pred, List) -> {Satisfying, NonSatisfying}  
  [page 275] Partition a list into two lists based on a predicate
• prefix(List1, List2) -> bool()  
  [page 275] Test for list prefix
• reverse(List1) -> List2  
  [page 275] Reverse a list
• reverse(List1, Tail) -> List2  
  [page 275] Reverse a list appending a tail
• seq(From, To) -> Seq  
  [page 275] Generate a sequence of integers
• seq(From, To, Incr) -> Seq  
  [page 275] Generate a sequence of integers
• sort(List1) -> List2  
  [page 276] Sort a list
• sort(Fun, List1) -> List2  
  [page 276] Sort a list
• split(N, List1) -> {List2, List3}  
  [page 276] Split a list into two lists
• splitwith(Pred, List) -> {List1, List2}  
  [page 276] Split a list into two lists based on a predicate
• sublist(List1, Len) -> List2  
  [page 277] Return a sub-list of a certain length, starting at the first position
• sublist(List1, Start, Len) -> List2  
  [page 277] Return a sub-list starting at a given position and with a given number of elements
• subtract(List1, List2) -> List3  
  [page 277] Subtract the element in one list from another list
• suffix(List1, List2) -> bool()  
  [page 278] Test for list suffix
• sum(List) -> number()  
  [page 278] Return sum of elements in a list
• takewhile(Pred, List1) -> List2  
  [page 278] Take elements from a list while a predicate is true
The following functions are exported:

- **init(Dir, MaxBytes, MaxFiles)**
  - [page 282] Initiate the event handler
- **init(Dir, MaxBytes, MaxFiles, Pred) -> Args**
  - [page 282] Initiate the event handler

The following functions are exported:

- **pi() -> float()**
  - [page 283] A useful number
- **sin(X)**
  - [page 283] Diverse math functions
- **cos(X)**
  - [page 283] Diverse math functions
The following functions are exported:

- `parse_transform(Forms, Options) -> Forms`
  [page 294] Transforms Erlang abstract format containing calls to ets/dbg:fun2ms into literal match specifications.

- `transform_from_shell(Dialect, Clauses, BoundEnvironment) -> term()`
  [page 294] Used when transforming fun's created in the shell into match specifications.

- `format_error(Errcode) -> ErrorMessage`
  [page 295] Error formatting function as required by the parse_transform interface.
orddict

The following functions are exported:

- `append(Key, Value, Orddict1) -> Orddict2`
  [page 296] Append a value to keys in a dictionary
- `append_list(Key, ValList, Orddict1) -> Orddict2`
  [page 296] Append new values to keys in a dictionary
- `erase(Key, Orddict1) -> Orddict2`
  [page 296] Erase a key from a dictionary
- `fetch(Key, Orddict) -> Value`
  [page 297] Look-up values in a dictionary
- `fetch_keys(Orddict) -> Keys`
  [page 297] Return all keys in a dictionary
- `filter(Pred, Orddict1) -> Orddict2`
  [page 297] Choose elements which satisfy a predicate
- `find(Key, Orddict) -> {ok, Value} | error`
  [page 297] Search for a key in a dictionary
- `fold(Fun, Acc0, Orddict) -> Acc1`
  [page 297] Fold a function over a dictionary
- `from_list(List) -> Orddict`
  [page 297] Convert a list of pairs to a dictionary
- `is_key(Key, Orddict) -> bool()`
  [page 298] Test if a key is in a dictionary
- `map(Fun, Orddict1) -> Orddict2`
  [page 298] Map a function over a dictionary
- `merge(Fun, Orddict1, Orddict2) -> Orddict3`
  [page 298] Merge two dictionaries
- `new() -> ordered_dictionary()`
  [page 298] Create a dictionary
- `size(Orddict) -> int()`
  [page 298] Return the number of elements in an ordered dictionary
- `store(Key, Value, Orddict1) -> Orddict2`
  [page 298] Store a value in a dictionary
- `to_list(Orddict) -> List`
  [page 299] Convert a dictionary to a list of pairs
- `update(Key, Fun, Orddict1) -> Orddict2`
  [page 299] Update a value in a dictionary
- `update(Key, Fun, Initial, Orddict1) -> Orddict2`
  [page 299] Update a value in a dictionary
- `update_counter(Key, Increment, Orddict1) -> Orddict2`
  [page 299] Increment a value in a dictionary
ordsets

The following functions are exported:

- **new()** -> Ordset  
  [page 301] Return an empty set
- **is_set(Ordset)** -> bool()  
  [page 301] Test for an Ordset
- **size(Ordset)** -> int()  
  [page 301] Return the number of elements in a set
- **to_list(Ordset)** -> List  
  [page 301] Convert an Ordset into a list
- **from_list(List)** -> Ordset  
  [page 302] Convert a list into an Ordset
- **is_element(Element, Ordset)** -> bool()  
  [page 302] Test for membership of an Ordset
- **add_element(Element, Ordset1)** -> Ordset2  
  [page 302] Add an element to an Ordset
- **del_element(Element, Ordset1)** -> Ordset2  
  [page 302] Remove an element from an Ordset
- **union(Ordset1, Ordset2)** -> Ordset3  
  [page 302] Return the union of two Ordsets
- **union(OrdsetList)** -> Ordset  
  [page 302] Return the union of a list of Ordsets
- **intersection(Ordset1, Ordset2)** -> Ordset3  
  [page 302] Return the intersection of two Ordsets
- **intersection(OrdsetList)** -> Ordset  
  [page 303] Return the intersection of a list of Ordsets
- **is_disjoint(Ordset1, Ordset2)** -> bool()  
  [page 303] Check whether two Ordsets are disjoint
- **subtract(Ordset1, Ordset2)** -> Ordset3  
  [page 303] Return the difference of two Ordsets
- **is_subset(Ordset1, Ordset2)** -> bool()  
  [page 303] Test for subset
- **fold(Function, Acc0, Ordset)** -> Acc1  
  [page 303] Fold over set elements
- **filter(Pred, Ordset1)** -> Set2  
  [page 303] Filter set elements

pg

The following functions are exported:

- **create(PgName)** -> ok | {error, Reason}  
  [page 304] Create an empty group
- **create(PgName, Node)** -> ok | {error, Reason}  
  [page 304] Create an empty group on another node
- `join(PgName, Pid)` → Members
  [page 304] Join a pid to a process group
- `send(PgName, Msg)` → `void()`
  [page 305] Send a message to all members of a process group
- `esend(PgName, Msg)` → `void()`
  [page 305] Send a message to all members of a process group, except ourselves
- `members(PgName)` → Members
  [page 305] Return a list of all members of a process group

**pool**

The following functions are exported:

- `start(Name)` →
  [page 306] Start a new pool
- `start(Name, Args)` → Nodes
  [page 306] Start a new pool
- `attach(Node)` → `alreadyAttached | attached`
  [page 306] Ensure that a pool master is running
- `stop()` → `stopped`
  [page 307] Stop the pool and kill all the slave nodes
- `getNodes()` → Nodes
  [page 307] Return a list of the current member nodes of the pool
- `pspawn(Mod, Fun, Args)` → `pid()`
  [page 307] Spawn a process on the pool node with expected lowest future load
- `pspawn_link(Mod, Fun, Args)` → `pid()`
  [page 307] Spawn and link to a process on the pool node with expected lowest future load
- `getNode()` → `node()`
  [page 307] Return the node with the expected lowest future load

**proc_lib**

The following functions are exported:

- `spawn(Fun)` → `pid()`
  [page 308] Spawn a new process
- `spawn(Node, Fun)` → `pid()`
  [page 308] Spawn a new process
- `spawn(Module, Function, Args)` → `pid()`
  [page 308] Spawn a new process
- `spawn(Node, Module, Function, Args)` → `pid()`
  [page 308] Spawn a new process
- `spawn_link(Fun)` → `pid()`
  [page 308] Spawn and link to a new process.
- `spawn_link(Node, Fun)` → `pid()`
  [page 308] Spawn and link to a new process.
• spawn_link(Module, Function, Args) -> pid()
  [page 309] Spawn and link to a new process.
• spawn_link(Node, Module, Function, Args) -> pid()
  [page 309] Spawn and link to a new process.
• spawn_opt(Fun, SpawnOpts) -> pid()
  [page 309] Spawn a new process with given options.
• spawn_opt(Node, Fun, SpawnOpts) -> pid()
  [page 309] Spawn a new process with given options.
• spawn_opt(Module, Function, Args, SpawnOpts) -> pid()
  [page 309] Spawn a new process with given options.
• spawn_opt(Node, Module, Fun, Args, SpawnOpts) -> pid()
  [page 309] Spawn a new process with given options.
• start(Module, Function, Args) -> Ret
  [page 309] Start a new process synchronously.
• start(Module, Function, Args, Time) -> Ret
  [page 309] Start a new process synchronously.
• start(Module, Function, Args, Time, SpawnOpts) -> Ret
  [page 309] Start a new process synchronously.
• start_link(Module, Function, Args) -> Ret
  [page 309] Start a new process synchronously.
• start_link(Module, Function, Args, Time) -> Ret
  [page 309] Start a new process synchronously.
• start_link(Module, Function, Args, Time, SpawnOpts) -> Ret
  [page 309] Start a new process synchronously.
• init_ack(Parent, Ret) -> void()
  [page 310] Used by a process when it has started.
• init_ack(Ret) -> void()
  [page 310] Used by a process when it has started.
• format(CrashReport) -> string()
• initial_call(Process) -> {Module,Function,Args} | false
  [page 311] Extract the initial call of a proc_libspawned process.
• translate_initial_call(Process) -> {Module,Function,Arity} | Fun
  [page 311] Extract and translate the initial call of a proc_libspawned process.
• hibernate(Module, Function, Args)
  [page 312] Hibernate a process until a message is sent to it

proplists

The following functions are exported:

• append_values(Key, List) -> List
  [page 313]
• compact(List) -> List
  [page 313]
• delete(Key, List) -> List
  [page 313]
The following functions are exported:

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

qlc

The following functions are exported:

- `append(QHL)` -> QH
- `append(QH1, QH2)` -> QH3
- `cursor(QueryHandleOrList [, Options])` -> QueryCursor
- `delete_cursor(QueryCursor)` -> ok
- `eval(QueryHandleOrList [, Options])` -> Answers | Error
• `e(QueryHandleOrList [, Options])` -> `Answers`
  [page 327] Return all answers to a query.
• `fold(Function, Acc0, QueryHandleOrList [, Options])` -> `Acc1 | Error`
  [page 327] Fold a function over the answers to a query.
• `format_error(Error)` -> `Chars`
  [page 328] Return an English description of a an error tuple.
• `info(QueryHandleOrList [, Options])` -> `Info`
  [page 328] Return code describing a query handle.
• `keysort(KeyPos, QH1 [, SortOptions])` -> `QH2`
  [page 329] Return a query handle.
• `next_answers(QueryCursor [, NumberOfAnswers])` -> `Answers | Error`
  [page 330] Return some or all answers to a query.
• `q(QueryListComprehension [, Options])` -> `QueryHandle`
  [page 330] Return a handle for a query list comprehension.
• `sort(QH1 [, SortOptions])` -> `QH2`
  [page 333] Return a query handle.
• `string_to_handle(QueryString [, Options [, Bindings]]])` -> `QueryHandle | Error`
  [page 333] Return a handle for a query list comprehension.
• `table(TraverseFun, Options)` -> `QueryHandle`
  [page 333] Return a query handle for a table.

queue

The following functions are exported:

• `new()` -> `Q`
  [page 338] Create an empty queue
• `is_queue(Term)` -> `true | false`
  [page 338] Test if a term is a queue
• `is_empty(Q)` -> `true | false`
  [page 338] Test if a queue is empty
• `len(Q)` -> `N`
  [page 338] Get the length of a queue
• `in(Item, Q1)` -> `Q2`
  [page 338] Insert an item at the rear of a queue
• `in_r(Item, Q1)` -> `Q2`
  [page 338] Insert an item at the front of a queue
• `out(Q1)` -> `Result`
  [page 338] Remove the front item from a queue
• `out_r(Q1)` -> `Result`
  [page 339] Remove the rear item from a queue
• `from_list(L)` -> `queue()
  [page 339] Convert a list to a queue
• `to_list(Q)` -> `list()`
  [page 339] Convert a queue to a list
• reverse(Q1) -> Q2
  [page 339] Reverse a queue
• split(N, Q1) -> {Q2,Q3}
  [page 339] Split a queue in two
• join(Q1, Q2) -> Q3
  [page 339] Join two queues
• filter(Fun, Q1) -> Q2
  [page 339] Filter a queue
• member(Item, Q) -> bool()
  [page 340] Test if an item is in a queue
• get(Q) -> Item
  [page 340] Return the front item of a queue
• get_r(Q) -> Item
  [page 340] Return the rear item of a queue
• drop(Q1) -> Q2
  [page 340] Remove the front item from a queue
• drop_r(Q1) -> Q2
  [page 340] Remove the rear item from a queue
• peek(Q) -> {value,Item} | empty
  [page 341] Return the front item of a queue
• peek_r(Q) -> {value,Item} | empty
  [page 341] Return the rear item of a queue
• cons(Item, Q1) -> Q2
  [page 341] Insert an item at the head of a queue
• head(Q) -> Item
  [page 341] Return the item at the head of a queue
• tail(Q1) -> Q2
  [page 341] Remove the head item from a queue
• snoc(Q1, Item) -> Q2
  [page 342] Insert an item at the tail of a queue
• daeh(Q) -> Item
  [page 342] Return the tail item of a queue
• last(Q) -> Item
  [page 342] Return the tail item of a queue
• lait(Q1) -> Q2
  [page 342] Remove the tail item from a queue
• init(Q1) -> Q2
  [page 342] Remove the tail item from a queue
• lait(Q1) -> Q2
  [page 342] Remove the tail item from a queue

random

The following functions are exported:
• seed() -> ran()
  [page 343] Seeds random number generation with default values
The following functions are exported:

- **compile(Regexp) -> {ok, MP} | {error, ErrSpec}**
  [page 346] Compile a regular expression into a match program
- **compile(Regexp,Options) -> {ok, MP} | {error, ErrSpec}**
  [page 346] Compile a regular expression into a match program
- **run(Subject,RE) -> {match, Captured} | nomatch**
  [page 348] Match a subject against regular expression and capture subpatterns
- **run(Subject,RE,Options) -> {match, Captured} | match | nomatch**
  [page 348] Match a subject against regular expression and capture subpatterns
- **replace(Subject,RE,Replacement) -> iodata() | charlist()**
  [page 353] Match a subject against regular expression and replace matching elements with Replacement
- **replace(Subject,RE,Replacement,Options) -> iodata() | charlist() | binary() | list()**
  [page 353] Match a subject against regular expression and replace matching elements with Replacement
- **split(Subject,RE) -> SplitList**
  [page 354] Split a string by tokens specified as a regular expression
- **split(Subject,RE,Options) -> SplitList**
  [page 354] Split a string by tokens specified as a regular expression

**regexp**

The following functions are exported:

- **match(String, RegExp) -> MatchRes**
  [page 392] Match a regular expression
- **first_match(String, RegExp) -> MatchRes**
  [page 392] Match a regular expression
- **matches(String, RegExp) -> MatchRes**
  [page 393] Match a regular expression
• `sub(String, RegExp, New) -> SubRes`
  [page 393] Substitute the first occurrence of a regular expression
• `gsub(String, RegExp, New) -> SubRes`
  [page 393] Substitute all occurrences of a regular expression
• `split(String, RegExp) -> SplitRes`
  [page 393] Split a string into fields
• `sh_to_awk(ShRegExp) -> AwkRegExp`
  [page 394] Convert an shregular expression into an AWK one
• `parse(RegExp) -> ParseRes`
  [page 394] Parse a regular expression
• `format_error(Descriptor) -> Error`
  [page 394] Format an error descriptor

sets

The following functions are exported:

• `new() -> Set`
  [page 397] Return an empty set
• `is_set(Set) -> bool()`
  [page 397] Test for an Set
• `size(Set) -> int()`
  [page 397] Return the number of elements in a set
• `to_list(Set) -> List`
  [page 397] Convert an Set into a list
• `from_list(List) -> Set`
  [page 398] Convert a list into an Set
• `is_element(Element, Set) -> bool()`
  [page 398] Test for membership of an Set
• `add_element(Element, Set1) -> Set2`
  [page 398] Add an element to an Set
• `del_element(Element, Set1) -> Set2`
  [page 398] Remove an element from an Set
• `union(Set1, Set2) -> Set3`
  [page 398] Return the union of two Sets
• `union(SetList) -> Set`
  [page 398] Return the union of a list of Sets
• `intersection(Set1, Set2) -> Set3`
  [page 398] Return the intersection of two Sets
• `intersection(SetList) -> Set`
  [page 399] Return the intersection of a list of Sets
• `is_disjoint(Set1, Set2) -> bool()`
  [page 399] Check whether two Sets are disjoint
• `subtract(Set1, Set2) -> Set3`
  [page 399] Return the difference of two Sets
• `is_subset(Set1, Set2) -> bool()`
  [page 399] Test for subset
- `fold(Function, Acc0, Set) -> Acc1`
  [page 399] Fold over set elements
- `filter(Pred, Set1) -> Set2`
  [page 399] Filter set elements

**shell**

The following functions are exported:

- `history(N) -> integer()`
  [page 409] Sets the number of previous commands to keep
- `results(N) -> integer()`
  [page 409] Sets the number of previous results to keep
- `catch_exception(Bool) -> Bool`
  [page 409] Sets the exception handling of the shell
- `start_restricted(Module) -> ok | {error, Reason}`
  [page 409] Exits a normal shell and starts a restricted shell.
- `stop_restricted() -> ok`
  [page 410] Exits a restricted shell and starts a normal shell.

**shell_default**

No functions are exported.

**slave**

The following functions are exported:

- `start(Host) ->`
  [page 412] Start a slave node on a host
- `start(Host, Name) ->`
  [page 412] Start a slave node on a host
- `start(Host, Name, Args) -> {ok, Node} | {error, Reason}`
  [page 412] Start a slave node on a host
- `start_link(Host) ->`
  [page 413] Start and link to a slave node on a host
- `start_link(Host, Name) ->`
  [page 413] Start and link to a slave node on a host
- `start_link(Host, Name, Args) -> {ok, Node} | {error, Reason}`
  [page 413] Start and link to a slave node on a host
- `stop(Node) -> ok`
  [page 414] Stop (kill) a node
- `pseudo([Master | ServerList]) -> ok`
  [page 414] Start a number of pseudo servers
- `pseudo(Master, ServerList) -> ok`
  [page 414] Start a number of pseudo servers
- `relay(Pid)`
  [page 414] Run a pseudo server
The following functions are exported:

- `a_function(Tuples [, Type])` -> `Function
  [page 419] Create a function.

- `canonical_relation(SetOfSets)` -> `BinRel
  [page 419] Return the canonical map.

- `composite(Function1, Function2)` -> `Function3
  [page 419] Return the composite of two functions.

- `constant_function(Set, AnySet)` -> `Function
  [page 419] Create the function that maps each element of a set onto another set.

- `converse(BinRel1)` -> `BinRel2
  [page 420] Return the converse of a binary relation.

- `difference(Set1, Set2)` -> `Set3
  [page 420] Return the difference of two sets.

- `digraph_to_family(Graph [, Type])` -> `Family
  [page 420] Create a family from a directed graph.

- `domain(BinRel)` -> `Set
  [page 420] Return the domain of a binary relation.

- `drestriction(BinRel1, Set)` -> `BinRel2
  [page 420] Return a restriction of a binary relation.

- `drestriction(SetFun, Set1, Set2)` -> `Set3
  [page 421] Return a restriction of a relation.

- `empty_set()` -> `Set
  [page 421] Return the untyped empty set.

- `extension(BinRel1, Set, AnySet)` -> `BinRel2
  [page 421] Extend the domain of a binary relation.

- `family(Tuples [, Type])` -> `Family
  [page 422] Create a family of subsets.

- `family_difference(Family1, Family2)` -> `Family3
  [page 422] Return the difference of two families.

- `family_domain(Family1)` -> `Family2
  [page 422] Return a family of domains.

- `family_field(Family1)` -> `Family2
  [page 422] Return a family of fields.

- `family_intersection(Family1)` -> `Family2
  [page 423] Return the intersection of a family of sets of sets.

- `family_intersection(Family1, Family2)` -> `Family3
  [page 423] Return the intersection of two families.

- `family_projection(SetFun, Family1)` -> `Family2
  [page 423] Return a family of modified subsets.

- `family_range(Family1)` -> `Family2
  [page 423] Return a family of ranges.

- `family_specification(Fun, Family1)` -> `Family2
  [page 424] Select a subset of a family using a predicate.
- `family_to_digraph(Family [, GraphType]) -> Graph`
  [page 424] Create a directed graph from a family.
- `family_to_relation(Family) -> BinRel`
  [page 424] Create a binary relation from a family.
- `family_union(Family1) -> Family2`
  [page 425] Return the union of a family of sets of sets.
- `family_union(Family1, Family2) -> Family3`
  [page 425] Return the union of two families.
- `field(BinRel) -> Set`
  [page 425] Return the field of a binary relation.
- `from_external(ExternalSet, Type) -> AnySet`
  [page 425] Create a set.
- `from_sets(ListOfSets) -> Set`
  [page 425] Create a set out of a list of sets.
- `from_sets(TupleOfSets) -> Ordset`
  [page 425] Create an ordered set out of a tuple of sets.
- `from_term(Term [, Type]) -> AnySet`
  [page 426] Create a set.
- `image(BinRel, Set1) -> Set2`
  [page 427] Return the image of a set under a binary relation.
- `intersection(SetOfSets) -> Set`
  [page 427] Return the intersection of a set of sets.
- `intersection(Set1, Set2) -> Set3`
  [page 427] Return the intersection of two sets.
- `intersection_of_family(Family) -> Set`
  [page 427] Return the intersection of a family.
- `inverse(Function1) -> Function2`
  [page 427] Return the inverse of a function.
- `inverse_image(BinRel, Set1) -> Set2`
  [page 428] Return the inverse image of a set under a binary relation.
- `is_a_function(BinRel) -> Bool`
  [page 428] Test for a function.
- `is_disjoint(Set1, Set2) -> Bool`
  [page 428] Test for disjoint sets.
- `is_empty_set(AnySet) -> Bool`
  [page 428] Test for an empty set.
- `is_equal(AnySet1, AnySet2) -> Bool`
  [page 428] Test two sets for equality.
- `is_set(AnySet) -> Bool`
  [page 428] Test for an unordered set.
- `is_sofs_set(Term) -> Bool`
  [page 429] Test for a type.
- `is_subset(Set1, Set2) -> Bool`
  [page 429] Test two sets for subset.
- `is_type(Term) -> Bool`
  [page 429] Test for a type.
\begin{itemize}
  \item \texttt{join(Relation1, I, Relation2, J) -> Relation3} [page 429] Return the join of two relations.
  \item \texttt{multiple_relative_product(TupleOfBinRels, BinRel1) -> BinRel2} [page 429] Return the multiple relative product of a tuple of binary relations and a relation.
  \item \texttt{no_elements(ASet) -> NoElements} [page 430] Return the number of elements of a set.
  \item \texttt{partition(SetOfSets) -> Partition} [page 430] Return the coarsest partition given a set of sets.
  \item \texttt{partition(SetFun, Set) -> Partition} [page 430] Return a partition of a set.
  \item \texttt{partition(SetFun, Set1, Set2) -> \{Set3, Set4\}} [page 430] Return a partition of a set.
  \item \texttt{partition_family(SetFun, Set) -> Family} [page 431] Return a family indexing a partition.
  \item \texttt{product(TupleOfSets) -> Relation} [page 431] Return the Cartesian product of a tuple of sets.
  \item \texttt{product(Set1, Set2) -> BinRel} [page 431] Return the Cartesian product of two sets.
  \item \texttt{projection(SetFun, Set1) -> Set2} [page 432] Return a set of substituted elements.
  \item \texttt{range(BinRel) -> Set} [page 432] Return the range of a binary relation.
  \item \texttt{relation(Tuples [, Type]) -> Relation} [page 432] Create a relation.
  \item \texttt{relation_to_family(BinRel) -> Family} [page 432] Create a family from a binary relation.
  \item \texttt{relative_product(TupleOfBinRels [, BinRel1]) -> BinRel2} [page 433] Return the relative product of a tuple of binary relations and a binary relation.
  \item \texttt{relative_product(BinRel1, BinRel2) -> BinRel3} [page 433] Return the relative product of two binary relations.
  \item \texttt{relative_product1(BinRel1, BinRel2) -> BinRel3} [page 433] Return the relative product of two binary relations.
  \item \texttt{restriction(BinRel1, Set) -> BinRel2} [page 433] Return a restriction of a binary relation.
  \item \texttt{restriction(SetFun, Set1, Set2) -> Set3} [page 434] Return a restriction of a set.
  \item \texttt{set(Terms [, Type]) -> Set} [page 434] Create a set of atoms or any type of sets.
  \item \texttt{specification(Fun, Set1) -> Set2} [page 434] Select a subset using a predicate.
  \item \texttt{strict_relation(BinRel1) -> BinRel2} [page 434] Return the strict relation corresponding to a given relation.
  \item \texttt{substitution(SetFun, Set1) -> Set2} [page 435] Return a function with a given set as domain.
\end{itemize}
• `symdiff(Set1, Set2) -> Set3` [page 435] Return the symmetric difference of two sets.
• `symmetric_partition(Set1, Set2) -> {Set3, Set4, Set5}` [page 436] Return a partition of two sets.
• `to_external(AnySet) -> ExternalSet` [page 436] Return the elements of a set.
• `to_sets(ASet) -> Sets` [page 436] Return a list or a tuple of the elements of set.
• `type(AnySet) -> Type` [page 436] Return the type of a set.
• `union(SetOfSets) -> Set` [page 436] Return the union of a set of sets.
• `union(Set1, Set2) -> Set3` [page 436] Return the union of two sets.
• `union_of_family(Family) -> Set` [page 437] Return the union of a family.
• `weak_relation(BinRel1) -> BinRel2` [page 437] Return the weak relation corresponding to a given relation.

string
The following functions are exported:

• `len(String) -> Length` [page 438] Return the length of a string
• `equal(String1, String2) -> bool()` [page 438] Test string equality
• `concat(String1, String2) -> String3` [page 438] Concatenate two strings
• `chr(String, Character) -> Index` [page 438] Return the index of the first/last occurrence of Character in String
• `rchr(String, Character) -> Index` [page 438] Return the index of the first/last occurrence of Character in String
• `str(String, SubString) -> Index` [page 438] Find the index of a substring
• `rstr(String, SubString) -> Index` [page 438] Find the index of a substring
• `span(String, Chars) -> Length` [page 439] Span characters at start of string
• `cspan(String, Chars) -> Length` [page 439] Span characters at start of string
• `substr(String, Start) -> SubString` [page 439] Return a substring of String
• `substr(String, Start, Length) -> Substring` [page 439] Return a substring of String
• `tokens(String, SeparatorList) -> Tokens` [page 439] Split string into tokens
- join(StringList, Separator) -> String
  [page 439] Join a list of strings with separator

- chars(Character, Number) -> String
  [page 440] Returns a string consisting of numbers of characters

- chars(Character, Number, Tail) -> String
  [page 440] Returns a string consisting of numbers of characters

- copies(String, Number) -> Copies
  [page 440] Copy a string

- words(String) -> Count
  [page 440] Count blank separated words

- words(String, Character) -> Count
  [page 440] Count blank separated words

- sub_word(String, Number) -> Word
  [page 440] Extract subword

- sub_word(String, Number, Character) -> Word
  [page 440] Extract subword

- strip(String) -> Stripped
  [page 441] Strip leading or trailing characters

- strip(String, Direction) -> Stripped
  [page 441] Strip leading or trailing characters

- strip(String, Direction, Character) -> Stripped
  [page 441] Strip leading or trailing characters

- left(String, Number) -> Left
  [page 441] Adjust left end of string

- left(String, Number, Character) -> Left
  [page 441] Adjust left end of string

- right(String, Number) -> Right
  [page 441] Adjust right end of string

- right(String, Number, Character) -> Right
  [page 441] Adjust right end of string

- centre(String, Number) -> Centered
  [page 441] Center a string

- centre(String, Number, Character) -> Centered
  [page 441] Center a string

- sub_string(String, Start) -> SubString
  [page 442] Extract a substring

- sub_string(String, Start, Stop) -> SubString
  [page 442] Extract a substring

- to_float(String) -> {Float,Rest} | {error,Reason}
  [page 442] Returns a float whose text representation is the integers (ASCII values) in String.

- to_integer(String) -> {Int,Rest} | {error,Reason}
  [page 442] Returns an integer whose text representation is the integers (ASCII values) in String.

- to_lower(String) -> Result
  [page 443] Convert case of string (ISO/IEC 8859-1)
to_lower(Char) -> CharResult
[page 443] Convert case of string (ISO/IEC 8859-1)
to_upper(String) -> Result
[page 443] Convert case of string (ISO/IEC 8859-1)
to_upper(Char) -> CharResult
[page 443] Convert case of string (ISO/IEC 8859-1)

supervisor

The following functions are exported:

- start_link(Module, Args) -> Result
  [page 446] Create a supervisor process.
- start_link(SupName, Module, Args) -> Result
  [page 446] Create a supervisor process.
- start_child(SupRef, ChildSpec) -> Result
  [page 447] Dynamically add a child process to a supervisor.
- terminate_child(SupRef, Id) -> Result
  [page 447] Terminate a child process belonging to a supervisor.
- delete_child(SupRef, Id) -> Result
  [page 448] Delete a child specification from a supervisor.
- restart_child(SupRef, Id) -> Result
  [page 448] Restart a terminated child process belonging to a supervisor.
- which_children(SupRef) -> [{Id, Child, Type, Modules}]
  [page 449] Return information about all children specifications and child processes belonging to a supervisor.
- check_childspecs([ChildSpec]) -> Result
  [page 449] Check if child specifications are syntactically correct.
- Module:init(Args) -> Result
  [page 450] Return a supervisor specification.

supervisor_bridge

The following functions are exported:

- start_link(Module, Args) -> Result
  [page 451] Create a supervisor bridge process.
- start_link(SupBridgeName, Module, Args) -> Result
  [page 451] Create a supervisor bridge process.
- Module:init(Args) -> Result
  [page 452] Initialize process and start subsystem.
- Module:terminate(Reason, State)
  [page 452] Clean up and stop subsystem.
sys

The following functions are exported:

- `log(Name, Flag)`  
  [page 455] Log system events in memory
- `log(Name, Flag, Timeout) -> ok | {ok, [system_event()]}`  
  [page 455] Log system events in memory
- `log_to_file(Name, Flag)`  
  [page 455] Log system events to the specified file
- `log_to_file(Name, Flag, Timeout) -> ok | {error, open_file}`  
  [page 455] Log system events to the specified file
- `statistics(Name, Flag)`  
  [page 455] Enable or disable the collections of statistics
- `statistics(Name, Flag, Timeout) -> ok | {ok, Statistics}`  
  [page 455] Enable or disable the collections of statistics
- `trace(Name, Flag)`  
  [page 456] Print all system events on standard io
- `trace(Name, Flag, Timeout) -> void()`  
  [page 456] Print all system events on standard io
- `no_debug(Name)`  
  [page 456] Turn off debugging
- `no_debug(Name, Timeout) -> void()`  
  [page 456] Turn off debugging
- `suspend(Name)`  
  [page 456] Suspend the process
- `suspend(Name, Timeout) -> void()`  
  [page 456] Suspend the process
- `resume(Name)`  
  [page 456] Resume a suspended process
- `resume(Name, Timeout) -> void()`  
  [page 456] Resume a suspended process
- `change_code(Name, Module, OldVsn, Extra)`  
  [page 456] Send the code change system message to the process
- `change_code(Name, Module, OldVsn, Extra, Timeout) -> ok | {error, Reason}`  
  [page 456] Send the code change system message to the process
- `get_status(Name)`  
  [page 456] Get the status of the process
- `get_status(Name, Timeout) -> {status, Pid, {module, Mod}, [PDict, SysState, Parent, Dbg, Misc]}`  
  [page 456] Get the status of the process
- `install(Name, {Func, FuncState})`  
  [page 457] Install a debug function in the process
- `install(Name, {Func, FuncState}, Timeout)`  
  [page 457] Install a debug function in the process
- `remove(Name, Func)`  
  [page 457] Remove a debug function from the process
• \texttt{remove(Name,Func,Timeout) -> void()}
  [page 457] Remove a debug function from the process
• \texttt{debug_options(Options) -> [dbg_opt()]}  
  [page 458] Convert a list of options to a debug structure
• \texttt{get_debug(Item,Debug,Default) -> term()}
  [page 458] Get the data associated with a debug option
• \texttt{handle_debug([dbg_opt()],FormFunc,Extra,Event) -> [dbg_opt()]}  
  [page 458] Generate a system event
• \texttt{handle_system_msg(Msg,From,Parent,Module,Debug,Misc)}
  [page 458] Take care of system messages
• \texttt{print_log(Debug) -> void()}
  [page 459] Print the logged events in the debug structure
• \texttt{Mod:system\_continue(Parent, Debug, Misc)}
  [page 459] Called when the process should continue its execution
• \texttt{Mod:system\_terminate(Reason, Parent, Debug, Misc)}
  [page 459] Called when the process should terminate
• \texttt{Mod:system\_code\_change(Misc, Module, OldVsn, Extra) -> \{ok, NMisc\}}
  [page 459] Called when the process should perform a code change

**timer**

The following functions are exported:

• \texttt{start() -> ok}
  [page 461] Start a global timer server (named \texttt{timer\_server}).
• \texttt{apply\_after(Time, Module, Function, Arguments) -> \{ok, TRef\} | \{error, Reason\}}
• \texttt{send\_after(Time, Pid, Message) -> \{ok, TRef\} | \{error,Reason\}}
  [page 461] Send Message to Pid after a specified Time.
• \texttt{send\_after(Time, Message) -> \{ok, TRef\} | \{error, Reason\}}
  [page 461] Send Message to Pid after a specified Time.
• \texttt{exit\_after(Time, Pid, Reason1) -> \{ok, TRef\} | \{error,Reason2\}}
  [page 462] Send an exit signal with Reason1 after a specified Time.
• \texttt{exit\_after(Time, Reason1) -> \{ok, TRef\} | \{error,Reason2\}}
  [page 462] Send an exit signal with Reason1 after a specified Time.
• \texttt{kill\_after(Time, Pid) -> \{ok, TRef\} | \{error,Reason2\}}
  [page 462] Send an exit signal with Reason2 after a specified Time.
• \texttt{kill\_after(Time) -> \{ok, TRef\} | \{error,Reason2\}}
  [page 462] Send an exit signal with Reason2 after a specified Time.
• \texttt{apply\_interval(Time, Module, Function, Arguments) -> \{ok, TRef\} | \{error, Reason\}}
• \texttt{send\_interval(Time, Pid, Message) -> \{ok, TRef\} | \{error, Reason\}}
  [page 462] Send Message repeatedly at intervals of Time.
• \texttt{send\_interval(Time, Message) -> \{ok, TRef\} | \{error, Reason\}}
  [page 462] Send Message repeatedly at intervals of Time.
- `cancel(TRef)` \to \{ok, cancel\} | \{error, Reason\}

  [page 462] Cancel a previously requested timeout identified by TRef.

- `sleep(Time)` \to \ok

  [page 462] Suspend the calling process for Time amount of milliseconds.

- `tc(Module, Function, Arguments)` \to \{Time, Value\}

  [page 463] Measure the real time it takes to evaluate apply(Module, Function, Arguments)

- `now_diff(T2, T1)` \to Tdiff

  [page 463] Calculate time difference between now/0 timestamps

- `seconds(Seconds)` \to Milliseconds


- `minutes(Minutes)` \to Milliseconds

  [page 463] Converts Minutes to Milliseconds.

- `hours(Hours)` \to Milliseconds

  [page 463] Convert Hours to Milliseconds.

- `hms(Hours, Minutes, Seconds)` \to Milliseconds

  [page 463] Convert Hours+Minutes+Seconds to Milliseconds.

### unicode

The following functions are exported:

- `bom_to_encoding(Bin)` \to \{Encoding, Length\}

  [page 466] Identify UTF byte order marks in a binary.

- `characters_to_list(Data)` \to list() | \{error, list(), RestData\} | incomplete, list(), binary()

  [page 466] Convert a collection of characters to list of Unicode characters

- `characters_to_list(Data, InEncoding)` \to list() | \{error, list(), RestData\} | incomplete, list(), binary()

  [page 466] Convert a collection of characters to list of Unicode characters

- `characters_to_binary(Data)` \to binary() | \{error, binary(), RestData\} | incomplete, binary(), binary()

  [page 467] Convert a collection of characters to an UTF-8 binary

- `characters_to_binary(Data, InEncoding)` \to binary() | \{error, binary(), RestData\} | incomplete, binary(), binary()

  [page 468] Convert a collection of characters to an UTF-8 binary

- `characters_to_binary(Data, InEncoding, OutEncoding)` \to binary() | \{error, binary(), RestData\} | incomplete, binary(), binary()

  [page 468] Convert a collection of characters to an UTF-8 binary

- `encoding_to_bom(InEncoding)` \to Bin

  [page 468] Create a binary UTF byte order mark from encoding.

### win32reg

The following functions are exported:

- `change_key(RegHandle, Key)` \to ReturnValue

  [page 470] Move to a key in the registry
- `change_key_create(RegHandle, Key) -> ReturnValue`
  [page 470] Move to a key, create it if it is not there
- `close(RegHandle) -> ReturnValue`
  [page 470] Close the registry.
- `current_key(RegHandle) -> ReturnValue`
  [page 470] Return the path to the current key.
- `delete_key(RegHandle) -> ReturnValue`
  [page 470] Delete the current key.
- `delete_value(RegHandle, Name) -> ReturnValue`
  [page 471] Delete the named value on the current key.
- `expand(String) -> ExpandedString`
  [page 471] Expand a string with environment variables.
- `format_error(ErrorId) -> ErrorString`
  [page 471] Convert an POSIX errorcode to a string.
- `open(OpenModeList) -> ReturnValue`
  [page 471] Open the registry for reading or writing.
- `set_value(RegHandle, Name, Value) -> ReturnValue`
  [page 471] Set value at the current registry key with specified name.
- `sub_keys(RegHandle) -> ReturnValue`
  [page 472] Get subkeys to the current key.
- `value(RegHandle, Name) -> ReturnValue`
  [page 472] Get the named value on the current key.
- `values(RegHandle) -> ReturnValue`
  [page 472] Get all values on the current key.

**zip**

The following functions are exported:

- `zip(Name, FileList) -> ReturnValue`
  [page 474] Create a zip archive with options.
- `zip(Name, FileList, Options) -> ReturnValue`
  [page 474] Create a zip archive with options.
- `create(Name, FileList) -> ReturnValue`
  [page 474] Create a zip archive with options.
- `create(Name, FileList, Options) -> ReturnValue`
  [page 474] Create a zip archive with options.
- `unzip(Archive) -> ReturnValue`
  [page 475] Extract files from a zip archive.
- `unzip(Archive, Options) -> ReturnValue`
  [page 475] Extract files from a zip archive.
- `extract(Archive) -> ReturnValue`
  [page 475] Extract files from a zip archive.
- `extract(Archive, Options) -> ReturnValue`
  [page 475] Extract files from a zip archive.
- `list_dir(Archive) -> ReturnValue`
  [page 476] Retrieve the name of all files in a zip archive.
- `list_dir(Archive, Options)`
  [page 476] Retrieve the name of all files in a zip archive
- `table(Archive)` -> `RetVal`  
  [page 476] Retrieve the name of all files in a zip archive
- `table(Archive, Options)`  
  [page 476] Retrieve the name of all files in a zip archive
- `t(Archive)`  
  [page 477] Print the name of each file in a zip archive
- `tt(Archive)`  
  [page 477] Print name and information for each file in a zip archive
- `zip_open(Archive) -> {ok, ZipHandle} | {error, Reason}`  
  [page 477] Open an archive and return a handle to it
- `zip_open(Archive, Options) -> {ok, ZipHandle} | {error, Reason}`  
  [page 477] Open an archive and return a handle to it
- `zip_list_dir(ZipHandle) -> Result | {error, Reason}`  
  [page 477] Return a table of files in open zip archive
- `zip_get(ZipHandle) -> {ok, [Result]} | {error, Reason}`  
  [page 478] Extract files from an open archive
- `zip_get(FileName, ZipHandle) -> {ok, Result} | {error, Reason}`  
  [page 478] Extract files from an open archive
- `zip_close(ZipHandle) -> ok | {error, EINVAL}`  
  [page 478] Close an open archive
The STDLIB is mandatory in the sense that the minimal system based on Erlang/OTP consists of Kernel and STDLIB. The STDLIB application contains no services.

Configuration

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

- `shell_esc`: `icl | abort` This parameter can be used to alter the behaviour of the Erlang shell when `^G` is pressed.
- `restricted_shell`: `module()` This parameter can be used to run the Erlang shell in restricted mode.
- `shell_catch_exception`: `bool()` This parameter can be used to set the exception handling of the Erlang shell’s evaluator process.
- `shell_history_length`: `integer() >= 0` This parameter can be used to determine how many commands are saved by the Erlang shell.
- `shell_saved_results`: `integer() >= 0` This parameter can be used to determine how many results are saved by the Erlang shell.

See Also

[app(4)], [application(3)], shell(3) [page 400],
array

Erlang Module

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

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

Unless specified by the user when the array is created, the default value is the atom undefined. There is no difference between an unset entry and an entry which has been explicitly set to the same value as the default one (cf. reset/2 [page 84]). If you need to differentiate between unset and set entries, you must make sure that the default value cannot be confused with the values of set entries.

The array never shrinks automatically; if an index \( I \) has been used successfully to set an entry, all indices in the range \([0, I]\) will stay accessible unless the array size is explicitly changed by calling resize/2 [page 84].

Examples:

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

%% Create an extendible array and set entry 17 to 'true',
%% causing the array to grow automatically
A1 = array:set(17, true, array:new(10)).
18 = array:size(A1).

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

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

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

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

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

%% A fixed-size array does not grow automatically and does not
%% allow accesses beyond the last set entry
```
{'EXIT',[badarg,\_]} = (catch array:set(18, true, A3)).
{'EXIT',[badarg,\_]} = (catch array:get(18, A3)).

DATA TYPES

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

Exports

default(Array::array()) -> term()
Get the value used for uninitialized entries.
See also: new/2 [page 83].

fix(Array::array()) -> array()
Fix the size of the array. This prevents it from growing automatically upon insertion; see also set/3 [page 84].
See also: relax/1 [page 83].

foldl(Function, InitialAcc::term(), Array::array()) -> term()
Types:
• Function = (Index::integer(), Value::term(), Acc::term()) -> term()
Fold the elements of the array using the given function and initial accumulator value. The elements are visited in order from the lowest index to the highest. If Function is not a function, the call fails with reason badarg.
See also: foldr/3 [page 81], map/2 [page 82], sparse_foldl/3 [page 84].

foldr(Function, InitialAcc::term(), Array::array()) -> term()
Types:
• Function = (Index::integer(), Value::term(), Acc::term()) -> term()
Fold the elements of the array right-to-left using the given function and initial accumulator value. The elements are visited in order from the highest index to the lowest. If Function is not a function, the call fails with reason badarg.
See also: foldl/3 [page 81], map/2 [page 82].

from_list(List::list()) -> array()
Equivalent to from_list(List, undefined) [page 82].

from_list(List::list(), Default::term()) -> array()
Convert a list to an extendible array. Default is used as the value for uninitialized entries of the array. If List is not a proper list, the call fails with reason badarg.
See also: new/2 [page 83], to_list/1 [page 85].

from_orrddict(Orrddict::list()) -> array()
Equivalent to from_orrddict(Orrddict, undefined) [page 82].

from_orrddict(List::list(), Default::term()) -> array()
Convert an ordered list of pairs \{Index, Value\} to a corresponding extendible array. Default is used as the value for uninitialized entries of the array. If List is not a proper, ordered list of pairs whose first elements are nonnegative integers, the call fails with reason badarg.
See also: new/2 [page 83], to_orrddict/1 [page 85].

get(I::integer(), Array::array()) -> term()
Get the value of entry I. If I is not a nonnegative integer, or if the array has fixed size and I is larger than the maximum index, the call fails with reason badarg.
If the array does not have fixed size, this function will return the default value for any index I greater than size(Array)-1.
See also: set/3 [page 84].

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

is_fix(Array::array()) -> bool()
Check if the array has fixed size. Returns true if the array is fixed, otherwise false.
See also: fix/1 [page 81].

map(Function, Array::array()) -> array()
Types:
* Function = (Index::integer(), Value::term()) -> term()
Map the given function onto each element of the array. The elements are visited in order from the lowest index to the highest. If Function is not a function, the call fails with reason badarg.
See also: foldl/3 [page 81], foldr/3 [page 81], sparse_map/2 [page 85].

new() -> array()
Create a new, extendible array with initial size zero.
See also: new/1 [page 83], new/2 [page 83].

new(Options::term()) -> array()
Create a new array according to the given options. By default, the array is extendible and has initial size zero. Array indices start at 0.

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

- \( N \)::integer() or \([\text{size, } N]::\text{integer()}\) Specifies the initial size of the array; this also implies \([\text{fixed, } true]\). If \( N \) is not a nonnegative integer, the call fails with reason \text{badarg}.
- \text{fixed} or \([\text{fixed, } true]\) Creates a fixed-size array; see also \text{fix/1 [page 81]}. 
- \([\text{fixed, false}]\) Creates an extendible (non-fixed-size) array.
- \([\text{default, Value}]\) Sets the default value for the array to \( \text{Value} \).

Options are processed in the order they occur in the list, i.e., later options have higher precedence. The default value is used as the value of uninitialized entries, and cannot be changed once the array has been created.

Examples:

```erlang
array:new(100)
```
creates a fixed-size array of size 100.

```erlang
array:new([default,0])
```
creates an empty, extendible array whose default value is 0.

```erlang
array:new([[size,10],[fixed,false],[default,-1]])
```
creates an extendible array with initial size 10 whose default value is -1.

See also: \text{fix/1 [page 81]}, \text{from_list/2 [page 82]}, \text{get/2 [page 82]}, \text{new/0 [page 82]}, \text{new/2 [page 83]}, \text{set/3 [page 84]}. 

\text{new(Size::integer(), Options::term()) -> array()}

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

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

Example:

```erlang
array:new(100, [default,0])
```
creates a fixed-size array of size 100, whose default value is 0.

See also: \text{new/1 [page 83]}. 

\text{relax(Array::array()) -> array()}

Make the array resizable. (Reverses the effects of \text{fix/1 [page 81]}.)

See also: \text{fix/1 [page 81]}. 

\text{reset(I::integer(), Array::array()) -> array()}

Reset entry \( I \) to the default value for the array. This is equivalent to \( \text{set}(I, \text{default}(\text{Array}), \text{Array}) \), and hence may cause the array to grow in size, but will not shrink it. Shrinking can be done explicitly by calling \text{resize/2} [page 84].

If \( I \) is not a nonnegative integer, or if the array has fixed size and \( I \) is larger than the maximum index, the call fails with reason badarg; cf. \text{set/3} [page 84].

See also: \text{new/2} [page 83], \text{set/3} [page 84].

\[ \text{resize}(\text{Array}::\text{array()} \rightarrow \text{array}) \]

Change the size of the array to that reported by \text{sparse_size/1} [page 85]. If the given array has fixed size, the resulting array will also have fixed size.

See also: \text{resize/2} [page 84], \text{sparse_size/1} [page 85].

\[ \text{resize}(\text{Size}::\text{integer()}, \text{Array}::\text{array()} \rightarrow \text{array}) \]

Change the size of the array. If \( \text{Size} \) is not a nonnegative integer, the call fails with reason badarg. If the given array has fixed size, the resulting array will also have fixed size.

\[ \text{set}(\text{I}::\text{integer()}, \text{Value}::\text{term()}, \text{Array}::\text{array()} \rightarrow \text{array}) \]

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

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

See also: \text{get/2} [page 82], \text{reset/2} [page 84].

\[ \text{size}(\text{Array}::\text{array()} \rightarrow \text{integer}) \]

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

See also: \text{set/3} [page 84], \text{sparse_size/1} [page 85].

\[ \text{sparse_foldl}(\text{Function}, \text{InitialAcc}::\text{term()}, \text{Array}::\text{array()} \rightarrow \text{term()}) \]

Types:

- \( \text{Function} = (\text{Index}::\text{integer()}, \text{Value}::\text{term()}, \text{Acc}::\text{term()} \rightarrow \text{term()} \)

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

See also: \text{foldl/3} [page 81], \text{sparse_foldr/3} [page 85].

\[ \text{sparse_foldr}(\text{Function}, \text{InitialAcc}::\text{term()}, \text{Array}::\text{array()} \rightarrow \text{term()}) \]

Types:

- \( \text{Function} = (\text{Index}::\text{integer()}, \text{Value}::\text{term()}, \text{Acc}::\text{term()} \rightarrow \text{term()} \)
Fold the elements of the array right-to-left using the given function and initial accumulator value, skipping default-valued entries. The elements are visited in order from the highest index to the lowest. If Function is not a function, the call fails with reason badarg.

See also: foldr/3 [page 81], sparse_foldl/3 [page 84].

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

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

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

See also: map/2 [page 82].

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

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

See also: resize/1 [page 84], size/1 [page 84].

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

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

See also: to_list/1 [page 85].

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

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

See also: to_orddict/1 [page 85].

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

Converts the array to a list.

See also: from_list/2 [page 82], sparse_to_list/1 [page 85].

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

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

See also: from_orddict/2 [page 82], sparse_to_orddict/1 [page 85].
**base64**

Erlang Module

Implements base 64 encode and decode, see RFC 2045.

**Exports**

- `encode(Data) -> Base64`
- `encode_to_string(Data) -> Base64String`
  
  **Types:**
  - `Data = string() | binary()`
  - `Base64 = binary()`
  - `Base64String = string()`

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

- `decode(Base64) -> Data`
- `decode_to_string(Base64) -> DataString`
- `mime_decode(Base64) -> Data`
- `mime_decode_to_string(Base64) -> DataString`
  
  **Types:**
  - `Base64 = string() | binary()`
  - `Data = binary()`
  - `DataString = string()`

  Decodes a base64 encoded string to plain ASCII. The string should only consist of characters in the base64 set, see 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 92],
strip_files/1 [page 92] and/or strip_release/1 [page 93].
Reconstructing source code
Here is an example of how to reconstruct source code from the debug information in a
BEAM file Beam:
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 by the time of writing, it is infeasible to break `des3_cbc` encryption without any knowledge of the key. Therefore, as long as the key is kept safe and is unguessable, the encrypted debug information should be safe from intruders.

There are two ways to provide the key:

1. Use the compiler option `{debug_info, Key}`, see `compile(3)`, and the function `crypto_key(fun)/1` [page 93] 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/D%6Cga*68$Mw|L&_Gejr|G"}]
```

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

```
[{debug_info, des3_cbc, t, "My KEY"}],
{debug_info, des3_cbc, [], "%>7|pc/D%6Cga*68$Mw|L&_Gejr|G"}]
```
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}]}n
  | {labeled_exports, [{Function, Arity, Label}]}n
  | {imports, [{Module, Function, Arity}]}n
  | {indexed_imports, [{Index, Module, Function, Arity}]}n
  | {locals, [{Function, Arity}]}n
  | {labeled_locals, [{Function, Arity, Label}]}n
  | {atoms, [{integer(), atom()}]}n
AbstractCode = {AbstVersion, Forms} | no_abstract_code

AbstVersion = atom()
Attribute = atom()
AttributeValue = term()
Arity = int()
Label = int()

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

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

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

chunkname() = abstract_code | attributes | compile_info
  | exports | labeled_exports
  | imports | indexed_imports
  | locals | labeled_locals
  | atoms

chunkref() = chunkname() | chunkid()
Exports

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

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

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

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

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

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

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

Types:
- Beam = beam()
- Module = atom()
- Version = term()
- Reason = see chunks/2
Returns the module version(s). A version is defined by the module attribute -vsn(Vsn). If this attribute is not specified, the version defaults to the checksum of the module. Note that if the version Vsn is not a list, it is made into one, that is [ok, {Module, [Vsn]}] is returned. If there are several -vsn module attributes, the result is the concatenated list of versions. Examples:

1> beam_lib:version(a). % -vsn(1).
   {ok, [a, [1]]}
2> beam_lib:version(b). % -vsn([1]).
   {ok, [b, [1]]}
3> beam_lib:version(c). % -vsn([1]). -vsn(2).
   {ok, [c, [1, 2]]}
4> beam_lib:version(d). % no -vsn attribute
   {ok, [d, [27561320817697377698094100858909383631]]}

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

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

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

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

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

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

{file, Filename} | {binary, Binary} The name (string) of the BEAM file, or the binary from which the information was extracted.
{module, Module} The name (atom) of the module.
{chunks, [{ChunkId, Pos, Size}]} For each chunk, the identifier (string) and the position and size of the chunk data, in bytes.
cmp(Beam1, Beam2) -> ok | {error, beam_lib, Reason}

Types:
- Beam1 = Beam2 = beam()
- Reason = \{modules\_different, Module1, Module2\}
- \{chunks\_different, ChunkId\}
- \{Reason1 - see \text{info/1}\}
- Module1 = Module2 = atom()
- ChunkId = chunkid()

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

cmp\_dirs(Dir1, Dir2) -> \{Only1, Only2, Different\} | {error, beam_lib, Reason1}

Types:
- Dir1 = Dir2 = string() | atom()
- Different = \{\{Filename1, Filename2\}\}
- Only1 = Only2 = \{Filename\}
- Filename = Filename1 = Filename2 = string()
- Reason1 = \{\textit{not a directory}, \textit{term()\} | - see \text{info/1}\}

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

diff\_dirs(Dir1, Dir2) -> ok | \{error, beam_lib, Reason1\}

Types:
- Dir1 = Dir2 = string() | atom()
- Reason1 = \{\textit{not a directory}, \textit{term()\} | - see \text{info/1}\}

The 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.

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

Types:
- Beam1 = Beam2 = beam()
- Module = atom()
- Reason1 - see \text{info/1}\}

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

strip\_files(Files) -> \{ok, \{\{Module, Beam2\}\}\} | \{error, beam_lib, Reason1\}

Types:
- Files = \{Beam1\}
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`.

```erlang
strip_files(Dir) -> ok, {{Module, Filename}} | {error, beam_lib, Reason1}
```

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

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

```erlang
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.

```erlang
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:

```erlang
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.

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

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

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

```
clear_crypto_key_fun() -> {ok, Result}
```

Types:

- `Result = undefined | term()`

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

The `clear_crypto_key_fun/1` either returns `{ok, undefined}` if there was no crypto key fun registered, or `{ok, Term}`, where `Term` is the return value from `CryptoKeyFun(clear)`, see `crypto_key_fun/1`. 
The \texttt{c} module enables users to enter the short form of some commonly used commands.

\textbf{Note:}
These functions are intended for interactive use in the Erlang shell only. The module prefix may be omitted.

\section*{Exports}

\begin{verbatim}
bt(Pid) -> void()
Types:
  • Pid = pid()
Stack backtrace for a process. Equivalent to \texttt{erlang:process:display(Pid, backtrace)}.

c(File) -> {ok, Module} | error
c(File, Options) -> {ok, Module} | error
Types:
  • File = name() – see \texttt{filename(3)}
  • Options = [Opt] – see \texttt{compile:file/2}
\texttt{c/1,2} compiles and then purges and loads the code for a file. \texttt{Options} defaults to \texttt{[\]}. Compilation is equivalent to:
\texttt{compile:file(File, Options ++ \texttt{[report_errors, report_warnings]})}
Note that purging the code means that any \texttt{processes} lingering in old code for the module are killed without warning. See \texttt{code/3} for more information.

cd(Dir) -> void()
Types:
  • Dir = name() – see \texttt{filename(3)}
Changes working directory to \texttt{Dir}, which may be a relative name, and then prints the name of the new working directory.
\end{verbatim}
flush() -> void()
    Flushes any messages sent to the shell.

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

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

i(X, Y, Z) -> void()
    Types:
    • X = Y = Z = int()
    Displays information about a process. Equivalent to process_info(pid(X, Y, Z)), but
    location transparent.

l(Module) -> void()
    Types:
    • Module = atom()
    Purges and loads, or reloads, a module by calling code:purge(Module) followed by
    code:load_file(Module).
    Note that purging the code means that any processes lingering in old code for the
    module are killed without warning. See code/3 for more information.

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

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

ls(Dir) -> void()
    Types:
    • Dir = name() – see filename(3)
    Lists files in directory Dir.
Displays information about the loaded modules, including the files from which they have been loaded.

\[ m(Module) \rightarrow \text{void()} \]

Types:
- Module = atom()
  Displays information about Module.

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

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

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

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

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

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

\[ \text{nl(Module)} \rightarrow \text{void()} \]

Types:
- Module = atom()
  Loads Module on all nodes.

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

Types:
- X = Y = Z = int()
  Converts X, Y, Z to the pid <X.Y.Z>. This function should only be used when debugging.

\[ \text{pwd()} \rightarrow \text{void()} \]

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 198], [erlang(3)], [yecc(3)], [xref(3)]
calendar

Erlang Module

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

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

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

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

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

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

DATA TYPES

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

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

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

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

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

datetime_to_gregorian_seconds(Date, Time) -> Seconds

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

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

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

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

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

gregorian_days_to_date(Days) -> Date

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

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

gregorian_seconds_to_datetime(Seconds) -> {Date, Time}

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

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

is_leap_year(Year) -> bool()

This function checks if a year is a leap year.

last_day_of_the_month(Year, Month) -> int()
This function computes the number of days in a month.

local_time() -> {Date, Time}

Types:
  • Date = date()
  • Time = time()

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

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

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

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

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

Types:
  • Date1 = Date = date()
  • Time1 = Time = time()

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

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

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

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

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

now_to_local_time(Now) -> {Date, Time}

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

This function returns local date and time converted from the return value from erlang:now().
now_to_universal_time(Now) -> {Date, Time}
now_to_datetime(Now) -> {Date, Time}

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

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

seconds_to_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 a valid.

### Leap Years

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

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

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

### Date and Time Source

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

The following facts apply:

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

Erlang Module

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

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

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

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

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

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

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

It is worth noting that the ordered_set type present in Ets is not yet implemented by Dets, neither is the limited support for concurrent updates which makes a sequence of first and next calls safe to use on fixed Ets tables. Both these features will be implemented by Dets in a future release of Erlang/OTP. Until then, the Mnesia
application (or some user implemented method for locking) has to be used to
implement safe concurrency. Currently, no library of Erlang/OTP has support for
ordered disk based term storage.

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

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

Types

access() = read | read_write
auto_save() = infinity | int()
binding_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 \texttt{bchunk} to the \texttt{format} option of the \texttt{init_table/3} function. The \texttt{Mnesia} application uses this function for copying open tables. Unless the table is protected using \texttt{safe_fixtable/2}, calls to \texttt{bchunk/2} may not work as expected if concurrent updates are made to the table.

The first time \texttt{bchunk/2} is called, an initial continuation, the atom \texttt{start}, must be provided. The \texttt{bchunk/2} function returns a tuple \{\texttt{Continuation2}, \texttt{Data}\}, where \texttt{Data} is a list of objects. \texttt{Continuation2} is another continuation which is to be passed on to a subsequent call to \texttt{bchunk/2}. With a series of calls to \texttt{bchunk/2} it is possible to extract all objects of the table. \texttt{bchunk/2} returns \texttt{'$end_of_table'} when all objects have been returned, or \{\texttt{error}, \texttt{Reason}\} if an error occurs.

\texttt{close(Name)} \rightarrow \texttt{ok} \mid \{\texttt{error}, \texttt{Reason}\}

Types:

- \texttt{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.

\texttt{delete(Name, Key)} \rightarrow \texttt{ok} \mid \{\texttt{error}, \texttt{Reason}\}

Types:

- \texttt{Name = name()}

Deletes all objects with the key \texttt{Key} from the table \texttt{Name}.

\texttt{delete_all_objects(Name)} \rightarrow \texttt{ok} \mid \{\texttt{error}, \texttt{Reason}\}

Types:

- \texttt{Name = name()}

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

\texttt{delete_object(Name, Object)} \rightarrow \texttt{ok} \mid \{\texttt{error}, \texttt{Reason}\}

Types:

- \texttt{Name = name()}
- \texttt{Object = object()}

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

\texttt{first(Name)} \rightarrow \texttt{Key} \mid \texttt{'$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) -&gt; 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]) -&gt; ok | {error, Reason}
Types:
- \( \text{Name} = \text{atom()} \)
- \( \text{InitFun} = \text{fun}(\text{Arg}) \rightarrow \text{Res} \)
- \( \text{Arg} = \text{read} \mid \text{close} \)
- \( \text{Res} = \text{end_of_input} \mid \{\text{object()}\}, \text{InitFun} \mid \{\text{Data}, \text{InitFun}\} \mid \text{term()} \)
- \( \text{Data} = \text{binary()} \mid \text{tuple()} \)

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

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

If the type of the table is \( \text{set} \) 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. Duplicate keys should be avoided, or the file will be unnecessarily fragmented. This holds also for duplicated objects stored in tables of type \( \text{bag} \).

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 \( \text{init_table/2} \) returns which will significantly slow down access to the table for a period of time. The minimum number of slots is set by the \( \text{open_file/2} \) option \( \text{min_no_slots} \) and returned by the \( \text{info/2} \) item \( \text{no_slots} \). See also the \( \text{min_no_slots} \) option below.

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

- \( \{\text{min_no_slots}, \text{no_slots()}\} \). Specifies the estimated number of different keys that will be stored in the table. The \( \text{open_file} \) 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.
- \( \{\text{format}, \text{Format}\} \). Specifies the format of the objects returned by the function \( \text{InitFun} \). If \( \text{Format} \) is \text{term} (the default), \( \text{InitFun} \) is assumed to return a list of tuples. If \( \text{Format} \) is \text{bchunk}, \( \text{InitFun} \) is assumed to return \( \text{Data} \) as returned by \( \text{bchunk/2} \). This option overrides the \( \text{min_no_slots} \) option.

\( \text{insert(Name, Objects)} \rightarrow \text{ok} \mid \{\text{error}, \text{Reason}\} \)

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

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

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

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

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

is_dets_file(FileName) -> Bool | {error, Reason}
Types:
- FileName = file()
- Bool = bool()

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

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

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

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

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

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

match(Continuation) -> [{Match}, Continuation2] | '$end_of_table' | {error, Reason}
Types:
- Continuation = Continuation2 = bindings_cont()
- Match = [term()]

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

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

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

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

Returns for each object of the table Name that matches Pattern a list of bindings in some unspecified order. See ets(3) [page 166] for a description of patterns. If the keypos'th element of Pattern is unbound, all objects of the table are matched. If the keypos'th element is bound, only the objects with the right key are matched.

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

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

Matches some or all objects of the table Name and returns a non-empty list of the bindings that match Pattern in some unspecified order. See ets(3) [page 166] for a description of patterns.

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

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

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

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

Types:
- Name = name()
- Pattern = tuple()
Deletes all objects that match Pattern from the table Name. See ets:match/2 [page 174] 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 166] for a description of patterns.

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

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

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

Types:
- Name = name()
- Pattern = tuple()
- N = default | int()
- Object = object()
- Continuation = object_cont()
Matches some or all objects stored in the table \texttt{Name} and returns a non-empty list of the objects that match \texttt{Pattern} in some unspecified order. See \texttt{ets(3)} [page 166] for a description of patterns.

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

If the keypos'th element of \texttt{Pattern} is bound, all objects of the table are matched. If the keypos'th element is unbound, all objects of the table are matched, \texttt{N} objects at a time, 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 matching objects with the same key are always returned in the same reply which implies that more than \texttt{N} objects may sometimes be returned.

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

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

\textbf{Types:}
\begin{itemize}
  \item Name = \texttt{name()}
  \item Key = \texttt{term()}
  \item Bool = \texttt{bool()}
\end{itemize}

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

\texttt{next(Name, Key1) }\rightarrow\texttt{ Key2 | \$end_of_table}

\textbf{Types:}
\begin{itemize}
  \item Name = \texttt{name()}
  \item Key1 = Key2 = \texttt{term()}
\end{itemize}

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

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

\texttt{open_file(Filename) }\rightarrow\texttt{ \{ok, Reference\} | \{error, Reason\}}

\textbf{Types:}
\begin{itemize}
  \item FileName = \texttt{file()}
  \item Reference = \texttt{ref()}
\end{itemize}

Opens an existing table. If the table has not been properly closed, it will be repaired. The returned reference is to be used as the name of the table. This function is most useful for debugging purposes.

\texttt{open_file(Name, Args) }\rightarrow\texttt{ \{ok, Name\} | \{error, Reason\}}

\textbf{Types:}
\begin{itemize}
  \item Name = \texttt{atom()}
\end{itemize}
Opens a table. An empty DetS table is created if no file exists.

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

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

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

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

The value force means that a reparation will take place even if the table has been properly closed. This is how to convert tables created by older versions of STDLIB. An example is tables hashed with the deprecated erlang:hash/2 BIF. Tables created with DetS from a STDLIB version of 1.8.2 and later use the erlang:phash/2 function or the erlang:phash2/1 function, which is preferred. The repair option is ignored if the table is already open.
- `{type, type()}`, the type of the table. The default value is `set`.
- `{version, version()}`, the version of the format used for the table. The default value is `9`. Tables on the format used before OTP R8 can be created by giving the value `8`. A version `8` table can be converted to a version `9` table by giving the options `{version,9}` and `{repair,force}`.

```
pid2name(Pid) -> {ok, Name} | undefined
```
Types:
- `Name = name()`
- `Pid = pid()`

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

This function is meant to be used for debugging only.

```
repair_continuation(Continuation, MatchSpec) -> Continuation2
```
Types:
- `Continuation = Continuation2 = select_cont()`
- `MatchSpec = match_spec()`

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

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

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

**Note:**
This function is very rarely needed in application code. It is used by Mnesia to implement distributed `select/3` and `select/1` sequences. A normal application would either use Mnesia or keep the continuation from being converted to external format.

The reason for not having an external representation of compiled match specifications is performance. It may be subject to change in future releases, while this interface will remain for backward compatibility.

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

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

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

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

---

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

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

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:
  • Name = name()
  • MatchSpec = match_spec()
  • N = int()

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

If the 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:
  • Name = name()
  • I = int()
  • Object = object()

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

\begin{verbatim}
sync(Name) -> ok | {error, Reason}
Types:
  • Name = name()

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

Note that the space management data structures kept in RAM, the buddy system, is also written to the disk. This may take some time if the table is fragmented.
\end{verbatim}
**table**  (Name [, Options]) -> QueryHandle

**Types:**
- \texttt{Name} = \text{name()}
- \texttt{QueryHandle} = \text{a query handle, see qlc(3)-}
- \texttt{Options} = [\text{Option} | \text{Option}]
- \texttt{Option} = \{n\_objects, Limit | \{traverse, TraverseMethod\}
- \texttt{Limit} = \text{default | integer() >= 1}
- \texttt{TraverseMethod} = \text{first, next | select | \{select, MatchSpec\}}
- \texttt{MatchSpec} = \text{match\_spec()}

Returns a QLC (Query List Comprehension) query handle. The module \texttt{qlc} implements a query language aimed mainly at Mnesia but \texttt{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 \texttt{qlc} uses \texttt{dets:lookup/2} to look up the keys, but when that is not possible the whole table is traversed. The option \texttt{traverse} determines how this is done:

- \texttt{first, next}. The table is traversed one key at a time by calling \texttt{dets:first/1} and \texttt{dets:next/2}.
- \texttt{select}. The table is traversed by calling \texttt{dets:select/3} and \texttt{dets:select/1}. The option \texttt{n\_objects} determines the number of objects returned (the third argument of \texttt{select/3}). The match specification (the second argument of \texttt{select/3}) is assembled by \texttt{qlc}: simple filters are translated into equivalent match specifications while more complicated filters have to be applied to all objects returned by \texttt{select/3} given a match specification that matches all objects.
- \texttt{\{select, MatchSpec\}}. As for \texttt{select} the table is traversed by calling \texttt{dets:select/3} and \texttt{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 \texttt{qlc}.

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

\begin{verbatim}
1> dets:open_file(t, []),
  ok = 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}]]).
\end{verbatim}

An example with implicit match specification:

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

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

\begin{verbatim}
3> qlc:info(QH1) == qlc:info(QH2).
true
\end{verbatim}

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

**to\_ets** (Name, EtsTab) -> EtsTab | \{error, Reason\}

**Types:**
- Name = name()
- EtsTab = -see ets(3)-

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

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

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

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

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

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

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

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

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

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

update_counter(Name, Key, Increment) -> Result

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

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

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

See Also
ets(3) [page 166], mnesia(3), qlc(3) [page 318]
dict

Erlang Module

Dict implements a Key - Value dictionary. The representation of a dictionary is not defined.
This module provides exactly the same interface as the module orddict. One difference is that while this module considers two keys as different if they do not match (=:=), orddict considers two keys as different if and only if they do not compare equal (==).

DATA TYPES

dictionary()
    as returned by new/0

Exports

append(Key, Value, Dict1) -> Dict2
    Types:
    • Key = Value = term()
    • Dict1 = Dict2 = dictionary()
    This function appends a new Value to the current list of values associated with Key. An exception is generated if the initial value associated with Key is not a list of values.

append_list(Key, ValList, Dict1) -> Dict2
    Types:
    • ValList = [Value]
    • Key = Value = term()
    • Dict1 = Dict2 = dictionary()
    This function appends a list of values ValList to the current list of values associated with Key. An exception is generated if the initial value associated with Key is not a list of values.

erase(Key, Dict1) -> Dict2
    Types:
    • Key = term()
    • Dict1 = Dict2 = dictionary()
This function erases all items with a given key from a dictionary.

```haskell
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.

```haskell
fetch_keys(Dict) -> Keys
```

Types:
- `Dict` = `dictionary()`
- `Keys` = `term()`

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

```haskell
filter(Pred, Dict1) -> Dict2
```

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

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

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

Types:
- `Key` = `Value` = `term()`
- `Dict` = `dictionary()`

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

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

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

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

```haskell
from_list(List) -> Dict
```

Types:
- `List` = `[(Key, Value)]`

**dict**

- Dict = dictionary()

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

is_key(Key, Dict) -> bool()

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

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

map(Fun, Dict1) -> Dict2

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

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

merge(Fun, Dict1, Dict2) -> Dict3

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

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

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

but is faster.

new() -> dictionary()

This function creates a new dictionary.

size(Dict) -> int()

Types:
- Dict = dictionary()

Returns the number of elements in a Dict.

store(Key, Value, Dict1) -> Dict2

Types:
- Key = Value = term()
- Dict1 = Dict2 = dictionary()
  This function stores a Key - Value pair in a dictionary. If the Key already exists in Dict1, the associated value is replaced by Value.

to_list(Dict) -> List
Types:
- Dict = dictionary()
- List = [{Key, Value}]
This function converts the dictionary to a list representation.

update(Key, Fun, Dict1) -> Dict2
Types:
- Key = term()
- Fun = fun(Value1) -> Value2
- Value1 = Value2 = term()
- Dict1 = Dict2 = dictionary()
Update the a value in a dictionary by calling Fun on the value to get a new value. An exception is generated if Key is not present in the dictionary.

update(Key, Fun, Initial, Dict1) -> Dict2
Types:
- Key = Initial = term()
- Fun = fun(Value1) -> Value2
- Value1 = Value2 = term()
- Dict1 = Dict2 = dictionary()
Update the a value in a dictionary by calling Fun on the value to get a new value. If Key is not present in the dictionary then Initial will be stored as the first value. For example append/3 could be defined as:
append(Key, Val, D) ->
    update(Key, fun (Old) -> Old ++ [Val] end, [Val], D).

update_counter(Key, Increment, Dict1) -> Dict2
Types:
- Key = term()
- Increment = number()
- Dict1 = Dict2 = dictionary()
Add Increment to the value associated with Key and store this value. If Key is not present in the dictionary then Increment will be stored as the first value. This could be defined as:
update_counter(Key, Incr, D) ->
    update(Key, fun (Old) -> Old + Incr end, Incr, D).
but is faster.
Notes

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

```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).
[files, [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 210], `orddict(3)` [page 296]
digraph

Erlang Module

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

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

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

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

Exports

```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:
- \(G = \text{digraph}()\)
- \(E = \text{edge}()\)
- \(V1 \equiv V2 = \text{vertex}()\)
- \(Label = \text{label}()\)
- \(Reason = \{\text{bad_edge}, \text{Path}\} | \{\text{bad_vertex}, V\}\)
- \(Path = [\text{vertex}()]\)
add_edge/5 creates (or modifies) the edge $E$ of the digraph $G$, using Label as the (new) label [page 125] of the edge. The edge is emanating [page 125] from $V_1$ and incident [page 125] on $V_2$. Returns $E$.

add_edge($G$, $V_1$, $V_2$, Label) is equivalent to add_edge($G$, $E$, $V_1$, $V_2$, Label), where $E$ is a created edge. The created edge is represented by the term ['$e'$ | $N$], where $N$ is an integer $>0$.

add_edge($G$, $V_1$, $V_2$) is equivalent to add_edge($G$, $V_1$, $V_2$, []).

If the edge would create a cycle in an acyclic digraph [page 125], then {error, {bad_edge, Path}} is returned. If either of $V_1$ or $V_2$ is not a vertex of the digraph $G$, then {error, {bad_vertex, V}} is returned, $V = V_1$ or $V = V_2$.

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 125] 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. The created vertex is represented by the term ['$v'$ | $N$], where $N$ is an integer $>0$.

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($G$, $V_1$, $V_2$) -> true

types:
- $G$ = digraph()
- $V_1$ = $V_2$ = vertex()
Deletes edges from the digraph \( G \) until there are no paths [page 125] from the vertex \( V_1 \) to the vertex \( V_2 \).

A sketch of the procedure employed: Find an arbitrary simple path [page 125] \( v[1], v[2], \ldots, v[k] \) from \( V_1 \) to \( V_2 \) in \( G \). Remove all edges of \( G \) emanating [page 125] from \( v[i] \) and incident [page 125] to \( v[i+1] \) for \( 1 < i < k \) (including multiple edges). Repeat until there is no path between \( V_1 \) and \( V_2 \).

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

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

Deletes the vertex \( V \) from the digraph \( G \). Any edges emanating [page 125] from \( V \) or incident [page 125] on \( V \) are also deleted.

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

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

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

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

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

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

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

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

Returns \( \{E, V_1, V_2, \text{Label}\} \) where \( \text{Label} \) is the label [page 125] of the edge \( E \) emanating [page 125] from \( V_1 \) and incident [page 125] on \( V_2 \) of the digraph \( G \). If there is no edge \( E \) of the digraph \( G \), then false is returned.

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

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

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

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

Returns a list of all edges emanating [page 125] from or incident [page 125] on \( V \) of the digraph \( G \), in some unspecified order.

\[
\text{get_cycle}(G, V) \to \text{Vertices} | \text{false}
\]
Types:
- \( G = \text{digraph()} \)
- \( V1 = V2 = \text{vertex()} \)
- \( \text{Vertices} = [\text{vertex()}] \)

If there is a simple cycle [page 125] 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 125] through \( V \), then the loop is returned as a list \([V]\). If there are no cycles through \( V \), then \text{false} is returned.

\( \text{get_path/3} \) is used for finding a simple cycle through \( V \).

\[
\text{get_path}(G, V1, V2) \to \text{Vertices} | \text{false}
\]
Types:
- \( G = \text{digraph()} \)
- \( V1 = V2 = \text{vertex()} \)
- \( \text{Vertices} = [\text{vertex()}] \)

Tries to find a simple path [page 125] from the vertex \( V1 \) to the vertex \( V2 \) of the digraph \( G \). Returns the path as a list \([V1, \ldots, V2]\) of vertices, or \text{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.

\[
\text{get_short_cycle}(G, V) \to \text{Vertices} | \text{false}
\]
Types:
- \( G = \text{digraph()} \)
- \( V1 = V2 = \text{vertex()} \)
- \( \text{Vertices} = [\text{vertex()}] \)

Tries to find an as short as possible simple cycle [page 125] through the vertex \( V \) of the digraph \( G \). Returns the cycle as a list \([V, \ldots, V]\) of vertices, or \text{false} if no simple cycle through \( V \) exists. Note that a loop [page 125] through \( V \) is returned as the list \([V, V]\).

\( \text{get_short_path/3} \) is used for finding a simple cycle through \( V \).

\[
\text{get_short_path}(G, V1, V2) \to \text{Vertices} | \text{false}
\]
Types:
- \( G = \text{digraph()} \)
- \( V1 = V2 = \text{vertex()} \)
- \( \text{Vertices} = [\text{vertex()}] \)
Tries to find an as short as possible simple path [page 125] from the vertex \( V_1 \) to the vertex \( V_2 \) of the digraph \( G \). Returns the path as a list \([V_1, \ldots, V_2]\) of vertices, or false if no simple path from \( V_1 \) to \( V_2 \) of length one or more exists.

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

\[
in\text{degree}(G, V) \to \text{integer()}
\]

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

Returns the in-degree [page 125] of the vertex \( V \) of the digraph \( G \).

\[
in\text{edges}(G, V) \to \text{Edges}
\]

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

Returns a list of all edges incident [page 125] on \( V \) of the digraph \( G \), in some unspecified order.

\[
in\text{neighbours}(G, V) \to \text{Vertices}
\]

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

Returns a list of all in-neighbours [page 125] of \( V \) of the digraph \( G \), in some unspecified order.

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

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

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

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

new(Type) -> digraph()
   Types:
   - Type = [cyclic | acyclic | private | protected]
   Returns an empty digraph [page 125] with properties according to the options in Type:
   cyclic Allow cycles [page 125] in the digraph (default).
   acyclic The digraph is to be kept acyclic [page 125].
   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 or Type is not a proper list, there will be a
   badarg exception.

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 125] 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 125] from V of the digraph G, in some
   unspecified order.

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

Returns a list of all out-neighbours [page 125] of \( V \) of the digraph \( G \), in some unspecified order.

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

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

Returns \( \{V, \text{Label}\} \) where \( \text{Label} \) is the label [page 125] of the vertex \( V \) of the digraph \( G \), or \text{false} if there is no vertex \( V \) of the digraph \( G \).

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

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

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

See Also

digraph\_utils(3) [page 132], ets(3) [page 166]
The `digraph_utils` module implements some algorithms based on depth-first traversal of directed graphs. See the `digraph` module for basic functions on directed graphs.

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

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

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

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

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

A subgraph \(G'\) of \(G\) is a digraph whose vertices and edges form subsets of the vertices and edges of \(G\). \(G'\) is maximal with respect to a property \(P\) if all other subgraphs that include the vertices of \(G'\) do not have the property \(P\). A strongly connected component is a maximal subgraph such that there is a path between each pair of vertices. A connected component is a maximal subgraph such that there is a path between each pair of vertices, considering all edges undirected.

An arborescence is an acyclic digraph with a vertex \(V\), the root, such that there is a unique path from \(V\) to every other vertex of \(G\). A tree is an acyclic non-empty digraph such that there is a unique path between every pair of vertices, considering all edges undirected.
Exports

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

Types:
  • Digraph = digraph()
  • Root = vertex()
Returns {yes, Root} if Root is the root [page 132] of the arborescence Digraph, no
otherwise.

components(Digraph) -> [Component]

Types:
  • Digraph = digraph()
  • Component = [vertex()]
Returns a list of connected components [page 132]. 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 132]
of Digraph as returned by strong_components/1. If X and Y are strongly connected
components, and there exist vertices x and y in X and Y respectively such that there is
an edge emanating [page 132] from x and incident [page 132] 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 132] [].
Each and every cycle [page 132] is included in some strongly connected component,
which implies that there always exists a topological ordering [page 132] of the created
digraph.

cyclic_strong_components(Digraph) -> [StrongComponent]

Types:
  • Digraph = digraph()
  • StrongComponent = [vertex()]
Returns a list of strongly connected components [page 132]. 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 132] 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 132].

```cpp
is_arborescence(Digraph) -> bool()
```

Types:
- `Digraph = digraph()`

Returns `true` if and only if the digraph `Digraph` is an arborescence [page 132].

```cpp
is_tree(Digraph) -> bool()
```

Types:
- `Digraph = digraph()`

Returns `true` if and only if the digraph `Digraph` is a tree [page 132].

```cpp
loop_vertices(Digraph) -> Vertices
```

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

Returns a list of all vertices of `Digraph` that are included in some loop [page 132].

```cpp
postorder(Digraph) -> Vertices
```

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

Returns all vertices of the digraph `Digraph`. The order is given by a depth-first traversal [page 132] 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.

```cpp
preorder(Digraph) -> Vertices
```

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

Returns all vertices of the digraph `Digraph`. The order is given by a depth-first traversal [page 132] of the digraph, collecting visited vertices in pre-order.

```cpp
reachable(Vertices, Digraph) -> Vertices
```

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

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path [page 132] in `Digraph` from some vertex of `Vertices` to the vertex. In particular, since paths may have length zero, the vertices of `Vertices` are included in the returned list.
reachable_neighbours(Vertices, Digraph) -> Vertices

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

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path [page 132] 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 132] are returned.

reaching(Vertices, Digraph) -> Vertices

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

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path [page 132] from the vertex to some vertex of Vertices. In particular, since paths may have length zero, the vertices of Vertices are included in the returned list.

reaching_neighbours(Vertices, Digraph) -> Vertices

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

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path [page 132] of length one or more from the vertex to some vertex of Vertices. As a consequence, only those vertices of Vertices that are included in some cycle [page 132] are returned.

strong_components(Digraph) -> [StrongComponent]

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

Returns a list of strongly connected components [page 132]. 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

Types:
  • Digraph = Subgraph = digraph()
  • Options = [{type, SubgraphType}, {keep_labels, bool()}]
  • SubgraphType = inherit | type()
  • Vertices = [vertex()]
Creates a maximal subgraph [page 132] of Digraph having as vertices those vertices of Digraph that are mentioned in Vertices.

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

If the value of the option keep_labels is true, which is the default, then the labels [page 132] 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, []).

There will be a badarg exception if any of the arguments are invalid.

topsort(Digraph) -> Vertices | false

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

Returns a topological ordering [page 132] of the vertices of the digraph Digraph if such an ordering exists, false otherwise. For each vertex in the returned list, there are no out-neighbours [page 132] that occur earlier in the list.

See Also
digraph(3) [page 125]
The Erlang code preprocessor includes functions which are used by `compile` to preprocess macros and include files before the actual parsing takes place.

Exports

\[
\text{open(FileName, IncludePath)} \rightarrow \text{\{ok, Epp\} | \{error, ErrorDescriptor\}}
\]

\[
\text{open(FileName, IncludePath, PredefMacros)} \rightarrow \text{\{ok, Epp\} | \{error, ErrorDescriptor\}}
\]

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.

\[
\text{close(Epp)} \rightarrow \text{ok}
\]

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

Closes the preprocessing of a file.

\[
\text{parse_erl_form(Epp)} \rightarrow \text{\{ok, AbsForm\} | \{eof, Line\} | \{error, ErrorInfo\}}
\]

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.

\[
\text{parse_file(FileName, IncludePath, PredefMacro)} \rightarrow \text{\{ok, [Form]\} | \{error, OpenError\}}
\]

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 148]
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}. 

STDLIB Reference Manual  erl_eval
expr_list(ExpressionList, Bindings) -> [ValueList, NewBindings]
expr_list(ExpressionList, Bindings, LocalFunctionHandler) -> [ValueList, NewBindings]
expr_list(ExpressionList, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) ->
[ValueList, NewBindings]

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

new_bindings() -> BindingStruct

Returns an empty binding structure.

bindings(BindingStruct) -> Bindings

Returns the list of bindings contained in the binding structure.

binding(Name, BindingStruct) -> Binding

Returns the binding of Name in BindingStruct.

add_binding(Name, Value, Bindings) -> BindingStruct

Adds the binding Name = Value to Bindings. Returns an updated binding structure.

del_binding(Name, Bindings) -> BindingStruct

Removes the binding of Name in Bindings. Returns an updated binding structure.

Local Function Handler

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

{value, Func} This defines a local function handler which is called with:
    Func(Name, Arguments)

Name is the name of the local function (an atom) and Arguments is a list of the
evaluated arguments. The function handler returns the value of the local function.
In this case, it is not possible to access the current bindings. To signal an error, the
function handler just calls exit/1 with a suitable exit value.

{eval, Func} This defines a local function handler which is called with:
    Func(Name, Arguments, Bindings)

Name is the name of the local function (an atom), Arguments is a list of the
unevaluated arguments, and Bindings are the current variable bindings. The
function handler returns:
    {value, Value, NewBindings}
Value is the value of the local function and NewBindings are the updated variable bindings. In this case, the function handler must itself evaluate all the function arguments and manage the bindings. To signal an error, the function handler just calls exit/1 with a suitable exit value.

none There is no local function handler.

Non-local Function Handler

The optional argument NonlocalFunctionHandler may be used to define a function which is called in the following cases: a functional object (fun) is called; a built-in function is called; a function is called using the M:F syntax, where M and F are atoms or expressions; an operator Op/A is called (this is handled as a call to the function erlang:Op/A). Exceptions are calls to erlang:apply/2,3; neither of the function handlers will be called for such calls. The argument can have the following formats:

\{value,Func\} This defines an nonlocal function handler which is called with:

    Func(FuncSpec, Arguments)

FuncSpec is the name of the function on the form \{Module,Function\} or a fun, and Arguments is a list of the evaluated arguments. The function handler returns the value of the function. To signal an error, the function handler just calls exit/1 with a suitable exit value.

none There is no nonlocal function handler.

Note:

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

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

Bugs

The evaluator is not complete. receive cannot be handled properly.

Any undocumented functions in erl_eval should not be used.
erl\_expand\_records

Erlang Module

Exports

module(AbsForms, CompileOptions) -> AbsForms

Types:

- AbsForms = [term()]
- CompileOptions = [term()]

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

See Also

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

Erlang Module

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

Exports

\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 148], compile(3).
erl_internal

Erlang Module

This module defines Erlang BIFs, guard tests and operators. This module is only of interest to programmers who manipulate Erlang code.

Exports

bif(Name, Arity) -> bool()

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

Returns true if Name/Arity is an Erlang BIF which is automatically recognized by the compiler, otherwise false.

guard_bif(Name, Arity) -> bool()

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

Returns true if Name/Arity is an Erlang BIF which is allowed in guards, otherwise false.

type_test(Name, Arity) -> bool()

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

Returns true if Name/Arity is a valid Erlang type test, otherwise false.

arith_op(OpName, Arity) -> bool()

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

Returns true if OpName/Arity is an arithmetic operator, otherwise false.

bool_op(OpName, Arity) -> bool()

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

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

comp_op(OpName, Arity) -> bool()

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

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

list_op(OpName, Arity) -> bool()

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

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

send_op(OpName, Arity) -> bool()

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

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

op_type(OpName, Arity) -> Type

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

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

```erlang
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 maintenance 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 137], or parser erl_parse(3) [page 148] The warnings and errors returned have the following format:

{[FileName2, [ErrorInfo]]}

The errors and warnings are listed in the order in which they are encountered in the forms. This means that the errors from one file may be split into different entries in the list of errors.

is_guard_test(Expr) -> bool()

Types:

- Expr = term()

This function tests if Expr is a legal guard test. Expr is an Erlang term representing the abstract form for the expression. erl_parse:parse_exprs(Tokens) can be used to generate a list of Expr.

format_error(ErrorDescriptor) -> Chars

Types:

- ErrorDescriptor = errordesc()
- Chars = [char() | Chars]

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

Error Information

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

{ErrorLine, Module, ErrorDescriptor}

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

apply(Module, format_error, ErrorDescriptor)

See Also

erl_parse(3) [page 148], epp(3) [page 137]
erl_parse

Erlang Module

This module is the basic Erlang parser which converts tokens into the abstract form of either forms (i.e., top-level constructs), expressions, or terms. The Abstract Format is described in the ERTS User’s Guide. Note that a token list must end with the dot token in order to be acceptable to the parse functions (see erl_scan).

Exports

parse_form(Tokens) -> {ok, AbsForm} | {error, ErrorInfo}
Types:
- Tokens = [Token]
- Token = {Tag,Line} | {Tag,Line,term()}
- Tag = atom()
- AbsForm = term()
- ErrorInfo = see section Error Information below.
This function parses Tokens as if it were a form. It returns:
- {ok, AbsForm} The parsing was successful. AbsForm is the abstract form of the parsed form.
- {error, ErrorInfo} An error occurred.

parse_exprs(Tokens) -> {ok, Expr_list} | {error, ErrorInfo}
Types:
- Tokens = [Token]
- Token = {Tag,Line} | {Tag,Line,term()}
- Tag = atom()
- Expr_list = [AbsExpr]
- AbsExpr = term()
- ErrorInfo = see section Error Information below.
This function parses Tokens as if it were a list of expressions. It returns:
- {ok, Expr_list} The parsing was successful. Expr_list is a list of the abstract forms of the parsed expressions.
- {error, ErrorInfo} An error occurred.

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

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

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

format_error(Descriptor) -> Chars

Types:
- Descriptor = errordesc()
- Chars = [char() | Chars]

Uses an Descriptor 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 246], erl_scan(3) [page 154], 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

\[
\text{form}(\text{Form}) \to \text{DeepCharList} \\
\text{form}(\text{Form}, \text{HookFunction}) \to \text{DeepCharList}
\]

Types:
- \text{Form} = \text{term()}
- \text{HookFunction} = \text{see separate description below.}
- \text{DeepCharList} = [\text{char()} | \text{DeepCharList}]

Pretty prints a \text{Form} which is an abstract form of a type which is returned by \text{erl\_parse:parse\_form}.

\[
\text{attribute}(\text{Attribute}) \to \text{DeepCharList} \\
\text{attribute}(\text{Attribute}, \text{HookFunction}) \to \text{DeepCharList}
\]

Types:
- \text{Attribute} = \text{term()}
- \text{HookFunction} = \text{see separate description below.}
- \text{DeepCharList} = [\text{char()} | \text{DeepCharList}]

The same as \text{form}, but only for the attribute \text{Attribute}.

\[
\text{function}(\text{Function}) \to \text{DeepCharList} \\
\text{function}(\text{Function}, \text{HookFunction}) \to \text{DeepCharList}
\]

Types:
- \text{Function} = \text{term()}
- \text{HookFunction} = \text{see separate description below.}
- \text{DeepCharList} = [\text{char()} | \text{DeepCharList}]

The same as \text{form}, but only for the function \text{Function}.

\[
\text{guard}(\text{Guard}) \to \text{DeepCharList} \\
\text{guard}(\text{Guard}, \text{HookFunction}) \to \text{DeepCharList}
\]
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 246], erl_parse(3) [page 148], erl_eval(3) [page 139]
erl_scan

Erlang Module

This module contains functions for tokenizing characters into Erlang tokens.

DATA TYPES

category() = atom()
column() = integer() > 0
line() = integer()
location() = line() | {line(), column()}
reserved_word_fun() -> fun(atom()) -> bool()
set_attribute_fun() -> fun(term()) -> term()
symbol() = atom() | float() | integer() | string()
token() = {category(), attributes()} | {category(), attributes(), symbol()}
attributes() = line() | list() | tuple()

Exports

string(String) -> Return
string(String, StartLocation) -> Return
string(String, StartLocation, Options) -> Return

Types:
- String = string()
- Return = {ok, Tokens, EndLocation} | Error
- Tokens = [token()]
- Error = {error, ErrorInfo, EndLocation}
- StartLocation = EndLocation = location()
- Option = Option | [Option]
- Option = [reserved_word_fun, reserved_word_fun()] | return_comments | return_white_spaces | return | text

Takes the list of characters String and tries to scan (tokenize) them. Returns {ok, Tokens, EndLocation}, where Tokens are the Erlang tokens from String. EndLocation is the first location after the last token.

{error, ErrorInfo, EndLocation} is returned if an error occurs. EndLocation is the first location after the erroneous token.

string(String) is equivalent to string(String, 1), and string(String, StartLocation) is equivalent to string(String, StartLocation, []).
StartLocation indicates the initial location when scanning starts. If StartLocation is a line attributes() as well as EndLocation and ErrorLocation will be lines. If StartLocation is a pair of a line and a column attributes() takes the form of an opaque compound data type, and EndLocation and ErrorLocation will be pairs of a line and a column. The token attributes contain information about the column and the line where the token begins, as well as the text of the token (if the text option is given), all of which can be accessed by calling token_info/1,2 [page 156] or attributes_info/1,2 [page 157].

A token is a tuple containing information about syntactic category, the token attributes, and the actual terminal symbol. For punctuation characters (e.g. ; , |) and reserved words, the category and the symbol coincide, and the token is represented by a two-tuple. Three-tuples have one of the following forms: {atom, Info, atom()}, {char, Info, integer()}, {comment, Info, string()}, {float, Info, float()}, {integer, Info, integer()}, {var, Info, atom()}, and {white_space, Info, string()}. The valid options are:

reserved_word_fun, reserved_word_fun() A callback function that is called when the scanner has found an unquoted atom. If the function returns true, the unquoted atom itself will be the category of the token; if the function returns false, atom will be the category of the unquoted atom.

return_comments Return comment tokens.

return_white_spaces Return white space tokens. By convention, if there is a newline character, it is always the first character of the text (there cannot be more than one newline in a white space token).

return Short for [return_comments, return_white_spaces].

text Include the token's text in the token attributes. The text is the part of the input corresponding to the token.

tokens(Continuation, CharSpec, StartLocation) -> Return
tokens(Continuation, CharSpec, StartLocation, Options) -> Return

Types:
- Continuation = [] | Continuation1
- Return = {done, Result, LeftOverChars} | {more, Continuation1}
- LeftOverChars = CharSpec
- CharSpec = string() | eof
- Continuation1 = tuple()
- Result = {ok, Tokens, EndLocation} | {eof, EndLocation} | Error
- Tokens = [token()]
- Error = {error, ErrorInfo, EndLocation}
- StartLocation = EndLocation = location()
- Options = Option | [Option]
- Option = {reserved_word_fun, reserved_word_fun()} | return_comments | return_white_spaces | return

This is the re-entrant scanner which scans characters until a dot (\' . \' followed by a white space) or eof has been reached. It returns:
{done, Result, LeftOverChars} This return indicates that there is sufficient input data to get a result. Result is:
{ok, Tokens, EndLocation} The scanning was successful. Tokens is the list of tokens including dot.
{eof, EndLocation} End of file was encountered before any more tokens.
{error, ErrorInfo, EndLocation} An error occurred.
{more, Continuation1} More data is required for building a term. Continuation1 must be passed in a new call to tokens/3,4 when more data is available.

The CharSpec eof signals end of file. LeftoverChars will then take the value eof as well.
tokens(Continuation, CharSpec, StartLocation) is equivalent to
tokens(Continuation, CharSpec, StartLocation, []).
See string/3 [page 154] for a description of the various options.

reserved_word(Atom) -> bool()
Types:
• Atom = atom()
Returns true if Atom is an Erlang reserved word, otherwise false.

token_info(Token) -> TokenInfo
Types:
• Token = token()
• TokenInfo = [TokenInfoTuple]
• TokenInfoTuple = {TokenItem, Info}
• TokenItem = atom()
• Info = term()
Returns a list containing information about the token Token. The order of the TokenInfoTuples is not defined. The following TokenItems are returned: category, column, length, line, symbol, and text. See token_info/2 [page 156] for information about specific TokenInfoTuples.
Note that if token_info(Token, TokenItem) returns undefined for some TokenItem in the list above, the item is not included in TokenInfo.

token_info(Token, TokenItemSpec) -> TokenInfo
Types:
• Token = token()
• TokenItemSpec = TokenItem | [TokenItem]
• TokenInfo = TokenInfoTuple | undefined | [TokenInfoTuple]
• TokenInfoTuple = {TokenItem, Info}
• TokenItem = atom()
• Info = term()
Returns a list containing information about the token Token. If TokenItemSpec is a single TokenItem, the returned value is the corresponding TokenInfoTuple, or undefined if the TokenItem has no value. If TokenItemSpec is a list of TokenItem, the result is a list of TokenInfoTuple. The TokenInfoTuple will appear with the corresponding TokenItems in the same order as the TokenItems appeared in the list of TokenItems. TokenItems with no value are not included in the list of TokenInfoTuple. The following TokenInfoTuples with corresponding TokenItems are valid:

- `{category, category()}` The category of the token.
- `{column, column()}` The column where the token begins.
- `{length, integer() > 0}` The length of the token's text.
- `{line, line()}` The line where the token begins.
- `{location, location()}` The line and column where the token begins, or just the line if the column unknown.
- `{symbol, symbol()}` The token's symbol.
- `{text, string()}` The token's text.

**attributes_info(Attributes) -> AttributesInfo**

Types:
- Attributes = attributes()
- AttributesInfo = [AttributeInfoTuple]
- AttributeInfoTuple = {AttributeItem, Info}
- AttributeItem = atom()
- Info = term()

Returns a list containing information about the token attributes Attributes. The order of the AttributeInfoTuples is not defined. The following AttributeItems are returned: column, length, line, and text. See attributes_info/2 [page 157] for information about specific AttributeInfoTuples.

Note that if attributes_info(Token, AttributeItem) returns undefined for some AttributeItem in the list above, the item is not included in AttributesInfo.

**attributes_info(Attributes, AttributeItemSpec) -> AttributesInfo**

Types:
- Attributes = attributes()
- AttributeItemSpec = AttributeItem | [AttributeItem]
- AttributesInfo = AttributeInfoTuple | undefined | [AttributeInfoTuple]
- AttributeInfoTuple = {AttributeItem, Info}
- AttributeItem = atom()
- Info = term()
Returns a list containing information about the token attributes Attributes. If AttributeItemSpec is a single AttributeItem, the returned value is the corresponding AttributeInfoTuple, or undefined if the AttributeItem has no value. If AttributeItemSpec is a list of AttributeItem, the result is a list of AttributeInfoTuple. The AttributeInfoTuples will appear with the corresponding AttributeItems in the same order as the AttributeItems appeared in the list of AttributeItemSpecs. AttributeItems with no value are not included in the list of AttributeInfoTuples.

The following AttributeInfoTuples with corresponding AttributeItems are valid:

- \{column, column()\} The column where the token begins.
- \{length, integer() > 0\} The length of the token's text.
- \{line, line()\} The line where the token begins.
- \{location, location()\} The line and column where the token begins, or just the line if the column unknown.
- \{text, string()\} The token's text.

```erl
set_attribute(AttributeItem, Attributes, SetAttributeFun) -> AttributesInfo
```

Types:
- AttributeItem = line
- Attributes = attributes()
- SetAttributeFun = set_attribute_fun()

Sets the value of the line attribute of the token attributes Attributes. The SetAttributeFun is called with the value of the line attribute, and is to return the new value of the line attribute.

```erl
format_error(ErrorDescriptor) -> string()
```

Types:
- ErrorDescriptor = errordesc()

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

## Error Information

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

```erl
{ErrorLocation, Module, ErrorDescriptor}
```

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

```erl
Module:format_error(ErrorDescriptor)
```
Notes

The continuation of the first call to the re-entrant input functions must be \[]. Refer to Armstrong, Virding and Williams, 'Concurrent Programming in Erlang', Chapter 13, for a complete description of how the re-entrant input scheme works.

See Also

io(3) [page 246], erl_parse(3) [page 148]
The erl_tar module archives and extract files to and from a tar file. The tar file format is the POSIX extended tar file format specified in IEEE Std 1003.1 and ISO/IEC9945-1. That is the same format as used by tar program on Solaris, but is not the same as used by the GNU tar program.

By convention, the name of a tar file should end in ".tar". To abide to the convention, you'll need to add ".tar" yourself to the name.

Tar files can be created in one operation using the create/2 [page 162] or create/3 [page 162] function. Alternatively, for more control, the open [page 164], add/3,4 [page 161], and close/1 [page 161] functions can be used.

To extract all files from a tar file, use the extract/1 [page 162] function. To extract only some files or to be able to specify some more options, use the extract/2 [page 163] function.

To return a list of the files in a tar file, use either the table/1 [page 164] or table/2 [page 164] function. To print a list of files to the Erlang shell, use either the t/1 [page 164] or tt/1 [page 165] function.

To convert an error term returned from one of the functions above to a readable message, use the format_error/1 [page 163] 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 164].

dereference  By default, symbolic links will be stored as symbolic links in the tar file. Use the dereference option to override the default and store the file that the symbolic link points to into the tar file.
verbose  Print an informational message about the file being added.

add(TarDescriptor, FilenameOrBin, NameInArchive, Options) -> RetValue

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

The `add/4` function adds a file to a tar file that has been opened for writing by `open/1` [page 164]. It accepts the same options as `add/3` [page 161].

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 164].

create(Name, FileList) -> RetValue

Types:
- Name = filename()
- FileList = [Filename | {NameInArchive, binary()}, {NameInArchive, Filename}]
- Filename = filename()
- NameInArchive = filename()
The `create/2` function creates a tar file and archives the files whose names are given in `FileList` into it. The files may either be read from disk or given as binaries.

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

**Types:**
- `Name = filename()`
- `FileList = [\text{Filename} | \text{NameInArchive, binary()}], [\text{NameInArchive, Filename}]`
- `Filename = filename()`
- `NameInArchive = filename()`
- `OptionList = [\text{Option}]`
- `Option = \text{compressed} | \text{cooked} | \text{dereference} | \text{verbose}`

**RetVal: ok | \text{error, \{Name, Reason\}}**

\text{Reason} = term()

The `create/3` function creates a tar file and archives the files whose names are given in `FileList` into it. The files may either be read from disk or given as binaries. The options in `OptionList` modify the defaults as follows:
- **compressed** The entire tar file will be compressed, as if it has been run through the `gzip` program. To abide to the convention that a compressed tar file should end in ".tar.gz" or ".tgz", you'll need to add the appropriate extension yourself.
- **cooked** By default, the `open/2` function will open the tar file in `raw` mode, which is faster but does not allow a remote (erlang) file server to be used. Adding `cooked` to the mode list will override the default and open the tar file without the `raw` option.
- **dereference** By default, symbolic links will be stored as symbolic links in the tar file. Use the `dereference` option to override the default and store the file that the symbolic link points to into the tar file.
- **verbose** Print an informational message about each file being added.

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

**Types:**
- `Name = filename()`
- `RetVal = ok | \text{error, \{Name, Reason\}}`
- `Reason = term()`

The `extract/1` function extracts all files from a tar archive.
- If the `Name` argument is given as `{binary, Binary}`, the contents of the binary is assumed to be a tar archive.
- If the `Name` argument is given as `{file, Fd}`, `Fd` is assumed to be a file descriptor returned from the `file:open/2` function.
- Otherwise, `Name` should be a filename.

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

**Types:**
- `Name = filename() | \text{binary, Binary} | \text{file, Fd}`
- `Binary = binary()`
- `Fd = file:descriptor()`
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.

- memory Instead of extracting to a directory, the memory option will give the result as a list of tuples {Filename, Binary}, where Binary is a binary containing the extracted data of the file named Filename in the tar file.

- keep_old_files By default, all existing files with the same name as file in the tar file will be overwritten. Given the keep_old_files option, the extract/2 function will not overwrite any existing files.

- verbose Print an informational message as each file is being extracted.

format_error(Reason) -> string()

types:
- Reason = term()

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

open(Name, OpenModeList) -> RetValue

types:
- Name = filename()
The open/2 function creates a tar file for writing. (Any existing file with the same name will be truncated.) 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.

Except for the write atom the following atoms may be added to OpenModeList:

- **compressed**: The entire tar file will be compressed, as if it has been run through the gzip program. To abide to the convention that a compressed tar file should end in ".tar.gz" or ".tgz", you'll need to add the appropriate extension yourself.
- **cooked**: By default, the open/2 function will open the tar file in raw mode, which is faster but does not allow a remote (erlang) file server to be used. Adding cooked to the mode list will override the default and open the tar file without the raw option.

Use the add/3,4 [page 161] functions to add one file at the time into an opened tar file. When you are finished adding files, use the close [page 161] function to close the tar file.

**Warning:**
The TarDescriptor term is not a file descriptor. You should not rely on the specific contents of the TarDescriptor term, as it may change in future versions as more features are added to the erl_tar module.

table(Name) -> RetValue
Types:
- Name = filename()
- RetValue = {ok,[string()]} | {error,Name,Reason}
- Reason = term()

The table/1 function retrieves the names of all files in the tar file Name.

table(Name, Options) -> RetValue
Types:
- Name = filename()

The table/2 function retrieves the names of all files in the tar file Name.

t(Name) -> RetValue
Types:
- Name = filename()

The t/1 function prints the names of all files in the tar file Name to the Erlang shell. (Similar to “tart.”)
tt(Name)

Types:
- Name = filename()

The `tt/1` function prints names and information about all files in the tar file `Name` to the Erlang shell. (Similar to "tartv").
This module is an interface to the Erlang built-in term storage BIFs. These provide the ability to store very large quantities of data in an Erlang runtime system, and to have constant access time to the data. (In the case of ordered_set, see below, access time is proportional to the logarithm of the number of objects stored).

Data is organized as a set of dynamic tables, which can store tuples. Each table is created by a process. When the process terminates, the table is automatically destroyed. Every table has access rights set at creation.

Tables are divided into four different types, set, ordered_set, bag and duplicate_bag. A set or ordered_set table can only have one object associated with each key. A bag or duplicate_bag can have many objects associated with each key.

The number of tables stored at one Erlang node is limited. The current default limit is approximately 1400 tables. The upper limit can be increased by setting the environment variable ERL_MAX_ETS_TABLES before starting the Erlang runtime system (i.e. with the -env option to erl/erl). The actual limit may be slightly higher than the one specified, but never lower.

Note that there is no automatic garbage collection for tables. Even if there are no references to a table from any process, it will not automatically be destroyed unless the owner process terminates. It can be destroyed explicitly by using delete/1.

Some implementation details:

- In the current implementation, every object insert and look-up operation results in a copy of the object.
- This module provides very limited support for concurrent updates. No locking is available, but the safe_fixtable/2 function can be used to guarantee that a sequence of first/1 and next/2 calls will traverse the table without errors and that each object in the table is visited exactly once, even if another process (or the same process) simultaneously deletes or inserts objects into the table. Nothing more is guaranteed; in particular any object inserted during a traversal may be visited in the traversal.

'\$end_of_table' should not be used as a key since this atom is used to mark the end of the table when using first/next.

Also worth noting is the subtle difference between matching and comparing equal, which is demonstrated by the different table types set and ordered_set. Two Erlang terms match if they are of the same type and have the same value, so that 1 matches 1, but not 1.0 (as 1.0 is a float() and not an integer()). Two Erlang terms compare equal if they either are of the same type and value, or if both are numeric types and extend to the same value, so that 1 compares equal to both 1 and 1.0. The ordered_set works on the Erlang term order and there is no defined order between an integer() and a float() that extends to the same value, hence the key 1 and the key 1.0 are regarded as equal in an ordered_set table.
In general, the functions below will exit with reason badarg if any argument is of the wrong format, or if the table identifier is invalid.

**Match Specifications**

Some of the functions uses a match specification, match_spec. A brief explanation is given in select/2 [page 181]. For a detailed description, see the chapter “Match specifications in Erlang” in ERTS User’s Guide.

**DATA TYPES**

```
match_spec()   a match specification, see above

tid()         a table identifier, as returned by new/2
```

**Exports**

```
all() -> [Tab]
Types:
  • Tab = tid() | atom()
Returns a list of all tables at the node. Named tables are given by their names, unnamed tables are given by their table identifiers.

delete(Tab) -> true
Types:
  • Tab = tid() | atom()
Deletes the entire table Tab.

delete(Tab, Key) -> true
Types:
  • Tab = tid() | atom()
  • Key = term()
Deletes all objects with the key Key from the table Tab.

delete_all_objects(Tab) -> true
Types:
  • Tab = tid() | atom()
Delete all objects in the ETS table Tab. The deletion is atomic.

delete_object(Tab, Object) -> true
```
Types:
- Tab = tid() | atom()
- Object = tuple()

Delete the exact object Object from the ETS table, leaving objects with the same key but other differences (useful for type bag).

file2tab(Filename) -> {ok,Tab} | {error,Reason}

Types:
- Filename = string() | atom()
- Tab = tid() | atom()
- Reason = term()

Reads a file produced by tab2file/2 [page 184] or tab2file/3 [page 184] and creates the corresponding table Tab.
Equivalent to file2tab(Filename, []).

file2tab(Filename,Options) -> {ok,Tab} | {error,Reason}

Types:
- Filename = string() | atom()
- Tab = tid() | atom()
- Options = [Option]
- Option = {verify, bool()}
- Reason = term()

Reads a file produced by tab2file/2 [page 184] or tab2file/3 [page 184] and creates the corresponding table Tab.
The currently only supported option is {verify, bool()}. If verification is turned on (by means of specifying {verify, true}), the function utilizes whatever information is present in the file to assert that the information is not damaged. How this is done depends on which extended_info was written using tab2file/3 [page 184].

If no extended_info is present in the file and {verify, true} is specified, the number of objects written is compared to the size of the original table when the dump was started. This might make verification fail if the table was public and objects were added or removed while the table was dumped to file. To avoid this type of problems, either do not verify files dumped while updated simultaneously or use the {extended_info, [object_count]} option to tab2file/3 [page 184], which extends the information in the file with the number of objects actually written.

If verification is turned on and the file was written with the option {extended_info, [md5sum]}, reading the file is slower and consumes radically more CPU time than otherwise.

{verify, false} is the default.

first(Tab) -> Key | '$end_of_table'

Types:
- Tab = tid() | atom()
- Key = term()
Returns the first key Key in the table Tab. If the table is of the ordered_set type, the first key in Erlang term order will be returned. If the table is of any other type, the first key according to the table's internal order will be returned. If the table is empty, `$end_of_table` will be returned.

Use `next/2` to find subsequent keys in the table.

foldl(Function, Acc0, Tab) -> Acc1

Types:
- Function = fun(A, AccIn) -> AccOut
- Tab = tid() | atom()
- Acc0 = Acc1 = AccIn = AccOut = term()

Acc0 is returned if the table is empty. This function is similar to `lists:foldl/3`. The order in which the elements of the table are traversed is unspecified, except for tables of type `ordered_set`, for which they are traversed first to last.

If Function inserts objects into the table, or another process inserts objects into the table, those objects may (depending on key ordering) be included in the traversal.

foldr(Function, Acc0, Tab) -> Acc1

Types:
- Function = fun(A, AccIn) -> AccOut
- Tab = tid() | atom()
- Acc0 = Acc1 = AccIn = AccOut = term()

Acc0 is returned if the table is empty. This function is similar to `lists:foldr/3`. The order in which the elements of the table are traversed is unspecified, except for tables of type `ordered_set`, for which they are traversed last to first.

If Function inserts objects into the table, or another process inserts objects into the table, those objects may (depending on key ordering) be included in the traversal.

from_dets(Tab, DetsTab) -> true

Types:
- Tab = tid() | atom()
- DetsTab = atom()

Fills an already created ETS table with the objects in the already opened Dets table named DetsTab. The existing objects of the ETS table are kept unless overwritten.

Throws a badarg error if any of the tables does not exist or the dets table is not open.

fun2ms(LiteralFun) -> MatchSpec

Types:
- LiteralFun – see below
- MatchSpec = match_spec()
Pseudo function that by means of a parse_transform translates LiteralFun typed as parameter in the function call to a match_spec [page 167]. With “literal” is meant that the fun needs to textually be written as the parameter of the function, it cannot be held in a variable which in turn is passed to the function).

The parse transform is implemented in the module ms_transform and the source must include the file ms_transform.hrl in stdlib for this pseudo function to work. Failing to include the hrl file in the source will result in a runtime error, not a compile time ditto. The include file is easiest included by adding the line
~include_lib("stdlib/include/ms_transform.hrl"). to the source file.

The fun is very restricted, it can take only a single parameter (the object to match): a sole variable or a tuple. It needs to use the is XXX guard tests. Language constructs that have no representation in a match_spec (like if, case, receive etc) are not allowed.

The return value is the resulting match_spec.

Example:

1> fun2ms(fun([M,N]) when N > 3 -> M end).

[[$1','$2'],[['>','$2',3]],[['$1']]]

Variables from the environment can be imported, so that this works:

2> X=3.

3> fun2ms(fun([M,N]) when N > X -> M end).

[[$1','$2'],[['>','$2',{const,3}]],[['$1']]]

The imported variables will be replaced by match_spec const expressions, which is consistent with the static scoping for Erlang funs. Local or global function calls can not be in the guard or body of the fun however. Calls to builtin match_spec functions of course is allowed:

4> 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> 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 285].
i() -> ok

Displays information about all ETS tables on tty.

i(Tab) -> ok

Types:
- Tab = tid() | atom()

Browses the table Tab on tty.

info(Tab) -> [{Item, Value}] | undefined

Types:
- Tab = tid() | atom()
- Item = atom(), see below
- Value = term(), see below

Returns information about the table Tab as a list of \{Item, Value\} tuples. If Tab has the correct type for a table identifier, but does not refer to an existing ETS table, undefined is returned. If Tab is not of the correct type, this function fails with reason badarg.

- Item=memory, Value=int()
  The number of words allocated to the table.
- Item=owner, Value=pid()
  The pid of the owner of the table.
- Item=name, Value=atom()
  The name of the table.
- Item=size, Value=int()
  The number of objects inserted in the table.
- Item=node, Value=atom()
  The node where the table is stored. This field is no longer meaningful as tables cannot be accessed from other nodes.
- Item=named_table, Value=true|false
  Indicates if the table is named or not.
- Item=type, Value=set|ordered_set|bag|duplicate_bag
  The table type.
- Item=keypos, Value=int()
  The key position.
- Item=protection, Value=public|protected|private
  The table access rights.

info(Tab, Item) -> Value | undefined

Types:
- Tab = tid() | atom()
- Item, Value - see below
Returns the information associated with \texttt{Item} for the table \texttt{Tab}, or returns \texttt{undefined} if \texttt{Tab} does not refer an existing ETS table. If \texttt{Tab} is not of the correct type, or if \texttt{Item} is not one of the allowed values, this function fails with reason badarg.

**Warning:**

In R11B and earlier, this function would not fail but return \texttt{undefined} for invalid values for \texttt{Item}.

In addition to the \{\texttt{Item},\texttt{Value}\} pairs defined for \texttt{info/1}, the following items are allowed:

- \texttt{Item}=\texttt{fixed}, \texttt{Value}=true|false
  
  Indicates if the table is fixed by any process or not.

- \texttt{Item}=\texttt{safe_fixed}, \texttt{Value}=[\texttt{FirstFixed},\texttt{Info}]false
  
  If the table has been fixed using \texttt{safe_fixtable/2}, the call returns a tuple where \texttt{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. \texttt{Info} is a possibly empty lists of tuples \{\texttt{Pid},\texttt{RefCount}\}, one tuple for every process the table is fixed by right now. \texttt{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 \texttt{false}.

\texttt{init_table(Name, InitFun) -> true}

Types:

- \texttt{Name} = atom()
- \texttt{InitFun} = fun(\texttt{Arg}) \to \texttt{Res}
- \texttt{Arg} = read | close
- \texttt{Res} = end_of_input | [{\texttt{object()}}, \texttt{InitFun}] | term()

Replaces the existing objects of the table \texttt{Tab} with objects created by calling the input function \texttt{InitFun}, see below. This function is provided for compatibility with the \texttt{dets} module, it is not more efficient than filling a table by using \texttt{ets:insert/2}.

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

If the type of the table is \texttt{set} and there is more than one object with a given key, one of the objects is chosen. This is not necessarily the last object with the given key in the sequence of objects returned by the input functions. This holds also for duplicated objects stored in tables of type \texttt{bag}.

\texttt{insert(Tab, Object0rObjects) -> true}

Types:

- \texttt{Tab} = tid() \mid atom()
- \texttt{Object0rObjects} = tuple() \mid [\texttt{tuple()}]
Inserts the object or all of the objects in the list ObjectOrObjects into the table Tab. If the table is a set and the key of the inserted objects matches the key of any object in the table, the old object will be replaced. If the table is an ordered_set and the key of the inserted object compares equal to the key of any object in the table, the old object is also replaced. If the list contains more than one object with matching keys and the table is a set, one will be inserted, which one is not defined. The same thing holds for ordered_set, but will also happen if the keys compare equal.

\[\text{insert\_new(Tab, ObjectOrObjects)} \rightarrow \text{bool()}\]

Types:
- \(\text{Tab} = \text{tid()} \mid \text{atom()}\)
- \(\text{ObjectOrObjects} = \text{tuple()} \mid [\text{tuple()}]\)

This function works exactly like \text{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.

\[\text{is\_compiled\_ms(Term)} \rightarrow \text{bool()}\]

Types:
- \(\text{Term} = \text{term()}\)

This function is used to check if a term is a valid compiled match_spec [page 167]. 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:

\[\text{ets:is\_compiled\_ms(ets:match\_spec\_compile([\{'\_\',[],[true]\}])}}\]

will yield true, while the following expressions:

\[\text{MS} = \text{ets:match\_spec\_compile([\{'\_\',[],[true]\}])},\]
\[\text{Broken} = \text{binary\_to\_term(term\_to\_binary(MS))},\]
\[\text{ets:is\_compiled\_ms(Broken)}\]

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.

\[\text{last(Tab)} \rightarrow \text{Key} \mid '\$end\_of\_table'\]

Types:
- \(\text{Tab} = \text{tid()} \mid \text{atom()}\)
- \(\text{Key} = \text{term()}\)
Returns the last key Key according to Erlang term order in the table Tab of the ordered_set type. If the table is of any other type, the function is synonymous to first/2. If the table is empty, '$end_of_table' is returned.

Use prev/2 to find preceding keys in the table.

lookup(Tab, Key) -> [Object]

Types:
- Tab = tid() | atom()
- Key = term()
- Object = tuple()

Returns a list of all objects with the key Key in the table Tab.

In the case of set, bag and duplicate_bag, an object is returned only if the given key matches the key of the object in the table. If the table is an ordered_set however, an object is returned if the key given compares equal to the key of an object in the table. The difference being the same as between =:= and ==. As an example, one might insert an object with the integer()1 as a key in an ordered_set and get the object returned as a result of doing a lookup/2 with the float()1.0 as the key to search for.

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

Note that the time order of object insertions is preserved; The first object inserted with the given key will be first in the resulting list, and so on.

Insert and look-up times in tables of type set, bag and duplicate_bag are constant, regardless of the size of the table. For the ordered_set data-type, time is proportional to the (binary) logarithm of the number of objects.

lookup_element(Tab, Key, Pos) -> Elem

Types:
- Tab = tid() | atom()
- Key = term()
- Pos = int()
- Elem = term() | [term()]

If the table Tab is of type set or ordered_set, the function returns the Pos:th element of the object with the key Key.

If the table is of type bag or duplicate_bag, the functions returns a list with the Pos:th element of every object with the key Key.

If no object with the key Key exists, the function will exit with reason badarg.

The difference between set, bag and duplicate_bag on one hand, and ordered_set on the other, regarding the fact that ordered_set's view keys as equal when they compare equal whereas the other table types only regard them equal when they match, naturally holds for lookup_element as well as for lookup.

match(Tab, Pattern) -> [Match]

Types:
- Tab = tid() | atom()
• Pattern = tuple()
• Match = [term()]

Matches the objects in the table Tab against the pattern Pattern.

A pattern is a term that may contain:

• bound parts (Erlang terms),
• "." which matches any Erlang term, and
• pattern variables: "$N" where N=0,1,...

The function returns a list with one element for each matching object, where each element is an ordered list of pattern variable bindings. An example:

6> ets:match(T, '$1'). % Matches every object in the table
[[[rufsen,dog,7]],[[brunte,horse,5]],[[ludde,dog,5]]]
7> ets:match(T, {'-','dog','$1'}). [[7],[5]]
8> ets:match(T, {'-','cow','$1'}). []

If the key is specified in the pattern, the match is very efficient. If the key is not specified, i.e. if it is a variable or an underscore, the entire table must be searched. The search time can be substantial if the table is very large.

On tables of the ordered_set type, the result is in the same order as in a first/next traversal.

match(Tab, Pattern, Limit) -> {[Match],Continuation} | '$end_of_table'

Types:
• Tab = tid() | atom()
• Pattern = tuple()
• Match = [term()]
• Continuation = term()

Works like ets:match/2 but only returns a limited (Limit) number of matching objects. The Continuation term can then be used in subsequent calls to ets:match/1 to get the next chunk of matching objects. This is a space efficient way to work on objects in a table which is still faster than traversing the table object by object using ets:first/1 and ets:next/1.

'$end_of_table' is returned if the table is empty.

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

Types:
• Match = [term()]
• Continuation = term()

Continues a match started with ets:match/3. The next chunk of the size given in the initial ets:match/3 call is returned together with a new Continuation that can be used in subsequent calls to this function.

'$end_of_table' is returned when there are no more objects in the table.

match_delete(Tab, Pattern) -> true
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]

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'

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'

Continues a match started with ets:match_object/3. The next chunk of the size given in the initial ets:match_object/3 call is returned together with a new Continuation that can be used in subsequent calls to this function.

'end_of_table' is returned when there are no more objects in the table.

match_spec_compile(MatchSpec) -> CompiledMatchSpec

Continues a match started with ets:match_object/3. The next chunk of the size given in the initial ets:match_object/3 call is returned together with a new Continuation that can be used in subsequent calls to this function.

'end_of_table' is returned when there are no more objects in the table.
This function transforms a match_spec [page 167] into an internal representation that can be used in subsequent calls to ets:match_spec_run/2. The internal representation is opaque and can not be converted to external term format and then back again without losing its properties (meaning it can not be sent to a process on another node and still remain a valid compiled match_spec, nor can it be stored on disk). The validity of a compiled match_spec can be checked using ets:is_compiled_ms/1.

If the term MatchSpec can not be compiled (does not represent a valid match_spec), a badarg fault is thrown.

**Note:**
This function has limited use in normal code, it is used by Dets to perform the dets:select operations.

match_spec_run(List,CompiledMatchSpec) -> list()

Types:
- List = [ tuple() ]
- CompiledMatchSpec = comp_match_spec()

This function executes the matching specified in a compiled match_spec [page 167] on a list of tuples. The CompiledMatchSpec term should be the result of a call to ets:match_spec_compile/1 and is hence the internal representation of the match_spec one wants to use.

The matching will be executed on each element in List and the function returns a list containing all results. If an element in List does not match, nothing is returned for that element. The length of the result list is therefore equal or less than the the length of the parameter List. The two calls in the following example will give the same result (but certainly not the same execution time...):

Table = ets:new...
MatchSpec = ....
% The following call...
ets:match_spec_run(ets:tab2list(Table), ets:match_spec_compile(MatchSpec)),
% ...will give the same result as the more common (and more efficient) ets:select(Table,MatchSpec),

**Note:**
This function has limited use in normal code, it is used by Dets to perform the dets:select operations and by Mnesia during transactions.

member(Tab, Key) -> true | false

Types:
- Tab = tid() | atom()
- Key = term()
Works like `lookup/2`, but does not return the objects. The function returns `true` if one or more elements in the table has the key `Key`, `false` otherwise.

```
new(Name, Options) -> tid() | atom()
```

Types:
- `Name` = `atom()`
- `Options` = [ `Option` ]
  - `Option` = `Type` | `Access` | named_table | `{keypos,Pos}`
- `Type` = set | ordered_set | bag | duplicate_bag
- `Access` = public | protected | private
- `Pos` = `int()`

Creates a new table and returns a table identifier which can be used in subsequent operations. The table identifier can be sent to other processes so that a table can be shared between different processes within a node.

The parameter `Options` is a list of atoms which specifies table type, access rights, key position and if the table is named or not. If one or more options are left out, the default values are used. This means that not specifying any options ([ ]) is the same as specifying `[set,protected,{keypos,1}]`.

- `set` The table is a `set` table - one key, one object, no order among objects. This is the default table type.
- `ordered_set` The table is an `ordered_set` table - one key, one object, ordered in Erlang term order, which is the order implied by the `<` and `>` operators. Tables of this type have a somewhat different behavior in some situations than tables of the other types. Most notably the `ordered_set` tables regard keys as equal when they compare equal, not only when they match. This means that to an `ordered_set`, the `integer()1` and the `float()1.0` are regarded as equal. This also means that the key used to lookup an element not necessarily matches the key in the elements returned, if `float()`'s and `integer()`'s are mixed in keys of a table.
- `bag` The table is a `bag` table which can have many objects, but only one instance of each object, per key.
- `duplicate_bag` The table is a `duplicate_bag` table which can have many objects, including multiple copies of the same object, per key.
- `public` Any process may read or write to the table.
- `protected` The owner process can read and write to the table. Other processes can only read the table. This is the default setting for the access rights.
- `private` Only the owner process can read or write to the table.
- `named_table` If this option is present, the name `Name` is associated with the table identifier. The name can then be used instead of the table identifier in subsequent operations.
- `{keypos,Pos}` 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 and each object in the table will only be returned once, even if objects are
removed or inserted during the traversal. The keys for new objects inserted during the
traversal may be returned by next/2 [page 178] (it depends on the internal ordering of
the keys). An example:
clean_all_with_value(Tab,X) ->
    safe_fixtable(Tab,true),
    clean_all_with_value(Tab,X,ets:first(Tab)),
    safe_fixtable(Tab,false).

clean_all_with_value(Tab,X,'$end_of_table') ->
    true;
clean_all_with_value(Tab,X,Key) ->
    case ets:lookup(Tab,Key) of
      [{Key,X}] ->
        ets:delete(Tab,Key);
      _ ->
        true
    end,
    clean_all_with_value(Tab,X,ets:next(Tab,Key)).

Note that no deleted objects are actually removed from a fixed table until it has been
released. If a process fixes a table but never releases it, the memory used by the deleted
objects will never be freed. The performance of operations on the table will also
degrade significantly.

Use info/2 to retrieve information about which processes have fixed which tables. A
system with a lot of processes fixing tables may need a monitor which sends alarms
when tables have been fixed for too long.

Note that for tables of the ordered_set type, safe_fixtable/2 is not necessary as calls
to first/1 and next/2 will always succeed.

select(Tab, MatchSpec) -> [Match]

Types:
- Tab = tid() | atom()
- Match = term()
- MatchSpec = match_spec()

Matches the objects in the table Tab using a match_spec [page 167]. This is a more
general call than the ets:match/2 and ets:match_object/2 calls. In its simplest forms
the match_specs look like this:

- MatchSpec = [MatchFunction]
- MatchFunction = {MatchHead, [Guard], [Result]}
- MatchHead = “Pattern as in ets:match”
- Guard = [“Guardtest name”, ...]
- Result = “Term construct”

This means that the match_spec is always a list of one or more tuples (of arity 3). The
tuples first element should be a pattern as described in the documentation of
ets:match/2. The second element of the tuple should be a list of 0 or more guard tests
(described below). The third element of the tuple should be a list containing a
description of the value to actually return. In almost all normal cases the list contains
exactly one term which fully describes the value to return for each object.

The return value is constructed using the “match variables” bound in the MatchHead or
using the special match variables ‘$-‘ (the whole matching object) and ‘$$‘ (all match
variables in a list), so that the following ets:match/2 expression:
ets\(\text{match}(\text{Tab}, \{\text{'$1', '$2', '$3'}\})\)

is exactly equivalent to:

\(\text{ets:select}(\text{Tab}, [[\{\text{'$1', '$2', '$3'}, [], ['$$']\}], in\) [123x758]ets:match(Tab,\{'$1','$2','$3'\})

- and the following \text{ets:match_object/2} call:

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

is exactly equivalent to

\(\text{ets:select}(\text{Tab}, [[\{\text{'$1', '$2', '$1'}, [], ['$_']\}], in\) [123x741]Composite terms can be constructed in the Result part either by simply writing a list, so that this code:

\(\text{ets:select}(\text{Tab}, [[\{\text{'$1', '$2', '$3'}, [], ['$$']\}], in\) [123x723]\(\text{ets:select}(\text{Tab}, [[\{\text{'$1', '$2', '$3'}, [], ['$$']\}], in\) [123x705]i.e. all the bound variables in the match head as a list. If tuples are to be constructed, one has to write a tuple of arity 1 with the single element in the tuple being the tuple one wants to construct (as an ordinary tuple could be mistaken for a guard). Therefore the following call:

\(\text{ets:select}(\text{Tab}, [[\{\text{'$1', '$2', '$1'}, [], ['$_']\}], in\) [123x687]gives the same output as:

\(\text{ets:select}(\text{Tab}, [[\{\text{'$1', '$2', '$3'}, [], ['$_']\}], in\) [123x669]this syntax is equivalent to the syntax used in the trace patterns (see \([\text{dbg}(3)]\)).

The \text{Guard} sections 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:

\text{is}\_\text{integer}(X)\), is\_\text{integer}(Y), X + Y < 4711

is expressed like this (\(X\) replaced with \text{'$1'}\) and \(Y\) with \text{'$2'}\):

\([\text{is\_}\text{integer}, \text{'$1'}], \text{is\_}\text{integer}, \text{'$2'}\], \{<, '+', '$1$', '$2$', 4711\]}

On tables of the \text{ordered}\_\text{set} type, objects are visited in the same order as in a \text{first/next} traversal. This means that the match specification will be executed against objects with keys in the \text{first/next} order and the corresponding result list will be in the order of that execution.

\text{select}(\text{Tab}, \text{MatchSpec}, \text{Limit}) \rightarrow \{[\text{Match}], \text{Continuation}\} \mid \text{'$end\_of\_table'}\)

Types:

- \(\text{Tab} = \text{tid() | atom()}\)
- \(\text{Match} = \text{term()}\)
- \(\text{MatchSpec} = \text{match\_spec()}\)
• 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 167]. If the match_spec returns true for an object, that object is removed from the table. For any other result from the match_spec the object is retained. This is a more general call than the ets:match_delete/2 call.
The function returns the number of objects actually deleted from the table.

**Note:**
The match_spec has to return the atom true if the object is to be deleted. No other return value will get the object deleted, why one can not use the same match specification for looking up elements as for deleting them.

select_count(Tab, MatchSpec) -> NumMatched
Types:
  • Tab = tid() | atom()
  • Object = tuple()
  • MatchSpec = match_spec()
  • NumMatched = integer()
Matches the objects in the table Tab using a match_spec [page 167]. 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.

\[
\text{slot(Tab, I)} \rightarrow \text{[Object]} \mid \text{'$end_of_table'}
\]

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 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.

Equivalent to \text{tab2file(Tab, Filename, [])}

\[
\text{tab2file(Tab, Filename, Options)} \rightarrow \text{ok} \mid \text{[error,Reason]}
\]

Types:
- \(\text{Tab} = \text{tid()} \mid \text{atom()}
- \text{Filename} = \text{string()} \mid \text{atom()}
- \text{Options} = \text{[Option]}
- \text{Option} = \text{[extended_info, [ExtInfo]]}
- \text{ExtInfo} = \text{object_count} \mid \text{md5sum}
- \text{Reason} = \text{term()}

Dumps the table Tab to the file Filename.

When dumping the table, certain information about the table is dumped to a header at the beginning of the dump. This information contains data about the table type, name, protection, size, version and if it's a named table. It also contains notes about what extended information is added to the file, which can be a count of the objects in the file or a MD5 sum of the header and records in the file.

The size field in the header might not correspond to the actual number of records in the file if the table is public and records are added or removed from the table during dumping. Public tables updated during dump, and that one wants to verify when reading, needs at least one field of extended information for the read verification process to be reliable later.

The extended_info option specifies what extra information is written to the table dump:

- **object_count** The number of objects actually written to the file is noted in the file footer, why verification of file truncation is possible even if the file was updated during dump.
- **md5sum** The header and objects in the file are checksummed using the built in MD5 functions. The MD5 sum of all objects is written in the file footer, so that verification while reading will detect the slightest bitflip in the file data. Using this costs a fair amount of CPU time.

Whenever the extended_info option is used, it results in a file not readable by versions of ets prior to that in stdlib-1.15.1

**tab2list(Tab) -> [Object]**

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

Returns a list of all objects in the table Tab.

**tabfile_info(Filename) -> {ok, TableInfo} | {error, Reason}**

Types:
- Filename = string() | atom()
- TableInfo = [InfoItem]
- InfoItem = {InfoTag, term()}
  - InfoTag = name | type | protection | named_table | keypos | size | extended_info | version
  - Reason = term()

Returns information about the table dumped to file by tab2file/2 [page 184] or tab2file/3 [page 184]

The following items are returned:

- **name** The name of the dumped table. If the table was a named table, a table with the same name cannot exist when the table is loaded from file with file2tab/2 [page 168]. If the table is not saved as a named table, this field has no significance at all when loading the table from file.
type The ets type of the dumped table (i.e. set, bag, duplicate_bag or ordered_set).
This type will be used when loading the table again.

protection The protection of the dumped table (i.e. private, protected or public).
A table loaded from the file will get the same protection.

named_table true if the table was a named table when dumped to file, otherwise false. Note that when a named table is loaded from a file, there cannot exist a table in the system with the same name.

keypos The keypos of the table dumped to file, which will be used when loading the table again.

size The number of objects in the table when the table dump to file started, which in case of a public table need not correspond to the number of objects actually saved to the file, as objects might have been added or deleted by another process during table dump.

extended_info The extended information written in the file footer to allow stronger verification during table loading from file, as specified to tab2file/3 [page 184]. Note that this function only tells which information is present, not the values in the file footer. The value is a list containing one or more of the atoms object_count and md5sum.

version A tuple [Major,Minor] containing the major and minor version of the file format for ets table dumps. This version field was added beginning with stdlib-1.5.1, files dumped with older versions will return [0,0] in this field.

An error is returned if the file is inaccessible, badly damaged or not an file produced with tab2file/2 [page 184] or tab2file/3 [page 184].

table(Tab [, Options]) -> QueryHandle

Types:
- Tab = tid() | atom()
- QueryHandle = -a query handle, see qlc(3)-
- Options = [Option] | Option
  - Option = {n_objects, NObjects} | {traverse, TraverseMethod}
  - NObjects = default | integer() > 0
  - TraverseMethod = first | next | last | prev | select | {select, MatchSpec}
  - MatchSpec = match_spec()

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

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

- first_next. The table is traversed one key at a time by calling ets:first/1 and ets:next/2.
- last_prev. The table is traversed one key at a time by calling ets:last/1 and ets:prev/2.
• select. The table is traversed by calling ets:select/3 and ets:select/1. The option \_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 167] (the second argument of select/3) is assembled by QLC: simple filters are translated into equivalent match_specs while more complicated filters have to be applied to all objects returned by select/3 given a match_spec that matches all objects.

• {select, MatchSpec}. As for select the table is traversed by calling ets:select/3 and ets:select/1. The difference is that the match_spec is explicitly given. This is how to state match_specs that cannot easily be expressed within the syntax provided by QLC.

The following example uses an explicit match_spec to traverse the table:

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

An example with implicit match_spec:

10> QH2 = qlc:q([[Y] || {X,Y} <- ets:table(Tab), (X > 1) or (X < 5)]).

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

11> qlc:info(QH1) =:= qlc:info(QH2).
true

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

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

Types:
• Tuple = tuple()
• MatchSpec = match_spec()
• Result = term()
• Errors = [{warning|error, string()}]

This function is a utility to test a match_spec [page 167] used in calls to ets:select/2. The function both tests MatchSpec for "syntactic" correctness and runs the match_spec against the object Tuple. If the match_spec contains errors, the tuple {error, Errors} is returned where Errors is a list of natural language descriptions of what was wrong with the match_spec. If the match_spec is syntactically OK, the function returns {ok,Term} where Term is what would have been the result in a real ets:select/2 call or false if the match_spec does not match the object Tuple.

This is a useful debugging and test tool, especially when writing complicated ets:select/2 calls.

to_dets(Tab, DetsTab) -> Tab

Types:
• Tab = tid() | atom()
• DetsTab = atom()}
Fills an already created/opened Dets table with the objects in the already opened ETS table named Tab. The Dets table is emptied before the objects are inserted.

```
update_counter(Tab, Key, UpdateOp) -> Result
update_counter(Tab, Key, [UpdateOp]) -> [Result]
update_counter(Tab, Key, Incr) -> Result
```

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

This function provides an efficient way to update one or more counters, without the hassle of having to look up an object, update the object by incrementing an element and insert the resulting object into the table again. (The update is done atomically; i.e. no process can access the ets table in the middle of the operation.)

It will destructively update the object with key Key in the table Tab by adding Incr to the element at the Pos:th position. The new counter value is returned. If no position is specified, the element directly following the key (\(keypos+1\)) is updated.

If a Threshold is specified, the counter will be reset to the value SetValue if the following conditions occur:

- The Incr is not negative (\(>= 0\)) and the result would be greater than (\(>\)) Threshold
- The Incr is negative (\(< 0\)) and the result would be less than (\(<\)) Threshold

A list of UpdateOp can be supplied to do several update operations within the object. The operations are carried out in the order specified in the list. If the same counter position occurs more than one time in the list, the corresponding counter will thus be updated several times, each time based on the previous result. The return value is a list of the new counter values from each update operation in the same order as in the operation list. If an empty list is specified, nothing is updated and an empty list is returned. If the function should fail, no updates will be done at all.

The given Key is used to identify the object by either matching the key of an object in a set table, or compare equal to the key of an object in an ordered set table (see lookup/2 [page 174] and new/2 [page 178] for details on the difference).

The function will fail with reason badarg if:

- the table is not of type set or ordered_set,
- no object with the right key exists,
- the object has the wrong arity,
- the element to update is not an integer,
- the element to update is also the key, or,
- any of Pos, Incr, Threshold or SetValue is not an integer

```
update_element(Tab, Key, [Pos,Value]) -> true | false
update_element(Tab, Key, [[Pos,Value]]) -> true | false
```

Types:
This function provides an efficient way to update one or more elements within an object, without the hassle of having to look up, update and write back the entire object. It will destructively update the object with key \texttt{Key} in the table \texttt{Tab}. The element at the \texttt{Pos}:th position will be given the value \texttt{Value}.

A list of \{\texttt{Pos}, \texttt{Value}\} can be supplied to update several elements within the same object. If the same position occurs more than one in the list, the last value in the list will be written. If the list is empty or the function fails, no updates will be done at all. The function is also atomic in the sense that other processes can never see any intermediate results.

The function returns \texttt{true} if an object with the key \texttt{Key} was found, \texttt{false} otherwise.

The given \texttt{Key} is used to identify the object by either matching the key of an object in a set table, or compare equal to the key of an object in an ordered set table (see \texttt{lookup/2} [page 174] and \texttt{new/2} [page 178] for details on the difference).

The function will fail with reason \texttt{badarg} if:

- the table is not of type \texttt{set} or \texttt{ordered_set},
- \texttt{Pos} is less than 1 or greater than the object arity, or,
- the element to update is also the key
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 \( \text{Size} \) of the binary is calculated as \( \langle \langle \text{Size}:\text{HeaderLength}/\text{unit}:8 \rangle \rangle = B \).

- `{format, Format}`. The format determines the function that is applied to binaries in order to create the terms that will be sorted. The default value is `binary_term`, which is equivalent to `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`. An ordering function is antisymmetric, transitive and total. `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:get_cwd() is used instead. The names of temporary files are derived from the Erlang nodename (node()), the process identifier of the current Erlang emulator (os:getpid()), and a timestamp (erlang:now()); a typical name would be fs_mynode@myhost_1763_1043_337000_266005.17, where 17 is a sequence number. Existing files will be overwritten. Temporary files are deleted unless some uncaught EXIT signal occurs.

Temporary files and the output file may be compressed. The default value false implies that written files are not compressed. Regardless of the value of the compressed option, compressed files can always be read. Note that reading and writing compressed files is significantly slower than reading and writing uncompressed files.

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

By default 16 files are merged at a time. This option should rarely be needed.

To summarize, here is the syntax of the options:

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 merge function. Each output function is called exactly once. When some output function has been applied to all of the results or an error occurs, the last function is called with the argument close, 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 \texttt{(end_of_input, Value)}, the function given as output will be called with the argument \texttt{(value, Value)}. This makes it easy to initiate the sequence of output functions with a value calculated by the input functions.

As an example, consider sorting the terms on a disk log file. A function that reads chunks from the disk log and returns a list of binaries is used as input. The results are collected in a list of terms.

\begin{verbatim}
sort(Log) ->
  \{ok, \_\} = disk_log:open([\{name,Log\}, \{mode,read\,only\}]),
  Input = input(Log, start),
  Output = output([],
  Reply = file_sorter:sort(Input, Output, \{format,term\}),
  ok = disk_log:close(Log),
  Reply.

input(Log, Cont) ->
  fun(close) ->
    ok;
  (read) ->
    case disk_log:chunk(Log, Cont) of
      \{error, Reason\} ->
        \{error, Reason\};
      \{Cont2, Terms\} ->
        \{Terms, input(Log, Cont2)\};
      \{Cont2, Terms, \_Badbytes\} ->
        \{Terms, input(Log, Cont2)\};
      eof ->
        end_of_input
    end.

output(L) ->
  fun(close) ->
    lists:append(lists:reverse(L));
  (Terms) ->
    output([Terms | L])
  end.
\end{verbatim}

Further examples of functions as input and output can be found at the end of the file_sorter module; the term format is implemented with functions.

The possible values of \texttt{Reason} returned when an error occurs are:

- \texttt{bad\_object,\{bad\_object, FileName\}}. Applying the format function failed for some binary, or the key(s) could not be extracted from some term.
- \texttt{\{bad\_term, FileName\}}. \texttt{io:read/2} failed to read some term.
- \texttt{\{file\_error, FileName, Reason2\}}. See file(3) for an explanation of \texttt{Reason2}.
- \texttt{\{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.

**DATA TYPES**

```
filename() = string() | atom() | DeepList
dirname() = filename()
DeepList = [char() | atom() | DeepList]
```

**Exports**

- `ensure_dir(Name) -> ok | {error, Reason}`
  - **Types:**
    - Name = filename() | dirname()
    - Reason = posix() – see file(3)
  - The `ensure_dir/1` function ensures that all parent directories for the given file or directory name `Name` exist, trying to create them if necessary. Returns ok if all parent directories already exist or could be created, or `error`, `Reason` if some parent directory does not exist and could not be created for some reason.

- `file_size(Filename) -> integer()`
  - The `file_size` function returns the size of the given file.

- `fold_files(Dir, RegExp, Recursive, Fun, AccIn) -> AccOut`
  - **Types:**
    - Dir = dirname()
    - RegExp = regular_expression_string()
    - 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` (see the `re` [page 345] module for a description of the allowed regular expressions). If `Recursive` is true all sub-directories to `Dir` are processed. The regular expression matching is done on just the filename without the directory part.

```prolog
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.
```

```prolog
is_file(Name) -> true | false
Types:
- Name = filename() | dirname()

The `is_file/1` function returns `true` if `Name` refers to a file or a directory, and `false` otherwise.
```

```prolog
is_regular(Name) -> true | false
Types:
- Name = filename()

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

```prolog
last_modified(Name) -> {{Year,Month,Day},{Hour,Min,Sec}} | 0
Types:
- Name = filename() | dirname()

The `last_modified/1` function returns the date and time the given file or directory was last modified, or 0 if the file does not exist.
```

```prolog
wildcard(Wildcard) -> list()
Types:
- Wildcard = filename() | dirname()

The `wildcard/1` function returns a list of all files that match Unix-style wildcard-string `Wildcard`

The wildcard string looks like an ordinary filename, except that certain “wildcard characters” are interpreted in a special way. The following characters are special:

- `?` Matches one character.
- `*` Matches any number of characters up to the end of the filename, the next dot, or the next slash.
- `{Item,...}` Alternation. Matches one of the alternatives.
Other characters represent themselves. Only filenames that have exactly the same character in the same position will match. (Matching is case-sensitive; i.e. “a” will not match “A”).

Note that multiple “*” characters are allowed (as in Unix wildcards, but opposed to Windows/DOS wildcards).

Examples:
The following examples assume that the current directory is the top of an Erlang/OTP installation.

To find all .beam files in all applications, the following line can be used:

    filelib:wildcard("lib/*/ebin/*.beam").

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

    filelib:wildcard("lib/*/src/*.?rl")

or the following line

    filelib:wildcard("lib/*/src/*.{erl,hrl}"

can be used.

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

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

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

    filelib:wildcard("lib/*/{src,include}/*.{erl,hrl}"

wildcard(Wildcard, Cwd) -> list()

Types:
• Wildcard = filename() | dirname()
• Cwd = dirname()

The wildcard/2 function works like wildcard/1, except that instead of the actual working directory, Cwd will be used.
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**

\[
\text{name()} = \text{string()} | \text{atom()} | \text{DeepList} \\
\text{DeepList} = [\text{char()} | \text{atom()} | \text{DeepList}]
\]

**Exports**

\[
\text{absname(Filename)} -> \text{string()}
\]

- **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()`
     
  2. `filename:absname("foo").` 
     
  3. `filename:absname("../x").` 
     
  4. `filename:absname("/").` 

  **Windows examples:**
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.joind(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.

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.
dirname(Filename) -> string()
Types:
  • Filename = name()
Returns the directory part of Filename.

13> filename:dirname("/usr/src/kalle.erl").
"/usr/src"
14> filename:dirname("kalle.erl").
""%

5> filename:dirname("\usr\src\kalle.erl"). % Windows
"/usr/src"

extension(Filename) -> string()
Types:
  • Filename = name()
Returns the file extension of Filename, including the period. Returns an empty string if there is no extension.

15> filename:extension("foo.erl").
".erl"
16> filename:extension("beam.src/kalle").
[]

flatten(Filename) -> string()
Types:
  • Filename = name()
Converts a possibly deep list filename consisting of characters and atoms into the corresponding flat string filename.

join(Components) -> string()
Types:
  • Components = [string()]
Joins a list of file name Components with directory separators. If one of the elements of Components includes an absolute path, for example "/xxx", the preceding elements, if any, are removed from the result.
The result is “normalized”: 
Redundant directory separators are removed.
In Windows, all directory separators are forward slashes and the drive letter is in lower case.

```
17> filename:join(["/usr", "local", "bin"]).
="/usr/local/bin"
18> filename:join(["a/b///c/"]).
="a/b/c"
6> filename:join(["B:a\b///c/"]). % Windows
="b:a/b/c"
```

`join(Name1, Name2) -> string()`

Types:
- `Name1 = Name2 = string()`
  Joins two file name components with directory separators. Equivalent to `join([Name1, Name2]).`

`nativename(Path) -> string()`

Types:
- `Path = string()`
  Converts `Path` to a form accepted by the command shell and native applications on the current platform. On Windows, forward slashes is converted to backward slashes. On all platforms, the name is normalized as done by `join/1`.

```
19> filename:nativename("/usr/local/bin/"). % Unix
="/usr/local/bin"
7> filename:nativename("/usr/local/bin/"). % Windows
="\usr\local\bin"
```

`pathtype(Path) -> absolute | relative | volumerelative`

Returns the type of path, one of absolute, relative, or volumerelative.

- **absolute** The path name refers to a specific file on a specific volume.
  Unix example: `/usr/local/bin`
  Windows example: `D:/usr/local/bin`

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

- **volumerelative** The path name is relative to the current working directory on a specified volume, or it is a specific file on the current working volume.
  Windows example: `D:bar.erl, /bar/foo.erl`

`rootname(Filename) -> string()`

`rootname(Filename, Ext) -> string()`

Types:
filename = Ext = name()

Remove a filename extension. rootname/2 works as rootname/1, except that the extension is removed only if it is Ext.

20> filename:rootname("/beam.src/kalle").
"/beam.src/kalle"
21> filename:rootname("/beam.src/foo.erl").
"/beam.src/foo"
22> filename:rootname("/beam.src/foo.erl", ".erl").
"/beam.src/foo"
23> filename:rootname("/beam.src/foo.beam", ".erl").
"/beam.src/foo.beam"

split(Filename) -> Components

Types:
- Filename = name()
- Components = [string()]

Returns a list whose elements are the path components of Filename.

24> filename:split("/usr/local/bin").
["/", "usr", "local", "bin"]
25> filename:split("foo/bar").
["foo", "bar"]
26> filename:split("a:\msdev\include").
["a:\", "msdev", "include"]

find_src(Beam) -> {SourceFile, Options} | error,[ErrorReason,Module])

find_src(Beam, Rules) -> {SourceFile, Options} | error,[ErrorReason,Module])

Types:
- Beam = Module | Filename
- Module = atom()
- Filename = string() | atom()
- SourceFile = string()
- Options = [Opt]
- Opt = [i, string()] | [outdir, string()] | [d, atom()]
- ErrorReason = non_existing | preloaded | interpreted

Finds the source filename and compiler options for a module. The result can be fed to compile:file/2 in order to compile the file again.

The Beam argument, which can be a string or an atom, specifies either the module name or the path to the source code, with or without the ".erl" extension. In either case, the module must be known by the code server, i.e. code:which(Module) must succeed.

Rules describes how the source directory can be found, when the object code directory is known. It is a list of tuples [BinSuffix, SourceSuffix] and is interpreted as follows: If the end of the directory name where the object is located matches BinSuffix, then the source code directory has the same name, but with BinSuffix replaced by SourceSuffix. Rules defaults to:

[{"", ""}, {"ebin", "src"}, {"ebin", "esrc"}]
If the source file is found in the resulting directory, then the function returns that location together with Options. Otherwise, the next rule is tried, and so on.

The function returns \{SourceFile, Options\} if it succeeds. SourceFile is the absolute path to the source file without the ".erl" extension. Options include the options which are necessary to recompile the file with compile:file/2, but excludes options such as report or verbose which do not change the way code is generated. The paths in the \{outdir, Path\} and \{i, Path\} options are guaranteed to be absolute.
gb_sets

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 $O(|T| \cdot \log(|S|))$, where $S$ is the largest given set, depending on which is fastest for any particular function call. For operating on sets of almost equal size, this implementation is about 3 times slower than using ordered-list sets directly. For sets of very different sizes, however, this solution can be arbitrarily much faster; in practical cases, often between 10 and 100 times. This implementation is particularly suited for accumulating elements a few at a time, building up a large set (more than 100-200 elements), and repeatedly testing for membership in the current set.

As with normal tree structures, lookup (membership testing), insertion and deletion have logarithmic complexity.

Compatibility

All of the following functions in this module also exist and do the same thing in the sets and ordsets modules. That is, by only changing the module name for each call, you can try out different set representations.

- add_element/2
- del_element/2
- filter/2
- fold/3
- from_list/1
- intersection/1
- intersection/2
- is_element/2
- is_set/1
- is_subset/2
- new/0
- size/1
- subtract/2
• to_list/1
• union/1
• union/2

DATA TYPES

gb_set() = a GB set

Exports

add(Element, Set1) -> Set2
add_element(Element, Set1) -> Set2

Types:
• Element = term()
• Set1 = Set2 = gb_set()

Returns a new gb_set formed from Set1 with Element inserted. If Element is already an element in Set1, nothing is changed.

balance(Set1) -> Set2

Types:
• Set1 = Set2 = gb_set()

Rebalances the tree representation of Set1. Note that this is rarely necessary, but may be motivated when a large number of elements have been deleted from the tree without further insertions. Rebalancing could then be forced in order to minimise lookup times, since deletion only does not rebalance the tree.

delete(Element, Set1) -> Set2

types:
• Element = term()
• Set1 = Set2 = gb_set()

Returns a new gb_set formed from Set1 with Element removed. Assumes that Element is present in Set1.

delete_any(Element, Set1) -> Set2
del_element(Element, Set1) -> Set2

Types:
• Element = term()
• Set1 = Set2 = gb_set()

Returns a new gb_set formed from Set1 with Element removed. If Element is not an element in Set1, nothing is changed.

difference(Set1, Set2) -> Set3
subtract(Set1, Set2) -> Set3
Types:
  - Set1 = Set2 = Set3 = gb_set()
Returns only the elements of Set1 which are not also elements of Set2.

empty() -> Set
new() -> Set
Types:
  - Set = gb_set()
Returns a new empty gb_set.

filter(Pred, Set1) -> Set2
Types:
  - Pred = fun (E) -> bool()
  - E = term()
  - Set1 = Set2 = gb_set()
Filters elements in Set1 using predicate function Pred.

fold(Function, Acc0, Set) -> Acc1
Types:
  - Function = fun (E, AccIn) -> AccOut
  - Acc0 = Acc1 = AccIn = AccOut = term()
  - E = term()
  - Set = gb_set()
Folds Function over every element in Set returning the final value of the accumulator.

from_list(List) -> Set
Types:
  - List = [term()]
  - Set = gb_set()
Returns a gb_set of the elements in List, where List may be unordered and contain duplicates.

from_ordset(List) -> Set
Types:
  - List = [term()]
  - Set = gb_set()
Turns an ordered-set list List into a gb_set. The list must not contain duplicates.

insert(Element, Set1) -> Set2
Types:
  - Element = term()
- Set1 = Set2 = gb_set()

  Returns a new gb_set formed from Set1 with Element inserted. Assumes that Element is not present in Set1.

intersection(Set1, Set2) -> Set3

Types:
- Set1 = Set2 = Set3 = gb_set()

  Returns the intersection of Set1 and Set2.

intersection(SetList) -> Set

Types:
- SetList = [gb_set()]
- Set = gb_set()

  Returns the intersection of the non-empty list of gb_sets.

is_disjoint(Set1, Set2) -> bool()

Types:
- Set1 = Set2 = gb_set()

  Returns true if Set1 and Set2 are disjoint (have no elements in common), and false otherwise.

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(Term) -> bool()

Types:
- Term = term()

  Returns true if Set appears to be a gb_set, otherwise false.

is_subset(Set1, Set2) -> bool()

Types:
- Set1 = Set2 = gb_set()
Returns true when every element of Set1 is also a member of Set2, otherwise false.

`iterator(Set) -> Iter`

**Types:**
- `Set = gb_set()`
- `Iter = term()`

Returns an iterator that can be used for traversing the entries of Set; see `next/1`. The implementation of this is very efficient; traversing the whole set using `next/1` is only slightly slower than getting the list of all elements using `to_list/1` and traversing that. The main advantage of the iterator approach is that it does not require the complete list of all elements to be built in memory at one time.

`largest(Set) -> term()`

**Types:**
- `Set = gb_set()`

Returns the largest element in Set. Assumes that Set is nonempty.

`next(Iter1) -> {Element, Iter2} | none`

**Types:**
- `Iter1 = Iter2 = Element = term()`

Returns `{Element, Iter2}` where `Element` is the smallest element referred to by the iterator `Iter1`, and `Iter2` is the new iterator to be used for traversing the remaining elements, or the atom `none` if no elements remain.

`singleton(Element) -> gb_set()`

**Types:**
- `Element = term()`

Returns a gb_set containing only the element `Element`.

`size(Set) -> int()`

**Types:**
- `Set = gb_set()`

Returns the number of elements in Set.

`smallest(Set) -> term()`

**Types:**
- `Set = gb_set()`

Returns the smallest element in Set. Assumes that Set is nonempty.

`take_largest(Set1) -> {Element, Set2}`

**Types:**
- `Set1 = Set2 = gb_set()`
- `Element = term()`
Returns \( \{\text{Element, Set2}\} \), where Element is the largest element in Set1, and Set2 is this set with Element deleted. Assumes that Set1 is nonempty.

\[
\text{take\_smallest(Set1) -> \{Element, Set2\}}
\]

**Types:**
- \( \text{Set1 = Set2 = gb\_set()} \)
- \( \text{Element = term()} \)

Returns \( \{\text{Element, Set2}\} \), where Element is the smallest element in Set1, and Set2 is this set with Element deleted. Assumes that Set1 is nonempty.

\[
\text{to\_list(Set) -> List}
\]

**Types:**
- \( \text{Set = gb\_set()} \)
- \( \text{List = [term()]} \)

Returns the elements of Set as a list.

\[
\text{union(Set1, Set2) -> Set3}
\]

**Types:**
- \( \text{Set1 = Set2 = Set3 = gb\_set()} \)

Returns the merged (union) gb\_set of Set1 and Set2.

\[
\text{union(SetList) -> Set}
\]

**Types:**
- \( \text{SetList = [gb\_set()]} \)
- \( \text{Set = gb\_set()} \)

Returns the merged (union) gb\_set of the list of gb\_sets.

**SEE ALSO**

gb\_trees(3) [page 210], ordsets(3) [page 301], sets(3) [page 397]
gb_trees
Erlang Module

An efficient implementation of Prof. Arne Andersson's General Balanced Trees. These have no storage overhead compared to unbalanced binary trees, and their performance is in general better than AVL trees.

Data structure

Data structure:
- \{Size, Tree\}, where 'Tree' is composed of nodes of the form:
  - \{Key, Value, Smaller, Bigger\}, and the "empty tree" node:
    - nil.

There is no attempt to balance trees after deletions. Since deletions do not increase the height of a tree, this should be O(K).

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

Performance is comparable to the AVL trees in the Erlang book (and faster in general due to less overhead); the difference is that deletion works for these trees, but not for the book's trees. Behaviour is logarithmic (as it should be).

DATA TYPES

gb_tree() = a GB tree

Exports

balance(Tree1) -> Tree2

Types:
- Tree1 = Tree2 = gb_tree()

Rebalances Tree1. Note that this is rarely necessary, but may be motivated when a large number of nodes have been deleted from the tree without further insertions.

Rebalancing could then be forced in order to minimise lookup times, since deletion only does not rebalance the tree.

delete(Key, Tree1) -> Tree2

Types:
- Key = term()
- Tree1 = Tree2 = gb_tree()

Removes the node with key Key from Tree1; returns new tree. Assumes that the key is present in the tree, crashes otherwise.

delete_any(Key, Tree1) -> Tree2

Types:
- Key = term()
- Tree1 = Tree2 = gb_tree()

Removes the node with key Key from Tree1 if the key is present in the tree, otherwise does nothing; returns new tree.

empty() -> Tree

Types:
- Tree = gb_tree()

Returns a new empty tree.

enter(Key, Val, Tree1) -> Tree2

Types:
- Key = Val = term()
- Tree1 = Tree2 = gb_tree()

Inserts Key with value Val into Tree1 if the key is not present in the tree, otherwise updates Key to value Val in Tree1. Returns the new tree.

from_orddict(List) -> Tree

Types:
- List = [{Key, Val}]
- Key = Val = term()
- Tree = gb_tree()

Turns an ordered list List of key-value tuples into a tree. The list must not contain duplicate keys.

get(Key, Tree) -> Val

Types:
- Key = Val = term()
- Tree = gb_tree()

Retrieves the value stored with Key in Tree. Assumes that the key is present in the tree, crashes otherwise.

lookup(Key, Tree) -> {value, Val} | none

Types:
- Key = Val = term()
- Tree = gb_tree()
Looks up Key in Tree; returns \{value, Val\}, or none if Key is not present.

```
insert(Key, Val, Tree1) -> Tree2
```

Types:
- Key = Val = term()
- Tree1 = Tree2 = gb_tree()

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

```
is_defined(Key, Tree) -> bool()
```

Types:
- Tree = gb_tree()

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

```
is_empty(Tree) -> bool()
```

Types:
- Tree = gb_tree()

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

```
iterator(Tree) -> Iter
```

Types:
- Tree = gb_tree()
- Iter = term()

Returns an iterator that can be used for traversing the entries of Tree; see `next/1`. The implementation of this is very efficient; traversing the whole tree using `next/1` is only slightly slower than getting the list of all elements using `to_list/1` and traversing that. The main advantage of the iterator approach is that it does not require the complete list of all elements to be built in memory at one time.

```
keys(Tree) -> [Key]
```

Types:
- Tree = gb_tree()
- Key = term()

Returns the keys in Tree as an ordered list.

```
largest(Tree) -> {Key, Val}
```

Types:
- Tree = gb_tree()
- Key = Val = term()

Returns \{Key, Val\}, where Key is the largest key in Tree, and Val is the value associated with this key. Assumes that the tree is nonempty.

```
map(Function, Tree1) -> Tree2
```

Types:
- Function = fun(K, V1) -> V2
- Tree1 = Tree2 = gb_tree()

maps the function F(K, V1) -> V2 to all key-value pairs of the tree Tree1 and returns a new tree Tree2 with the same set of keys as Tree1 and the new set of values V2.

next(Iter1) -> {Key, Val, Iter2} | none

Types:
- Iter1 = Iter2 = Key = Val = term()

Returns {Key, Val, Iter2} where Key is the smallest key referred to by the iterator Iter1, and Iter2 is the new iterator to be used for traversing the remaining nodes, or the atom none if no nodes remain.

size(Tree) -> int()

Types:
- Tree = gb_tree()

Returns the number of nodes in Tree.

smallest(Tree) -> {Key, Val}

Types:
- Tree = gb_tree()
- Key = Val = term()

Returns {Key, Val}, where Key is the smallest key in Tree, and Val is the value associated with this key. Assumes that the tree is nonempty.

take_largest(Tree1) -> {Key, Val, Tree2}

Types:
- Tree1 = Tree2 = gb_tree()
- Key = Val = term()

Returns {Key, Val, Tree2}, where Key is the largest key in Tree1, Val is the value associated with this key, and Tree2 is this tree with the corresponding node deleted. Assumes that the tree is nonempty.

take_smallest(Tree1) -> {Key, Val, Tree2}

Types:
- Tree1 = Tree2 = gb_tree()
- Key = Val = term()

Returns {Key, Val, Tree2}, where Key is the smallest key in Tree1, Val is the value associated with this key, and Tree2 is this tree with the corresponding node deleted. Assumes that the tree is nonempty.

to_list(Tree) -> [{Key, Val}]

Types:
- Tree = gb_tree()
gb_trees

- 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 204], dict(3) [page 120]
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>----- &gt; -</td>
</tr>
<tr>
<td>gen_event:add_handler</td>
<td></td>
</tr>
<tr>
<td>gen_event:add_sup_handler</td>
<td>----- &gt; Module:init/1</td>
</tr>
<tr>
<td>gen_event:notify</td>
<td></td>
</tr>
<tr>
<td>gen_event:sync_notify</td>
<td>----- &gt; Module:handle_event/2</td>
</tr>
<tr>
<td>gen_event:call</td>
<td>----- &gt; Module:handle_call/2</td>
</tr>
<tr>
<td>-</td>
<td>----- &gt; Module:handle_info/2</td>
</tr>
<tr>
<td>gen_event:delete_handler</td>
<td>----- &gt; Module:terminate/2</td>
</tr>
<tr>
<td>gen_event:swap_handler</td>
<td></td>
</tr>
<tr>
<td>gen_event:swap_sup_handler</td>
<td>----- &gt; Module1:terminate/2</td>
</tr>
<tr>
<td></td>
<td>Module2:init/1</td>
</tr>
<tr>
<td>gen_event:which_handlers</td>
<td>----- &gt; -</td>
</tr>
<tr>
<td>gen_event:stop</td>
<td>----- &gt; Module:terminate/2</td>
</tr>
<tr>
<td>-</td>
<td>----- &gt; Module:code_change/3</td>
</tr>
</tbody>
</table>
```

Since each event handler is one callback module, an event manager will have several callback modules which are added and deleted dynamically. Therefore gen_event is more tolerant of callback module errors than the other behaviours. If a callback function for an installed event handler fails with Reason, or returns a bad value Term, the event manager will not fail. It will delete the event handler by calling the callback function Module:terminate/2 (see below), giving as argument \{error,\{'EXIT',Reason\}\} or \{error,Term\}, respectively. No other event handler will be affected.
The sys module can be used for debugging an event manager. Note that an event manager does trap exit signals automatically.

The gen_event process can go into hibernation (see [erlang(3)]) if a callback function in a handler module specifies 'hibernate' in its return value. This might be useful if the server is expected to be idle for a long time. However this feature should be used with care as hibernation implies at least two garbage collections (when hibernating and shortly after waking up) and is not something you'd want to do between each event handled by a busy event manager.

It's also worth noting that when multiple event handlers are invoked, it's sufficient that one single event handler returns a 'hibernate' request for the whole event manager to go into hibernation.

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

Exports

start_link() -> Result
start_link(EventMgrName) -> Result

Types:
- EventMgrName = {local, Name} | {global, Name}
- Name = atom()
- Result = {ok, Pid} | {error, {already_started, Pid}}
- Pid = pid()

Creates an event manager process as part of a supervision tree. The function should be called, directly or indirectly, by the supervisor. It will, among other things, ensure that the event manager is linked to the supervisor.

If EventMgrName=local, Name, the event manager is registered locally as Name using register/2. If EventMgrName=global, Name, the event manager is registered globally as Name using global:register_name/2. If no name is provided, the event manager is not registered.

If the event manager is successfully created the function returns {ok, Pid}, where Pid is the pid of the event manager. If there already exists a process with the specified EventMgrName the function returns {error, {already_started, Pid}}, where Pid is the pid of that process.

start() -> Result
start(EventMgrName) -> Result

Types:
- EventMgrName = {local, Name} | {global, Name}
- Name = atom()
- Result = {ok, Pid} | {error, {already_started, Pid}}
- Pid = pid()}
**gen_event**

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.

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

**Types:**

- `EventMgr = Name |{Name,Node} | {global,Name} | pid()`
- `Name = Node = atom()`
- `Handler = Module | {Module,Id}`
- `Module = atom()`
- `Id = term()`
- `Args = term()`
- `Result = ok | {'EXIT',Reason} | term()`
- `Reason = term()`

Adds a new event handler to the event manager `EventMgrRef`. The event manager will call `Module:init/1` to initiate the event handler and its internal state. `EventMgrRef` can be:

- the pid,
- `Name`, if the event manager is locally registered,
- `{Name,Node}`, if the event manager is locally registered at another node, or
- `{global,Name}`, if the event manager is globally registered.

`Handler` is the name of the callback module `Module` or a tuple `{Module,Id}`, where `Id` is any term. The `{Module,Id}` representation makes it possible to identify a specific event handler when there are several event handlers using the same callback module. `Args` is an arbitrary term which is passed as the argument to `Module:init/1`. If `Module:init/1` returns a correct value, the event manager adds the event handler and this function returns `ok`. If `Module:init/1` fails with `Reason` or returns an unexpected value `Term`, the event handler is ignored and this function returns `{'EXIT',Reason}` or `Term`, respectively.

```erlang
add_sup_handler(EventMgrRef, Handler, Args) -> Result
```

**Types:**

- `EventMgr = Name |{Name,Node} | {global,Name} | pid()`
- `Name = Node = atom()`
- `Handler = Module | {Module,Id}`
- `Module = atom()`
- `Id = term()`
- `Args = term()`
- `Result = ok | {'EXIT',Reason} | term()`
- `Reason = term()`

Adds a new event handler in the same way as `add_handler/3` but will also supervise the connection between the event handler and the calling process.
If the calling process later terminates with Reason, the event manager will delete the event handler by calling Module:terminate/2 with \{stop,Reason\} as argument.

If the event handler later is deleted, the event manager sends a message \{gen_event,EXIT,Handler,Reason\} to the calling process. Reason is one of the following:

- normal, if the event handler has been removed due to a call to delete\_handler/3, or remove\_handler has been returned by a callback function (see below).
- shutdown, if the event handler has been removed because the event manager is terminating.
- \{swapped,NewHandler,Pid\}, if the process Pid has replaced the event handler with another event handler NewHandler using a call to swap\_handler/3 or swap\_sup\_handler/3.
- a term, if the event handler is removed due to an error. Which term depends on the error.

See add\_handler/3 for a description of the arguments and return values.

**notify**(EventMgrRef, Event) \rightarrow ok
**sync\_notify**(EventMgrRef, Event) \rightarrow 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) \rightarrow Result

call(EventMgrRef, Handler, Request, Timeout) \rightarrow Result

Types:
- EventMgrRef = Name | \{Name,Node\} | \{global,Name\} | pid()
  
- Name = Node = atom()
  
- Handler = Module | \{Module,Id\}
  
- Module = atom()
  
- Id = term()
  
- Request = term()
  
- Timeout = int()\(\geq 0\) | infinity
make a synchronous call to the event handler Handler installed in the event manager EventMgrRef by sending a request and waiting until a reply arrives or a timeout occurs. The event manager will call Module:handle_call/2 to handle the request.

See add_handler/3 for a description of EventMgrRef and Handler.

Request is an arbitrary term which is passed as one of the arguments to Module:handle_call/2.

Timeout is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom infinity to wait indefinitely. Default value is 5000. If no reply is received within the specified time, the function call fails.

The return value Reply is defined in the return value of Module:handle_call/2. If the specified event handler is not installed, the function returns [error,bad module]. If the callback function fails with Reason or returns an unexpected value Term, this function returns [error,{'EXIT',Reason}] or [error,Term], respectively.

\[
delete_handler(EventMgrRef, Handler, Args) \rightarrow Result
\]

Types:
- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler = Module | {Module,Id}
- Module = atom()
- Id = term()
- Args = term()
- Result = term() | {error,module_not_found} | {'EXIT',Reason}
- Reason = term()

Deletes an event handler from the event manager EventMgrRef. The event manager will call Module:terminate/2 to terminate the event handler.

See add_handler/3 for a description of EventMgrRef and Handler.

Args is an arbitrary term which is passed as one of the arguments to Module:terminate/2.

The return value is the return value of Module:terminate/2. If the specified event handler is not installed, the function returns [error,module_not_found]. If the callback function fails with Reason, the function returns {'EXIT',Reason}.

\[
swap_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) \rightarrow Result
\]

Types:
- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler1 = Handler2 = Module | {Module,Id}
- Module = atom()
- Id = term()
- Args1 = Args2 = term()
Replaces an old event handler with a new event handler in the event manager EventMgrRef.
See add_handler/3 for a description of the arguments.

First the old event handler Handler1 is deleted. The event manager calls Module1:terminate(Args1, ...), where Module1 is the callback module of Handler1, and collects the return value.

Then the new event handler Handler2 is added and initiated by calling Module2:init([Args2,Term]), where Module2 is the callback module of Handler2 and Term the return value of Module1:terminate/2. This makes it possible to transfer information from Handler1 to Handler2.

The new handler will be added even if the the specified old event handler is not installed in which case Term=error, or if Module1:terminate/2 fails with Reason in which case Term=[ 'EXIT',Reason]. The old handler will be deleted even if Module2:init/1 fails.

If there was a supervised connection between Handler1 and a process Pid, there will be a supervised connection between Handler2 and Pid instead.

If Module2:init/1 returns a correct value, this function returns ok. If Module2:init/1 fails with Reason or returns an unexpected value Term, this this function returns {error,[ 'EXIT',Reason]} or {error,Term}, respectively.

swap_sup_handler(EventMgrRef, [Handler1,Args1], [Handler2,Args2]) -> Result

Types:
- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler1 = Handler 2 = Module | {Module,Id}
- Module = atom()
- Id = term()
- Args1 = Args2 = term()
- Result = ok | {error,Error}
- Error = {'EXIT',Reason} | term()
- Reason = term()

Replaces an event handler in the event manager EventMgrRef in the same way as swap_handler/3 but will also supervise the connection between Handler2 and the calling process.
See swap_handler/3 for a description of the arguments and return values.

which_handlers(EventMgrRef) -> [Handler]

Types:
- EventMgrRef = Name | {Name,Node} | {global,Name} | pid()
- Name = Node = atom()
- Handler = Module | {Module,Id}
- Module = atom()
- Id = term()}
Returns a list of all event handlers installed in the event manager `EventMgrRef`. See `add_handler/3` for a description of `EventMgrRef` and `Handler`.

```erlang
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

```erlang
Module:init(InitArgs) -> {ok,State} | {ok,State,hibernate}
```

Types:
- `InitArgs = Args | {Args,Term}`
- `Args = Term = term()`
- `State = term()`

Whenever a new event handler is added to an event manager, this function is called to initialize the event handler. If the event handler is added due to a call to `gen_event:add_handler/3` or `gen_event:add_sup_handler/3`, `InitArgs` is the `Args` argument of these functions. If the event handler is replacing another event handler due to a call to `gen_event:swap_handler/3` or `gen_event:swap_sup_handler/3`, or due to a swap return tuple from one of the other callback functions, `InitArgs` is a tuple `{Args,Term}` where `Args` is the argument provided in the function call/return tuple and `Term` is the result of terminating the old event handler, see `gen_event:swap_handler/3`. The function should return `{ok,State}` or `{ok,State,hibernate}` where `State` is the initial internal state of the event handler. If `{ok,State,hibernate}` is returned, the event manager will go into hibernation (by calling `proc_lib:hibernate/3` [page 312]), waiting for the next event to occur.

```erlang
Module:handle_event(Event, State) -> Result
```

Types:
- `Event = term()`
- `State = term()`
- `Result = {ok,NewState} | {ok,NewState,hibernate}`
- `{swap_handler,Args1,NewState,Handler2,Args2} | remove_handler`
- `NewState = term()`
Whenever an event manager receives an event sent using `gen_event:notify/2` or `gen_event:sync_notify/2`, this function is called for each installed event handler to handle the event.

**Event** is the `Event` argument of `notify/sync_notify`.

**State** is the internal state of the event handler.

If the function returns `{ok, NewState}` or `{ok, NewState, hibernate}` the event handler will remain in the event manager with the possible updated internal state `NewState`.

If `{ok, NewState, hibernate}` is returned, the event manager will also go into hibernation (by calling `proc_lib:hibernate/3` [page 312]), waiting for the next event to occur. It is sufficient that one of the event handlers return `{ok, NewState, hibernate}` for the whole event manager process to hibernate.

If the function returns `{swap_handler, Args1, NewState, Handler2, Args2}` the event handler will be replaced by `Handler2` by first calling `Module:terminate(Args1, NewState)` and then `Module2:init(Args2, Term)` where `Term` is the return value of `Module:terminate/2`. See `gen_event:swap_handler/3` for more information.

If the function returns `remove_handler` the event handler will be deleted by calling `Module:terminate(remove_handler, State)`.

```
Module:handle_call(Request, State) -> Result
Types:
  - Request = term()
  - State = term()
  - Result = {ok, Reply, NewState}  |  {ok, Reply, NewState, hibernate}
  -  |  {swap_handler, Reply, Args1, NewState, Handler2, Args2}
  -  |  {remove_handler, Reply}
  - Reply = term()
  - NewState = term()
  - Args1 = Args2 = term()
  - Handler2 = Module2  |  {Module2, Id}
  - Module2 = atom()
  - Id = term()
```

Whenever an event manager receives a request sent using `gen_event:call/3,4`, this function is called for the specified event handler to handle the request.

**Request** is the `Request` argument of `call`.

**State** is the internal state of the event handler.

The return values are the same as for `handle_event/2` except they also contain a term `Reply` which is the reply given back to the client as the return value of `call`.

```
Module:handle_info(Info, State) -> Result
Types:
```
• Info = term()
• State = term()
• Result = \{ok,NewState\} \| \{ok,NewState,hibernate\}
• \| \{swap_handler,Args1,NewState,Handler2,Args2\} \| \{remove_handler
• NewState = term()
• Args1 = Args2 = term()
• Handler2 = Module2 \| \{Module2,Id\}
• Module2 = atom()
• Id = term()

This function is called for each installed event handler when an event manager receives
any other message than an event or a synchronous request (or a system message).
Info is the received message.
See Module:handle_event/2 for a description of State and possible return values.

Module:terminate(Arg, State) -> term()

Types:
• Arg = Args \| \{stop,Reason\} \| \{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 444], sys(3) [page 454]
Erlang Module

A behaviour module for implementing a finite state machine. A generic finite state machine process (gen_fsm) implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into an OTP supervision tree. Refer to [OTP Design Principles] for more information.

A gen_fsm assumes all specific parts to be located in a callback module exporting a pre-defined set of functions. The relationship between the behaviour functions and the callback functions can be illustrated as follows:

```
gen_fsm module                     Callback module
-----------------------------
gen_fsm:start_link                --> Module:init/1

gen_fsm:send_event               --> Module:StateName/2

gen_fsm:send_all_state_event     --> Module:handle_event/3

gen_fsm:sync_send_event          --> Module:StateName/3

gen_fsm:sync_send_all_state_event --> Module:handle_sync_event/4
-
                       --> Module:handle_info/3
-
                       --> Module:terminate/3
-
                       --> Module:code_change/4
```

If a callback function fails or returns a bad value, the gen_fsm will terminate.

The sys module can be used for debugging a gen_fsm.

Note that a gen_fsm does not trap exit signals automatically, this must be explicitly initiated in the callback module.

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

The gen_fsm process can go into hibernation (see [erlang(3)]) if a callback function specifies 'hibernate' instead of a timeout value. This might be useful if the server is expected to be idle for a long time. However this feature should be used with care as hibernation implies at least two garbage collections (when hibernating and shortly after waking up) and is not something you’d want to do between each call to a busy state machine.
Exports

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

Types:
- \(\text{FsmName} = \{\text{local,Name}\} \mid \{\text{global,GlobalName}\}\)
- \(\text{Name} = \text{atom}()\)
- \(\text{GlobalName} = \text{term}()\)
- \(\text{Module} = \text{atom}()\)
- \(\text{Args} = \text{term}()\)
- \(\text{Options} = [\text{Option}]\)
- \(\text{Option} = \{\text{debug,Dbgs}\} \mid \{\text{timeout,Time}\} \mid \{\text{spawn\_opt,SOpts}\}\)
- \(\text{Dbgs} = [\text{Dbg}]\)
- \(\text{Dbg} = \text{trace} \mid \text{log} \mid \text{statistics}\)
- \(\text{Time} = \text{term}()\)
- \(\text{SOpts} = [\text{SOpt}]\)
- \(\text{SOpt} = \text{term}()\)
- \(\text{Result} = \{\text{ok,Pid}\} \mid \text{ignore} \mid \{\text{error,Error}\}\)
- \(\text{Pid} = \text{pid}()\)
- \(\text{Error} = \{\text{already\_started,Pid}\} \mid \text{term}()\)

Creates a \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:}\text{init/1} to initialize. To ensure a synchronized start-up procedure, \text{start\_link/3,4} does not return until \text{Module:}\text{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:}\text{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 454].

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)}.

\textbf{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.

\begin{verbatim}
start(Module, Args, Options) -> Result
start(FsmName, Module, Args, Options) -> Result
\end{verbatim}

Types:
- \texttt{FsmName = \{local,Name\} | \{global,GlobalName\}}
- \texttt{Name = atom()}
- \texttt{GlobalName = term()}
- \texttt{Module = atom()}
- \texttt{Args = term()}
- \texttt{Options = \[Option\]}
- \texttt{Option = \{debug,Dbgs\} | \{timeout,Time\} | \{spawn_opt,SOpts\}}
- \texttt{Dbgs = \[Dbg\]}
- \texttt{Dbg = \{trace\} | \{log\} | \{statistics\}}
- \texttt{\{log_to_file,FileName\} | \{install,\{Func,FuncState\}\}}
- \texttt{SOpts = \[term()\]}
- \texttt{Result = \{ok,Pid\} | ignore | \{error,Error\}}
- \texttt{Pid = pid()}
- \texttt{Error = \{already_started,Pid\} | term()}

Creates a stand-alone \texttt{gen_fsm} process, i.e. a \texttt{gen_fsm} which is not part of a supervision tree and thus has no supervisor.

See \texttt{start_link/3,4} [page 226] for a description of arguments and return values.

\begin{verbatim}
send_event(FsmRef, Event) -> ok
\end{verbatim}

Types:
- \texttt{FsmRef = Name | \{Name,Node\} | \{global,GlobalName\} | pid()}
- \texttt{Name = N ode = atom()}
- \texttt{GlobalName = term()}
- \texttt{Event = term()}

Sends an event asynchronously to the \texttt{gen_fsm} \texttt{FsmRef} and returns \texttt{ok} immediately. The \texttt{gen_fsm} will call Module:StateName/2 to handle the event, where StateName is the name of the current state of the \texttt{gen_fsm}.

\texttt{FsmRef} can be:
- the pid,
- \texttt{Name}, if the \texttt{gen_fsm} is locally registered,
- \{\texttt{Name},Node\}, if the \texttt{gen_fsm} is locally registered at another node, or
- \{global,GlobalName\}, if the \texttt{gen_fsm} is globally registered.

Event is an arbitrary term which is passed as one of the arguments to Module:StateName/2.
send_all_state_event(FsmRef, Event) -> ok

Types:
- FsmRef = Name | {Name,Node} | {global,GlobalName} | pid()
- Name = Node = atom()
- GlobalName = term()
- Event = term()

Sends an event asynchronously to the gen_fsm FsmRef and returns ok immediately. The gen_fsm will call Module:handle_event/3 to handle the event.

See send_event/2 [page 227] for a description of the arguments.

The difference between send_event and send_all_state_event is which callback function is used to handle the event. This function is useful when sending events that are handled the same way in every state, as only one handle_event clause is needed to handle the event instead of one clause in each state name function.

sync_send_event(FsmRef, Event) -> Reply
sync_send_event(FsmRef, Event, Timeout) -> Reply

Types:
- FsmRef = Name | {Name,Node} | {global,GlobalName} | pid()
- Name = Node = atom()
- GlobalName = term()
- Event = term()
- Timeout = int() > 0 | infinity
- Reply = term()

Sends an event to the gen_fsm FsmRef and waits until a reply arrives or a timeout occurs. The gen_fsm will call Module:StateName/3 to handle the event, where StateName is the name of the current state of the gen_fsm.

See send_event/2 [page 227] for a description of FsmRef and Event.

Timeout is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom infinity to wait indefinitely. Default value is 5000. If no reply is received within the specified time, the function call fails.

The return value Reply is defined in the return value of Module:StateName/3.

The ancient behaviour of sometimes consuming the server exit message if the server died during the call while linked to the client has been removed in OTP R12B/Erlang 5.6.

sync_send_all_state_event(FsmRef, Event) -> Reply
sync_send_all_state_event(FsmRef, Event, Timeout) -> Reply

Types:
- FsmRef = Name | {Name,Node} | {global,GlobalName} | pid()
- Name = Node = atom()
- GlobalName = term()
- Event = term()
- Timeout = int() > 0 | infinity
- Reply = term()
Sends an event to the gen_fsm FsmRef and waits until a reply arrives or a timeout occurs. The gen_fsm will call Module:handle_sync_event/4 to handle the event. See send_event/2 [page 227] for a description of FsmRef and Event. See sync_send_event/3 [page 228] for a description of Timeout and Reply. See send_all_state_event/2 [page 228] for a discussion about the difference between sync_send_event and sync_send_all_state_event.

reply(Caller, Reply) -> true

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

This function can be used by a gen_fsm to explicitly send a reply to a client process that called sync_send_event/2,3 [page 228] or sync_send_all_state_event/2,3 [page 228], when the reply cannot be defined in the return value of Module:State/3 or Module:handle_sync_event/4.

Caller must be the From argument provided to the callback function. Reply is an arbitrary term, which will be given back to the client as the return value of sync_send_event/2,3 or sync_send_all_state_event/2,3.

send_event_after(Time, Event) -> Ref

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

Sends a delayed event internally in the gen_fsm that calls this function after Time ms. Returns immediately a reference that can be used to cancel the delayed send using cancel_timer/1 [page 230].

The gen_fsm will call Module:StateName/2 to handle the event, where StateName is the name of the current state of the gen_fsm at the time the delayed event is delivered. Event is an arbitrary term which is passed as one of the arguments to Module:StateName/2.

start_timer(Time, Msg) -> Ref

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

Sends a timeout event internally in the gen_fsm that calls this function after Time ms. Returns immediately a reference that can be used to cancel the timer using cancel_timer/1 [page 230].

The gen_fsm will call Module:StateName/2 to handle the event, where StateName is the name of the current state of the gen_fsm at the time the timeout message is delivered. Msg is an arbitrary term which is passed in the timeout message, {timeout, Ref, Msg}, as one of the arguments to Module:StateName/2.
cancel_timer(Ref) -> RemainingTime | false

Types:
- Ref = reference()
- RemainingTime = integer()

Cancels an internal timer referred by Ref in the gen_fsm that calls this function. Ref is a reference returned from send_event_after/2 [page 229] or start_timer/2 [page 229].

If the timer has already timed out, but the event not yet been delivered, it is cancelled as if it had not timed out, so there will be no false timer event after returning from this function.

Returns the remaining time in ms until the timer would have expired if Ref referred to an active timer, false otherwise.

enter_loop(Module, Options, StateName, StateData)
enter_loop(Module, Options, StateName, StateData, FsmName)
enter_loop(Module, Options, StateName, StateData, Timeout)
enter_loop(Module, Options, StateName, StateData, FsmName, Timeout)

Types:
- Module = atom()
- Options = [Option]
- Option = {debug, Dbgs}
- Dbgs = [Dbg]
- Dbg = trace | log | statistics
- {log_to_file, FileName} | {install, Func, FuncState}
- StateName = atom()
- StateData = term()
- FsmName = {local, Name} | {global, GlobalName}
- Name = atom()
- GlobalName = term()
- Timeout = int() | infinity

Makes an existing process into a gen_fsm. Does not return, instead the calling process will enter the gen_fsm receive loop and become a gen_fsm process. The process must have been started using one of the start functions in proc_lib, see proc_lib(3) [page 308]. 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 226]. 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 231]. Also, the callback module Module does not need to export an init/1 function.

Failure: If the calling process was not started by a proc_lib start function, or if it is not registered according to FsmName.
CALLBACK FUNCTIONS

The following functions should be exported from a gen\_fsm callback module.
In the description, the expression state name is used to denote a state of the state machine. state data is used to denote the internal state of the Erlang process which implements the state machine.

Exports

Module:init(Args) -> Result

Types:
- Args = term()
- Return = {ok,StateName,StateData} | {ok,StateName,StateData,Timeout} |
  {ok,StateName,StateData,hibernate} | {stop,Reason} | ignore
  StateName = atom()
  StateData = term()
  Timeout = int() > 0 | infinity
  Reason = term()

Whenever a gen\_fsm is started using gen\_fsm:start/3,4 [page 227] or gen\_fsm:start\_link/3,4 [page 226], this function is called by the new process to initialize. Args is the Args argument provided to the start function.
If initialization is successful, the function should return {ok,StateName,StateData}, {ok,StateName,StateData,Timeout} or {ok,StateName,StateData,hibernate}, where StateName is the initial state name and StateData the initial state data of the gen\_fsm.
If an integer timeout value is provided, a timeout will occur unless an event or a message is received within Timeout milliseconds. A timeout is represented by the atom timeout and should be handled by the Module:StateName/2 callback functions. The atom infinity can be used to wait indefinitely, this is the default value.
If hibernate is specified instead of a timeout value, the process will go into hibernation when waiting for the next message to arrive (by calling proc\_lib:hibernate/3 [page 312]).
If something goes wrong during the initialization the function should return {stop,Reason}, where Reason is any term, or ignore.

Module:StateName(Event, StateData) -> Result

Types:
- Event = timeout | term()
- StateData = term()
- Result = {next\_state,NextStateName,NewStateData} |
  {next\_state,NextStateName,NewStateData,Timeout} |
  {next\_state,NextStateName,NewStateData,hibernate} |
  {stop,Reason,NewStateData}
There should be one instance of this function for each possible state name. Whenever a gen_fsm receives an event sent using gen_fsm:send_event/2 [page 227], the instance of this function with the same name as the current state name StateName is called to handle the event. It is also called if a timeout occurs.

Event is either the atom timeout, if a timeout has occurred, or the Event argument provided to send_event/2.

StateData is the state data of the gen_fsm.

If the function returns \{next_state,NextStateName,NewStateData\}, \{next_state,NextStateName,NewStateData,Timeout\} or \{next_state,NextStateName,NewStateData,hibernate\}, the gen_fsm will continue executing with the current state name set to NextStateName and with the possibly updated state data NewStateData. See Module:init/1 for a description of Timeout and hibernate.

If the function returns \{stop,Reason,NewStateData\}, the gen_fsm will call Module:terminate(Reason,NewStateData) and terminate.

Module:handle_event(Event, StateName, StateData) -> Result

Types:
- Event = term()
- StateName = atom()
- StateData = term()
- Result = \{next_state,NextStateName,NewStateData\} |
  \{next_state,NextStateName,NewStateData,Timeout\} |
  \{next_state,NextStateName,NewStateData,hibernate\} |
  \{stop,Reason,NewStateData\} |
  NextStateName = atom()
  NewStateData = term()
  Timeout = int() > 0 | infinity
  Reason = term()

Whenever a gen_fsm receives an event sent using gen_fsm:send_all_state_event/2 [page 228], this function is called to handle the event. StateName is the current state name of the gen_fsm.

See Module:StateName/2 for a description of the other arguments and possible return values.

Module:StateName(Event, From, StateData) -> Result

Types:
- Event = term()
- From = \{pid(),Tag\}
- StateData = term()
- Result = \{reply,Reply,NextStateName,NewStateData\}
There should be one instance of this function for each possible state name. Whenever a \texttt{gen_fsm} receives an event sent using \texttt{gen_fsm:sync_send_event/2,3} [page 228], the instance of this function with the same name as the current state name \texttt{StateName} is called to handle the event. \texttt{Event} is the \texttt{Event} argument provided to \texttt{sync_send_event}. \texttt{From} is a tuple \{\texttt{Pid},\texttt{Tag}\} where \texttt{Pid} is the pid of the process which called \texttt{sync_send_event/2,3} and \texttt{Tag} is a unique tag. \texttt{StateData} is the state data of the \texttt{gen_fsm}. If the function returns \{\texttt{reply,Reply,NewStateName,NewStateData}, \texttt{Timeout}\} or \{\texttt{reply,Reply,NewStateName,NewStateData,hibernate}\}, \texttt{Reply} will be given back to \texttt{From} as the return value of \texttt{sync_send_event/2,3}. The \texttt{gen_fsm} then continues executing with the current state name set to \texttt{NewStateName} and with the possibly updated state data \texttt{NewStateData}. See Module:init/1 for a description of \texttt{Timeout} and \texttt{hibernate}. If the function returns \{\texttt{next_state,NewStateName,NewStateData}, \texttt{Timeout}\} or \{\texttt{next_state,NewStateName,NewStateData,hibernate}\}, the \texttt{gen_fsm} will continue executing in \texttt{NewStateName} with \texttt{NewStateData}. Any reply to \texttt{From} must be given explicitly using \texttt{gen_fsm:reply/2} [page 229]. If the function returns \{\texttt{stop,Reason,Reply,NewStateData}\}, \texttt{Reply} will be given back to \texttt{From}. If the function returns \{\texttt{stop,Reason,NewStateData}\}, any reply to \texttt{From} must be given explicitly using \texttt{gen_fsm:reply/2}. The \texttt{gen_fsm} will then call Module:terminate(\texttt{Reason,NewStateData}) and terminate.

\begin{verbatim}
Module:handle_sync_event(Event, From, StateName, StateData) -> Result

Types:
\begin{itemize}
\item Event = term()
\item From = \{\texttt{pid()},\texttt{Tag}\}
\item StateName = atom()
\item StateData = term()
\item Result = \{\texttt{reply,Reply,NewStateName,NewStateData}\}
\item | \{\texttt{reply,Reply,NewStateName,NewStateData,Timeout}\}
\item | \{\texttt{reply,Reply,NewStateName,NewStateData,hibernate}\}
\item | \{\texttt{next_state,NewStateName,NewStateData}\}
\item | \{\texttt{next_state,NewStateName,NewStateData,Timeout}\}
\end{itemize}
\end{verbatim}
Whenever a gen_fsm receives an event sent using gen_fsm:sync_send_all_state_event/2,3 [page 228], this function is called to handle the event.

StateName is the current state name of the gen_fsm.

See Module:StateMachine/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,hibernate} 
  | {stop,Reason,Reply,NewStateData} | {stop,Reason,NewStateData}
- Reply = term()
- 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:StateMachine/2 for a description of the other arguments and possible return values.

Module:terminate(Reason, StateName, StateData)

Types:
- Reason = normal | shutdown | {shutdown,term()} | term()
- StateName = atom()
- StateData = term()
This function is called by a gen_fsm when it is about to terminate. It should be the
opposite of Module:init/1 and do any necessary cleaning up. When it returns, the
gen_fsm terminates with Reason. The return value is ignored.

Reason is a term denoting the stop reason, StateName is the current state name, and
StateData is the state data of the gen_fsm.

Reason depends on why the gen_fsm is terminating. If it is because another callback
function has returned a stop tuple {stop,...}, Reason will have the value specified in
that tuple. If it is due to a failure, Reason is the error reason.

If the gen_fsm is part of a supervision tree and is ordered by its supervisor to terminate,
this function will be called with Reason=shutdown if the following conditions apply:

- the gen_fsm has been set to trap exit signals, and
- the shutdown strategy as defined in the supervisor's child specification is an integer
timeout value, not brutal_kill.

Otherwise, the gen_fsm will be immediately terminated.

Note that for any other reason than normal, shutdown, or {shutdown,Term} 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

genevent(3) [page 215], genserver(3) [page 236], supervisor(3) [page 444],
proc_lib(3) [page 308], sys(3) [page 454]
A behaviour module for implementing the server of a client-server relation. A generic server process (gen_server) implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into an OTP supervision tree. Refer to [OTP Design Principles] for more information.

A gen_server assumes all specific parts to be located in a callback module exporting a pre-defined set of functions. The relationship between the behaviour functions and the callback functions can be illustrated as follows:

```
<table>
<thead>
<tr>
<th>gen_server module</th>
<th>Callback module</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen_server:start_link</td>
<td>Module:init/1</td>
</tr>
<tr>
<td>gen_server:call</td>
<td>Module:handle_call/3</td>
</tr>
<tr>
<td>gen_server:multi_call</td>
<td>Module:handle_call/3</td>
</tr>
<tr>
<td>gen_server:cast</td>
<td>Module:cast/2</td>
</tr>
<tr>
<td>gen_server:abcast</td>
<td>Module:info/2</td>
</tr>
<tr>
<td></td>
<td>Module:terminate/2</td>
</tr>
<tr>
<td></td>
<td>Module:code_change/3</td>
</tr>
</tbody>
</table>
```

If a callback function fails or returns a bad value, the gen_server will terminate.

The sys module can be used for debugging a gen_server.

Note that a gen_server does not trap exit signals automatically, this must be explicitly initiated in the callback module.

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

The gen_server process can go into hibernation (see [erlang(3)]) if a callback function specifies 'hibernate' instead of a timeout value. This might be useful if the server is expected to be idle for a long time. However this feature should be used with care as hibernation implies at least two garbage collections (when hibernating and shortly after waking up) and is not something you’d want to do between each call to a busy server.
Exports

```
start_link(Module, Args, Options) -> Result
start_link(ServerName, Module, Args, Options) -> Result
```

Types:
- `ServerName` = `{local,Name} | {global,GlobalName}`
- `Name` = `atom()`
- `GlobalName` = `term()`
- `Module` = `atom()`
- `Args` = `term()`
- `Options` = `[Option]`
- `Option` = `{debug,Dbgs} | {timeout,Time} | {spawn_opt,SOpts}
- `Dbgs` = `[Dbg]
- `Dbg` = `trace` | `log` | `statistics` | `log_to_file,FileName` | `{install,{Func,FuncState}}`
- `SOpts` = `[term()]`
- `Result` = `{ok,Pid} | ignore | {error,Error}
- `Pid` = `pid()`
- `Error` = `{already_started,Pid} | term()`

Creates a gen_server process as part of a supervision tree. The function should be called, directly or indirectly, by the supervisor. It will, among other things, ensure that the gen_server is linked to the supervisor.

The gen_server process calls `Module:init/1` to initialize. To ensure a synchronized start-up procedure, `start_link/3,4` does not return until `Module:init/1` has returned.

If `ServerName` = `{local,Name}` the gen_server is registered locally as `Name` using `register/2`. If `ServerName` = `{global,GlobalName}` the gen_server is registered globally as `GlobalName` using `global:register_name/2`. If no name is provided, the gen_server is not registered.

`Module` is the name of the callback module.

`Args` is an arbitrary term which is passed as the argument to `Module:init/1`.

If the option `{timeout,Time}` is present, the gen_server is allowed to spend `Time` milliseconds initializing or it will be terminated and the start function will return `{error,timeout}`.

If the option `{debug,Dbgs}` is present, the corresponding sys function will be called for each item in `Dbgs`. See `sys(3)` [page 454].

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.

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

Types:
- ServerName = \{local, Name\} | \{global, GlobalName\}
- Name = atom()
- GlobalName = term()
- Module = atom()
- Args = term()
- Options = [Option]
- Option = \{debug, D bgs\} | \{timeout, Time\} | \{spawn_opt, SO pts\}
- D bgs = [Dbg]
- Dbg = trace | log | statistics | \{log_to_file, FileName\} | \{install, \{Func, FuncState\}\}
- SO pts = [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 \[page 237\] for a description of arguments and return values.

\[\text{call(ServerRef, Request) -> Reply}\]
\[\text{call(ServerRef, Request, Timeout) -> Reply}\]

Types:
- ServerRef = Name | \{Name, Node\} | \{global, GlobalName\} | pid()
- Node = atom()
- GlobalName = term()
- Request = term()
- Timeout = int() > 0 | infinity
- Reply = term()

Makes a synchronous call to the gen_server ServerRef by sending a request and waiting 
until a reply arrives or a timeout occurs. The gen_server will call 
Module: handle_call/3 to handle the request.

ServerRef can be:
- the pid,
- Name, if the gen_server is locally registered,
- \{Name, Node\}, if the gen_server is locally registered at another node, or
- \{global, GlobalName\}, if the gen_server is globally registered.
Request is an arbitrary term which is passed as one of the arguments to Module:handle_call/3.

Timeout is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom infinity to wait indefinitely. Default value is 5000. If no reply is received within the specified time, the function call fails.

The return value Reply is defined in the return value of Module:handle_call/3.

The call may fail for several reasons, including timeout and the called gen_server dying before or during the call.

The ancient behaviour of sometimes consuming the server exit message if the server died during the call while linked to the client has been removed in OTP R12B/Erlang 5.6.

\[
\text{multi_call(Name, Request)} \rightarrow \text{Result} \\
\text{multi_call(Nodes, Name, Request)} \rightarrow \text{Result} \\
\text{multi_call(Nodes, Name, Request, Timeout)} \rightarrow \text{Result}
\]

Types:
- Nodes = [Node]
- Node = atom()
- Name = atom()
- Request = term()
- Timeout = int() >= 0 | infinity
- Result = {Replies, BadNodes}
- Replies = [{Node, Reply}]
- Reply = term()
- BadNodes = [Node]

Makes a synchronous call to all gen_servers locally registered as Name at the specified nodes by first sending a request to every node and then waiting for the replies. The gen_servers will call Module:handle_call/3 to handle the request.

The function returns a tuple \( \{\text{Replies}, \text{BadNodes}\} \) where \( \text{Replies} \) is a list of \( \{\text{Node}, \text{Reply}\} \) and \( \text{BadNodes} \) is a list of node that either did not exist, or where the gen_server Name did not exist or did not reply.

Nodes is a list of node names to which the request should be sent. Default value is the list of all known nodes \([\text{node()}|\text{nodes()}] \). Name is the locally registered name of each gen_server.

Request is an arbitrary term which is passed as one of the arguments to Module:handle_call/3.

Timeout is an integer greater than zero which specifies how many milliseconds to wait for each reply, or the atom infinity to wait indefinitely. Default value is infinity. If no reply is received from a node within the specified time, the node is added to BadNodes.

When a reply Reply is received from the gen_server at a node Node, \( \{\text{Node}, \text{Reply}\} \) is added to Reponses. Reply is defined in the return value of Module:handle_call/3.
Warning:
If one of the nodes is not capable of process monitors, for example C or Java nodes, and the gen_server is not started when the requests are sent, but starts within 2 seconds, this function waits the whole Timeout, which may be infinity. This problem does not exist if all nodes are Erlang nodes.

To avoid that late answers (after the timeout) pollutes the caller's message queue, a middleman process is used to do the actual calls. Late answers will then be discarded when they arrive to a terminated process.

cast(ServerRef, Request) -> ok
Types:
- ServerRef = Name | {Name,Node} | {global,GlobalName} | pid()
- Node = atom()
- GlobalName = term()
- Request = term()
Sends an asynchronous request to the gen_server ServerRef and returns ok immediately, ignoring if the destination node or gen_server does not exist. The gen_server will call Module:handle_cast/2 to handle the request.
See call/2,3 [page 238] for a description of ServerRef.
Request is an arbitrary term which is passed as one of the arguments to Module:handle_cast/2.

abcast(Name, Request) -> abcast
abcast(Nodes, Name, Request) -> abcast
Types:
- Nodes = [Node]
- Node = atom()
- Name = atom()
- Request = term()
Sends an asynchronous request to the gen_servers locally registered as Name at the specified nodes. The function returns immediately and ignores nodes that do not exist, or where the gen_server Name does not exist. The gen_servers will call Module:handle_cast/2 to handle the request.
See multi_call/2,3,4 [page 239] for a description of the arguments.

reply(Client, Reply) -> Result
Types:
- Client - see below
- Reply = term()
- Result = term()
This function can be used by a gen_server to explicitly send a reply to a client that called call/2,3 or multi_call/2,3,4, when the reply cannot be defined in the return value of Module:handle_call/3.

Client must be the From argument provided to the callback function. Reply is an arbitrary term, which will be given back to the client as the return value of call/2,3 or multi_call/2,3,4.

The return value Result is not further defined, and should always be ignored.

```
enter_loop(Module, Options, State)
enter_loop(Module, Options, State, ServerName)
enter_loop(Module, Options, State, Timeout)
enter_loop(Module, Options, State, ServerName, Timeout)
```

Types:
- Module = atom()
- Options = [Option]
- Option = {debug,Dbgs}
- Dbgs = [Dbg]
- Dbg = trace | log | statistics
- Name = atom()
- GlobalName = term()
- ServerName = {local,Name} | {global,GlobalName}
- State = 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 308]. 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 237]. 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 242]. Also, the callback module Module does not need to export an init/1 function.

Failure: If the calling process was not started by a proc_lib start function, or if it is not registered according to ServerName.

### CALLBACK FUNCTIONS

The following functions should be exported from a gen_server callback module.
Exports

Module:init(Args) -> Result

Types:
- Args = term()
- Result = {ok,State} | {ok,State,Timeout} | {ok,State,hibernate}
- | {stop,Reason} | ignore
- State = term()
- Timeout = int()>=0 | infinity
- Reason = term()

Whenever a gen_server is started using gen_server:start/3,4 [page 238] or
gen_server:start_link/3,4 [page 237], this function is called by the new process to
initialize. Args is the Args argument provided to the start function.

If the initialization is successful, the function should return {ok,State},
{ok,State,Timeout} or {ok,State,hibernate}, where State is the internal state of
the gen_server.

If an integer timeout value is provided, a timeout will occur unless a request or a
message is received within Timeout milliseconds. A timeout is represented by the atom
timeout which should be handled by the handle_info/2 callback function. The atom
infinity can be used to wait indefinitely, this is the default value.

If hibernate is specified instead of a timeout value, the process will go into hibernation
when waiting for the next message to arrive (by calling proc_lib:hibernate/3 [page
312]).

If something goes wrong during the initialization the function should return
{stop,Reason} where Reason is any term, or ignore.

Module:handle_call(Request, From, State) -> Result

Types:
- Request = term()
- From = {pid(),Tag}
- State = term()
- Result = {reply,Reply,NewState} | {reply,Reply,NewState,Timeout}
- | {reply,Reply,NewState,hibernate}
- | {noreply,NewState} | {noreply,NewState,Timeout}
- | {noreply,NewState,hibernate}
- | {stop,Reason,Reply,NewState} | {stop,Reason,NewState}
- Reply = term()
- NewState = term()
- Timeout = int()>=0 | infinity
- Reason = term()
Whenever a gen_server receives a request sent using gen_server:call/2,3 [page 238] or gen_server:multi_call/2,3,4 [page 239], this function is called to handle the request. Request is the Request argument provided to call or multi_call. From is a tuple \((\text{Pid}, \text{Tag})\) where \(\text{Pid}\) is the pid of the client and \(\text{Tag}\) is a unique tag. State is the internal state of the gen_server.

If the function returns \(\text{reply} = \text{Reply, NewState}\), \(\text{reply} = \text{Reply, NewState, Timeout}\) or \(\text{reply} = \text{Reply, NewState, hibernate}\), \(\text{Reply}\) will be given back to \(\text{From}\) as the return value of \(\text{call}/2,3\) or included in the return value of \(\text{multi_call}/2,3,4\). The gen_server then continues executing with the possibly updated internal state \(\text{NewState}\).

See Module:init/1 for a description of \(\text{Timeout}\) and \(\text{hibernate}\).

If the functions returns \(\text{noreply} = \text{NewState}\), \(\text{noreply} = \text{NewState, Timeout}\) or \(\text{noreply} = \text{NewState, hibernate}\), the gen_server will continue executing with \(\text{NewState}\). Any reply to \(\text{From}\) must be given explicitly using gen_server:reply/2 [page 240].

If the function returns \(\text{stop} = \text{Reason, Reply, NewState}\), \(\text{Reply}\) will be given back to \(\text{From}\). If the function returns \(\text{stop} = \text{Reason, NewState}\), any reply to \(\text{From}\) must be given explicitly using gen_server:reply/2. The gen_server will then call Module:terminate(\(\text{Reason, NewState}\)) and terminate.

Module:handle_cast(Request, State) -> Result

Types:
- Request = term()
- State = term()
- Result = \{noreply, NewState\} \| \{noreply, NewState, Timeout\} \| \{noreply, NewState, hibernate\} \| \{stop, Reason, NewState\}
- NewState = term()
- Timeout = int() \(\geq 0\) \| \text{infinity}
- Reason = term()

Whenever a gen_server receives a request sent using gen_server:cast/2 [page 240] or gen_server:abcast/2,3 [page 240], this function is called to handle the request. See Module:handle_call/3 for a description of the arguments and possible return values.

Module:handle_info(Info, State) -> Result

Types:
- Info = timeout \| term()
- State = term()
- Result = \{noreply, NewState\} \| \{noreply, NewState, Timeout\} \| \{noreply, NewState, hibernate\} \| \{stop, Reason, NewState\}
- NewState = term()
- Timeout = int() \(\geq 0\) \| \text{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_info/3 for a description of the other arguments and possible return values.

Module:terminate(Reason, State)

Types:
- Reason = normal | shutdown | {shutdown,term()} | 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, shutdown, or {shutdown,Term} 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 215], gen_fsm(3) [page 225], supervisor(3) [page 444], proc_lib(3) [page 308], sys(3) [page 454]
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 the STDLIB Users Guide.

**Warning:**

As of R13A, data supplied to the `put_chars` [page 247] function should be in the `chardata()` format described below. This means that programs supplying binaries to this function need to convert them to UTF-8 before trying to output the data on an `io_device()`.

If an `io_device()` is set in binary mode, the functions `get_chars` [page 247] and `get_line` [page 247] may return binaries instead of lists. The binaries will, as of R13A, be encoded in UTF-8.

To work with binaries in ISO-latin-1 encoding, use the `[file]` module instead.

For conversion functions between character encodings, see the `[unicode]` module.

**DATA TYPES**

`io_device()`

- as returned by `file:open/2`, a process handling IO protocols

`unicode_binary()` = `binary()` with characters encoded in UTF-8 coding standard

`unicode_char()` = `integer()` representing valid unicode codepoint

`chardata()` = `charlist()` | `unicode_binary()`

`charlist()` = `[unicode_char() | unicode_binary() | charlist()]`

- a `unicode_binary` is allowed as the tail of the list
Exports

columns([IoDevice]) -> {ok,int()} | {error, enotsup}
Types:
  • IoDevice = io_device()
Retrieves the number of columns of the IoDevice (i.e. the width of a terminal). The function only succeeds for terminal devices, for all other devices the function returns {error, enotsup}

put_chars([IoDevice,] IoData) -> ok
Types:
  • IoDevice = io_device()
  • IoData = chardata()
Writes the characters of IoData to the io_server() (IoDevice).

nl([IoDevice]) -> ok
Types:
  • IoDevice = io_device()
Writes new line to the standard output (IoDevice).

get_chars([IoDevice,] Prompt, Count) -> Data | eof
Types:
  • IoDevice = io_device()
  • Prompt = atom() | string()
  • Count = int()
  • Data = [ unicode_char()] | unicode_binary()
Reads Count characters from standard input (IoDevice), prompting it with Prompt. It returns:

  Data  The input characters. If the device supports Unicode, the data may represent codepoints larger than 255 (the latin1 range). If the io_server() is set to deliver binaries, they will be encoded in UTF-8 (regardless of if the device actually supports Unicode or not).
  eof   End of file was encountered.
  {error,Reason} Other (rare) error condition, for instance {error,estale} if reading from an NFS file system.

get_line([IoDevice,] Prompt) -> Data | eof | {error,Reason}
Types:
  • IoDevice = io_device()
  • Prompt = atom() | string()
  • Data = [ unicode_char()] | unicode_binary()
Reads a line from the standard input (IoDevice), prompting it with Prompt. It returns:
Data: The characters in the line terminated by a LF (or end of file). If the device supports Unicode, the data may represent codepoints larger than 255 (the latin1 range). If the io_server() is set to deliver binaries, they will be encoded in UTF-8 (regardless of if the device actually supports Unicode or not).

eof: End of file was encountered.

{error,Reason}: Other (rare) error condition, for instance {error,estale} if reading from an NFS file system.

getopts([IoDevice]) -> Opts

Types:
- IoDevice = io_device()
- Opts = [O pt]
- O pt = {atom(),Value}
- Value = term()

This function requests all available options and their current values for a specific io_device(). Example:

1> {ok,F} = file:open("/dev/null", [read]).
   {ok,<0.42.0>}
2> io:getopts(F).
   [{binary,false},{encoding,latin1}]

Here the file I/O-server returns all available options for a file, which are the expected ones, encoding and binary. The standard shell however has some more options:

3> io:getopts().
   [{expand_fun,#Fun<group.0.120017273>},
    {echo,true},
    {binary,false},
    {encoding,unicode}]

This example is, as can be seen, run in an environment where the terminal supports Unicode input and output.

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

Types:
- IoDevice = io_device()
- Opts = [O pt]
- O pt = atom() | {atom(),Value}
- Value = term()
- Reason = term()

Set options for the io_device() (IoDevice).

Possible options and values vary depending on the actual io_device(). For a list of supported options and their current values on a specific device, use the getopts/1 function.

The options and values supported by the current OTP io_devices are:
binary, list or \{binary, bool()\}: If set in binary mode (binary or \{binary, true\}), the \texttt{io_server()} sends binary data (encoded in UTF-8) as answers to the \texttt{get_line}, \texttt{get_chars} and, if possible, \texttt{get_until} requests (see the I/O protocol description in \texttt{STDLIB User's Guide} for details). The immediate effect is that \texttt{get_chars(2,3)} and \texttt{get_line(1,2)} return UTF-8 binaries instead of lists of chars for the affected device.

By default, all \texttt{io_device}s in OTP are set in list mode, but the \texttt{io} functions can handle any of these modes and so should other, user written, modules behaving as clients to I/O-servers.

This option is supported by the standard shell (\texttt{group.erl}), the \texttt{‘oldshell’} (\texttt{user.erl}) and the file I/O servers.

\{echo, bool()\}: Denotes if the terminal should echo input. Only supported for the standard shell I/O-server (\texttt{group.erl}).

\{expand_fun, 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 \texttt{get_line(1,2)}.

The function is called with the current line, up to the cursor, as a reversed string. It should return a three-tuple: \{yes|no, string(), [string(), ...]\}. The first element gives a beep if \texttt{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”):

\[
\begin{align*}
\text{fun}("") & \rightarrow \{\text{yes, "quit", []}\}; \\
\_ & \rightarrow \{\text{no, ",", ["quit"]}\}
\end{align*}
\]

This option is supported by the standard shell only (\texttt{group.erl}).

\{encoding, latin1 | unicode\}: Specifies how characters are input or output from or to the actual device, implying that i.e. a terminal is set to handle Unicode input and output or a file is set to handle UTF-8 data encoding.

The option does not affect how data is returned from the \texttt{io} functions or how it is sent in the I/O-protocol, it only affects how the \texttt{io_device()} is to handle Unicode characters towards the “physical” device.

The standard shell will be set for either unicode or latin1 encoding when the system is started. The actual encoding is set with the help of the "LANG" or "LC\_CTYPE" environment variables on Unix-like system or by other means on other systems. The bottom line is that the user can input Unicode characters and the device will be in \{encoding, unicode\} mode if the device supports it. The mode can be changed, if the assumption of the runtime system is wrong, by setting this option.

The \texttt{io_device()} used when Erlang is started with the "-oldshell" or "-noshell" flags is by default set to latin1 encoding, meaning that any characters beyond codepoint 255 will be escaped and that input is expected to be plain 8-bit ISO-latin-1. If the encoding is changed to Unicode, input and output from the standard file descriptors will be in UTF-8 (regardless of operating system).

Files can also be set in \{encoding, unicode\}, meaning that data is written and read as UTF-8. More encodings are possible for files, see below.

\{encoding, unicode | latin1\} is supported by both the standard shell (\texttt{group.erl} including werl on windows), the ‘oldshell’ (\texttt{user.erl}) and the file I/O servers.
For disk files, the encoding can be set to various UTF variants. This will have the effect that data is expected to be read as the specified encoding from the file and the data will be written in the specified encoding to the disk file. 

- {encoding, utf8} will have the same effect as {encoding,unicode} on files.
- The extended encodings are only supported on disk files (opened by the [file:open/2] function).

**write([IoDevice,] Term) -> ok**

Types:
- IoDevice = io_device()
- Term = term()

Writers 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 | {error, ErrorInfo, EndLine}
- Term = term()
- EndLine = int()
- ErrorInfo - see section Error Information below

Reads a term Term from IoDevice, prompting it with Prompt. Reading starts at line number StartLine. It returns:

- {ok, Term, EndLine} The parsing was successful.
- {eof, EndLine} End of file was encountered.
- {error, ErrorInfo, EndLine} The parsing failed.
fwrite(Format) ->
fwrite([IoDevice,] Format, Data) -> ok
format(Format) ->
format([IoDevice,] Format, Data) -> ok

Types:
- IoDevice = io_device()
- Format = atom() | string() | binary()
- Data = [term()]

Writes the items in Data ([ ]) on the standard output (IoDevice) in accordance with Format. Format contains plain characters which are copied to the output device, and control sequences for formatting, see below. If Format is an atom or a binary, it is first converted to a list with the aid of atom_to_list/1 or binary_to_list/1.

1> io:fwrite("Hello world!\n", []). Hello world!
ok

The general format of a control sequence is \~F.P.PadModC. 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).

Mod is the control sequence modifier. It is either a single character (currently only 't', for unicode translation, is supported) that changes the interpretation of Data.

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:

2> io:fwrite("\~10.5c\~10.5c\~5c\n", [$a, $b, $c]).
 | aaaa|bbbb |cccccc|
ok

If the Unicode translation modifier ('t') is in effect, the integer argument can be any number representing a valid unicode codepoint, otherwise it should be an integer less than or equal to 255, otherwise it is masked with 16#FF:
The argument is a float which is written as \[-\]d.dd, 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 \(\pm\)f, if it is \(\geq 0.1\) and \(\leq 10000.0\). 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, if no Unicode translation modifier is present, an [I/O list], a binary, or an atom. If the Unicode translation modifier (‘t’) is in effect, the argument is chardata(), meaning that binaries are in UTF-8. 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:

A list with integers larger than 255 is considered an error if the Unicode translation modifier is not given:

Writes data with the standard syntax. This is used to output Erlang terms. Atoms are printed within quotes if they contain embedded non-printable characters, and floats are printed accurately as the shortest, correctly rounded string.

Writes the data with standard syntax in the same way as ~w, but breaks terms whose printed representation is longer than one line into many lines and indents each line sensibly. It also tries to detect lists of printable characters and to output these as strings. For example:
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 = \", [T]).
Here T = [{attributes,[[id,age,1.5],[mode,explicit],

{typename,"INTEGER"}],

[[id,cho],[mode,explicit],[typename,'Cho']]],

{typename,'Person'},

{tag,{'PRIVATE',3}},{mode,implicit}].
ok

W writes data in the same way as \w, but takes an extra argument which is the maximum depth to which terms are printed. Anything below this depth is replaced with \... For example, using T above:

9> io:fwrite("W\n", [T,9]).
[[attributes,[[id,age,1.5],[mode,explicit],[typename,...]],

[[id,cho],[mode,...],[...]],{typename,'Person'},

{tag,{'PRIVATE',3}},{mode,implicit}].
ok

If the maximum depth has been reached, then it is impossible to read in the resultant output. Also, the \... form in a tuple denotes that there are more elements in the tuple but these are below the print depth.

P writes data in the same way as \p, but takes an extra argument which is the maximum depth to which terms are printed. Anything below this depth is replaced with \... For example:
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:

\[10\] io:fwrite("-62P-n", [T,9]).  
[{attributes,[[[id,age,1.5],[mode,explicit],[typename,...]],  
    [[id,cho],[mode,...],[...]]],  
  {typename,'Person'},  
  {tag,{'PRIVATE',3}},  
  {mode,implicit}]}  
ok

\[11\] B writes an integer in base 2..36, the default base is 10. A leading dash is printed for negative integers.
The precision field selects base. For example:

\[11\] io:fwrite("-1.16B-n", [31]).  
1F  
ok
\[12\] 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:

\[12\] io:fwrite("-2B-n", [-19]).  
-10011  
ok
\[13\] 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:

\[13\] io:fwrite("-36B-n", [5*36+35]).  
5Z  
ok

X Like B, but takes an extra argument that is a prefix to insert before the number, but after the leading dash, if any.
The prefix can be a possibly deep list of characters or an atom.

\[14\] X 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:

\[14\] io:fwrite("^X-n", [31,"10#"]).  
10#31  
ok
\[15\] X 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:

\[15\] io:fwrite("^16X-n", [-31,"0x"]).  
-0x1F  
ok

# Like B, but prints the number with an Erlang style ‘#’-separated base prefix.

\[16\] # 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:

\[16\] io:fwrite("^10#-n", [31]).  
10#31  
ok
\[17\] # 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:

\[17\] io:fwrite("^16#-n", [-31]).  
-16#1F  
ok

b Like B, but prints lowercase letters.
  x Like X, but prints lowercase letters.
  + Like #, but prints lowercase letters.
  n Writes a new line.
  i Ignores the next term.

Returns:
The formatting succeeded.

If an error occurs, there is no output. For example:

```erl
18> io:fwrite("~s ~w ~i ~w ~c ~n", [\'abc def\', \'abc def\', \{foo, 1\}, \{foo, 1\}, 65]).
abc def 'abc def' \{foo,1\} A
ok
19> io:fwrite("~s", [65]).
** exception exit: \{badarg, [{io,format,\{0.22.0\},"~s","A"}],
{erl_eval,do_apply,5},
{shell,exprs,6},
{shell,eval_exprs,6},
{shell,eval_loop,3}\}
in function io:o_request/2
```

In this example, an attempt was made to output the single character '65' with the aid of the string formatting directive "~s".

`fread([IoDevice,] Prompt, Format) -> Result`

Types:
- IoDevice = io_device()
- Prompt = atom() | string()
- Format = string()
- Result = \{ok, Terms\} | eof | \{error, What\}
- Terms = [term()]
- What = term()

Reads characters from the standard input (IoDevice), prompting it with Prompt. Interprets the characters in accordance with Format. Format contains control sequences which directs the interpretation of the input.

Format may contain:

- White space characters (SPACE, TAB and NEWLINE) which cause input to be read to the next non-white space character.
- Ordinary characters which must match the next input character.
- Control sequences, which have the general format "~*FMC". 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, M is an optional translation modifier (of which 't' is the only currently supported, meaning Unicode translation) 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.

s A string of non-white-space characters is read. If a field width has been specified, this number of characters are read and all trailing white-space characters are stripped. An Erlang string (list of characters) is returned. If Unicode translation is in effect ("ts"), characters larger than 255 are accepted, otherwise not. With the translation modifier, the list returned may as a consequence also contain integers larger than 255:

```
1> io:fread("Prompt","~s").
Prompt> <Characters beyond latin1 range not printable in this medium>
{error,[fread,string]}
2> io:fread("Prompt","~ts").
Prompt> <Characters beyond latin1 range not printable in this medium>
{ok,[[1091,1085,1080,1094,1086,1076,1077]]}
```

a Similar to s, but the resulting string is converted into an atom. The Unicode translation modifier is not allowed (atoms can not contain characters beyond the latin1 range).

c The number of characters equal to the field width are read (default is 1) and returned as an Erlang string. However, leading and trailing white-space characters are not omitted as they are with s. All characters are returned. The Unicode translation modifier works as with s:

```
1> io:fread("Prompt","~c").
Prompt> <Character beyond latin1 range not printable in this medium>
{error,[fread,string]}
2> io:fread("Prompt","~tc").
Prompt> <Character beyond latin1 range not printable in this medium>
{ok,[[1091]]}
```

l Returns the number of characters which have been scanned up to that point, including white-space characters.

It returns:

```
{ok, Terms} The read was successful and Terms is the list of successfully matched and read items
```

eof End of file was encountered.

```
{error, What} The read operation failed and the parameter What gives a hint about the error.
```

Examples:

20> io:fread('enter>', "f"f"f").
enter> 1.9 35.5e3 15.0
{ok,[1.9,3.55e4,15.0]}
21> io:fread('enter>', "~10f"d").
enter> 5.67899
{ok,[5.678,99]}
22> io:fread('enter>', "~10s:"10c:"").
enter> alan joe
{ok, ["alan", " joe "]}

rows([IoDevice]) -> {ok, int()} | {error, enotsup}

Types:
- IoDevice = io_device()

Retrieves the number of rows of the IoDevice (i.e. the height of a terminal). The function only succeeds for terminal devices, for all other devices the function returns {error, enotsup}

scan_erl_exprs(Prompt) ->

scan_erl_exprs([IoDevice,] Prompt, StartLine) -> Result

Types:
- IoDevice = io_device()
- Prompt = atom() | string()
- StartLine = int()
- Result = {ok, Tokens, EndLine} | {eof, EndLine} | {error, ErrorInfo, EndLine}
- Tokens – see erl_scan(3)
- EndLine = int()
- ErrorInfo – see section Error Information below

Reads data from the standard input (IoDevice), prompting it with Prompt. Reading starts at line number StartLine (1). The data is tokenized as if it were a sequence of Erlang expressions until a final '.' is reached. This token is also returned. It returns:

- {ok, Tokens, EndLine} The tokenization succeeded.
- {eof, EndLine} End of file was encountered.
- {error, ErrorInfo, EndLine} An error occurred.

Example:
23> io:scan_erl_exprs('enter>').
enter> abc(), "hey".
{ok, [{atom,1,abc},{"'","1"},{',',"1"}],{"'","1"},{string,1,"hey"},{dot,1}],2}
24> io:scan_erl_exprs('enter>').
enter> 1.0er.
{error, [{1,erl_scan,[illegal,float]],2}}

scan_erl_form(Prompt) ->

scan_erl_form([IoDevice,] Prompt, StartLine) -> Result

Types:
- IoDevice = io_device()
- Prompt = atom() | string()
- StartLine = int()
- Result = {ok, Tokens, EndLine} | {eof, EndLine} | {error, ErrorInfo, EndLine}
- Tokens – see erl_scan(3)
- EndLine = int()
- ErrorInfo – see section Error Information below
Reads data from the standard input (IoDevice), prompting it with Prompt. Starts reading at line number StartLine (1). The data is tokenized as if it were an Erlang form - one of the valid Erlang expressions in an Erlang source file - until a final ‘.’ is reached. This last token is also returned. The return values are the same as for scan ErlExprs/1,2,3 above.

\[
\text{parse ErlExprs}(\text{Prompt}) \rightarrow \\
\text{parse ErlExprs}([\text{IoDevice,}] \text{Prompt, StartLine}) \rightarrow \text{Result}
\]

Types:
- IoDevice = io device()
- Prompt = atom() | string()
- StartLine = int()
- Result = \{ok, ExprList, EndLine\} | \{eof, EndLine\} | \{error, ErrorInfo, EndLine\}
- ExprList = see ErlParse(3)
- EndLine = int()
- ErrorInfo = see section Error Information below

Reads data from the standard input (IoDevice), prompting it with Prompt. Starts reading at line number StartLine (1). The data is tokenized and parsed as if it were a sequence of Erlang expressions until a final ‘.’ is reached. It returns:

\{ok, ExprList, EndLine\} The parsing was successful.
\{eof, EndLine\} End of file was encountered.
\{error, ErrorInfo, EndLine\} An error occurred.

Example:

25> io:parse ErlExprs('enter>').'.
enter> abc(), "hey".
\{ok, [\{call,1,\{atom,1,abc\},[]\},\{string,1,"hey"\}],2\}

26> io:parse ErlExprs('enter>').'.
enter> abc("hey").
\{error, [1,erl_parse,"syntax error before: ", ["."\]]],2\}

\[
\text{parse ErlForm}(\text{Prompt}) \rightarrow \\
\text{parse ErlForm}([\text{IoDevice,}] \text{Prompt, StartLine}) \rightarrow \text{Result}
\]

Types:
- IoDevice = io device()
- Prompt = atom() | string()
- StartLine = int()
- Result = \{ok, AbsForm, EndLine\} | \{eof, EndLine\} | \{error, ErrorInfo, EndLine\}
- AbsForm = see ErlParse(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:
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.

Standard Error

In certain situations, especially when the standard output is redirected, access to an io_server() specific for error messages might be convenient. The io_device 'standard_error' can be used to direct output to whatever the current operating system considers a suitable device for error output. Example on a Unix-like operating system:


Error: error 11

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)
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:

```erlang
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.

\[
\text{fwrite}((\text{Format}, \text{Data})) \rightarrow \\
\text{format}((\text{Format}, \text{Data})) \rightarrow \text{chars()} \mid \text{UnicodeList}
\]

Types:
- Format = atom() \mid string() \mid binary()
- Data = [\text{term() }]
- UnicodeList = [\text{Unicode}]
- Unicode = int() representing valid unicode codepoint

Returns a character list which represents Data formatted in accordance with Format. See io:fwrite/1,2,3 [page 251] 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. If (and only if) the Unicode translation modifier is used in the format string (i.e. \text{"ts} or \text{"tc}), the resulting list may contain characters beyond the ISO-latin-1 character range (in other words, numbers larger than 255). If so, the result is not an ordinary Erlang string(), but can well be used in any context where Unicode data is allowed.

\[
\text{fread}((\text{Format}, \text{String})) \rightarrow \text{Result}
\]

Types:
- Format = String = string()
- Result = \{ok, InputList, LeftOverChars} \mid \{more, RestFormat, Nchars, InputStack\}
  \mid \{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 255] 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", [15.6 17.3e-6 24.5]).
   {ok,[15.6,1.73e-5,24.5],[[]]}

fread(Continuation, String, Format) -> Return

Types:
- Continuation = see below
- String = Format = string()
- Return = \{done, Result, LeftOverChars\} \| \{more, Continuation\}
- Result = \{ok, InputList\} \| \{eof\} \| \{error, What\}
- InputList = chars()
- What = term()
- LeftOverChars = string()

This is the re-entrant formatted reader. The continuation of the first call to the
functions must be \[]. Refer to Armstrong, Virding, Williams, 'Concurrent
Programming in Erlang', Chapter 13 for a complete description of how the re-entrant
input scheme works.

The function returns:

\{done, Result, LeftOverChars\} The input is complete. The result is one of the
following:

\{ok, InputList\} The string was read. InputList is the list of successfully
matched and read items, and LeftOverChars are the remaining characters.
\{eof\} End of file has been encountered. LeftOverChars are the input characters not
used.
\{error, What\} An error occurred and the parameter What gives a hint about the
error.
\{more, Continuation\} More data is required to build a term. Continuation must be
passed to fread/3, when more data becomes available.

write_atom(Atom) -> chars()

Types:
- Atom = atom()

Returns the list of characters needed to print the atom Atom.

write_string(String) -> chars()

Types:
- String = string()

Returns the list of characters needed to print String as a string.

write_char(Integer) -> chars()

Types:
- Integer = int()

Returns the list of characters needed to print a character constant in the ISO-latin-1
character set.
indentation(String, StartIndent) \rightarrow int()

Types:
- String = string()
- StartIndent = int()

Returns the indentation if String has been printed, starting at StartIndent.

char_list(Term) \rightarrow bool()

Types:
- Term = term()

Returns true if Term is a flat list of characters in the ISO-latin-1 range, otherwise it returns false.

deep_char_list(Term) \rightarrow bool()

Types:
- Term = term()

Returns true if Term is a, possibly deep, list of characters in the ISO-latin-1 range, otherwise it returns false.

printable_list(Term) \rightarrow bool()

Types:
- Term = term()

Returns true if Term is a flat list of printable ISO-latin-1 characters, otherwise it returns false.
lib

Erlang Module

**Warning:**
This module is retained for compatibility. It may disappear without warning in a future release.

Exports

`flush_receive() -> void()`
Flushes the message buffer of the current process.

`error_message(Format, Args) -> ok`
Types:
- `Format = atom() | string() | binary()`
- `Args = [term()]`
Prints error message `Args` in accordance with `Format`. Similar to `io:format/2`, see `io(3)` [page 251].

`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 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:

\[
\text{sendw}(\text{To}, \text{Msg}) -> \\
\text{To} ! \{\text{self()}, \text{Msg}\}, \\
\text{receive} \\
\text{Reply} -> \text{Reply} \\
\text{end}. \\
\]

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.

Whenever an ordering function \( \mathcal{F} \) is expected as argument, it is assumed that the following properties hold of \( \mathcal{F} \) for all \( x, y \) and \( z \):

- if \( x \mathcal{F} y \) and \( y \mathcal{F} x \) then \( x = y \) (\( \mathcal{F} \) is antisymmetric);
- if \( x \mathcal{F} y \) and \( y \mathcal{F} z \) then \( x \mathcal{F} z \) (\( \mathcal{F} \) is transitive);
- \( x \mathcal{F} y \) or \( y \mathcal{F} x \) (\( \mathcal{F} \) is total).

Exports

\( \text{all(Pred, List)} \rightarrow \text{bool}() \)

Types:
- \( \text{Pred} = \text{fun(Elem)} \rightarrow \text{bool}() \)
- \( \text{Elem} = \text{term()} \)
- \( \text{List} = \text{[term()]} \)

Returns true if \( \text{Pred(Elem)} \) returns true for all elements \( \text{Elem} \) in \( \text{List} \), otherwise false.

\( \text{any(Pred, List)} \rightarrow \text{bool}() \)

Types:
- \( \text{Pred} = \text{fun(Elem)} \rightarrow \text{bool}() \)
- \( \text{Elem} = \text{term()} \)
- \( \text{List} = \text{[term()]} \)

Returns true if \( \text{Pred(Elem)} \) returns true for at least one element \( \text{Elem} \) in \( \text{List} \).

\( \text{append(ListOfLists)} \rightarrow \text{List1} \)

Types:
- \( \text{ListOfLists} = \text{[List]} \)
- \( \text{List} = \text{List1} = \text{[term()]} \)
Returns a list in which all the sub-lists of ListOfLists have been appended. For example:

```
> lists:append([[1, 2, 3], [a, b], [4, 5, 6]]).
[1,2,3,a,b,4,5,6]
```

```append(List1, List2) -> List3```

Types:
- `List1 = List2 = List3 = [term()]`

Returns a new list `List3` which is made from the elements of `List1` followed by the elements of `List2`. For example:

```
> lists:append("abc", "def").
"abcdef"
```

```lists:append(A, B) is equivalent to A ++ B.```

```concat(Things) -> string()```

Types:
- `Things = [Thing]`
- `Thing = atom() | integer() | float() | string()`

Concatenates the text representation of the elements of `Things`. The elements of `Things` can be atoms, integers, floats or strings.

```
> lists:concat([doc, '/', file, '.', 3]).
"doc/file.3"
```

```delete(Elem, List1) -> List2```

Types:
- `Elem = term()`
- `List1 = List2 = [term()]`

Returns a copy of `List1` where the first element matching `Elem` is deleted, if there is such an element.

```dropwhile(Pred, List1) -> List2```

Types:
- `Pred = fun(Elem) -> bool()`
- `Elem = term()`
- `List1 = List2 = [term()]`

Drops elements `Elem` from `List1` while `Pred(Elem)` returns `true` and returns the remaining list.

```duplicate(N, Elem) -> List```

Types:
- `N = int()`
- `Elem = term()`
- `List = [term()]`
Returns a list which contains N copies of the term Elem. For example:

```erlang
define do_duplicate() -> List.
define do_duplicate(N, Elem) -> lists:duplicate(N, Elem).
```

```
> lists:duplicate(5, xx).
[xx,xx,xx,xx,xx]
```

`filter(Pred, List1) -> List2`

Types:
- `Pred = fun(Elem) -> bool()`
- `Elem = term()`
- `List1 = List2 = [term()]`

List2 is a list of all elements Elem in List1 for which Pred(Elem) returns true.

`flatlength(DeepList) -> int()`

Types:
- `DeepList = [term() | DeepList]`

Equivalent to `length(flatten(DeepList))`, but more efficient.

`flatmap(Fun, List1) -> List2`

Types:
- `Fun = fun(A) -> [B]`
- `List1 = [A]`
- `List2 = [B]`
- `A = B = term()`

Takes a function from A 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:

```erlang
define do_flatmap() -> List2.
define do_flatmap(Fun, List1) -> append(map(Fun, List1)).
```

Example:

```
> lists:flatmap(fun(X) -> [X,X] end, [a,b,c]).
[a,a,b,b,c,c]
```

`flatten(DeepList) -> List`

Types:
- `DeepList = [term() | DeepList]`
- `List = [term()]`

Returns a flattened version of DeepList.

`flatten(DeepList, Tail) -> List`

Types:
- `DeepList = [term() | DeepList]`
- `Tail = List = [term()]`
Returns a flattened version of `DeepList` with the tail `Tail` appended.

```prolog
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:

```prolog
> 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
```

```prolog
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:

```prolog
> P = fun(A, AccIn) -> io:format(~p ~s, [A]), AccIn end.
> 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`.

```prolog
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.

```prolog
keydelete(Key, N, TupleList1) -> TupleList2
```

Types:
- `Key = term()`
- `N = 1..tuple_size(Tuple)`
keyfind(Key, N, TupleList) -> Tuple | false

Types:
- Key = term()
- N = 1..tuple_size(Tuple)
- TupleList = [Tuple]
- Tuple = tuple()

Searches the list of tuples TupleList for a tuple whose Nth element compares equal to Key. Returns Tuple if such a tuple is found, otherwise false.

Examples:

```erlang
> Fun = fun(Atom) -> atom_to_list(Atom) end.
#Fun<erl_eval.6.10732646>
2> lists:keymap(Fun, 2, [(name,jane,22),(name,lizzie,20),(name,lydia,15)]).
[(name,"jane",22),(name,"lizzie",20),(name,"lydia",15)]
```

keymember(Key, N, TupleList) -> bool()

Types:
- Key = term()
- N = 1..tuple_size(Tuple)
- TupleList = [Tuple]
- Tuple = tuple()

Returns true if there is a tuple in TupleList whose Nth element compares equal to Key, otherwise false.

keymerge(N, TupleList1, TupleList2) -> TupleList3

Types:
- N = 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = TupleList3 = [Tuple]
- Tuple = tuple()
Returns the sorted list formed by merging TupleList1 and TupleList2. The merge is performed on the Nth element of each tuple. Both TupleList1 and TupleList2 must be key-sorted prior to evaluating this function. When two tuples compare equal, the tuple from TupleList1 is picked before the tuple from TupleList2.

keyreplace(Key, N, TupleList1, NewTuple) -> TupleList2

Types:
- Key = term()
- N = 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = [Tuple]
- NewTuple = Tuple = tuple()

Returns a copy of TupleList1 where the first occurrence of a tuple whose Nth element compares equal to Key is replaced with NewTuple, if there is such a tuple T.

keysearch(Key, N, TupleList) -> {value, Tuple} | false

Types:
- Key = term()
- N = 1..tuple_size(Tuple)
- TupleList = [Tuple]
- Tuple = tuple()

Searches the list of tuples TupleList for a tuple whose Nth element compares equal to Key. Returns {value, Tuple} if such a tuple is found, otherwise false.

*Note:*
This function is retained for backward compatibility. The function lists: keyfind/3 (introduced in R13A) is in most cases more convenient.

keysort(N, TupleList1) -> TupleList2

Types:
- N = 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = [Tuple]
- Tuple = tuple()

Returns a list containing the sorted elements of the list TupleList1. Sorting is performed on the Nth element of the tuples.

keystore(Key, N, TupleList1, NewTuple) -> TupleList2

Types:
- Key = term()
- N = 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = [Tuple]
- NewTuple = Tuple = tuple()
Returns a copy of TupleList1 where the first occurrence of a tuple T whose Nth element compares equal to Key is replaced with NewTuple, if there is such a tuple T. If there is no such tuple T a copy of TupleList1 where [NewTuple] has been appended to the end is returned.

keytake(Key, N, TupleList1) -> {value, Tuple, TupleList2} | false

Types:
- Key = term()
- N = 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = [Tuple]
- Tuple = tuple()

Searches the list of tuples TupleList1 for a tuple whose Nth element compares equal to Key. Returns {value, Tuple, TupleList2} if such a tuple is found, otherwise false. TupleList2 is a copy of TupleList1 where the first occurrence of Tuple has been removed.

last(List) -> Last

Types:
- List = [term()], length(List) > 0
- Last = term()

Returns the last element in List.

map(Fun, List1) -> List2

Types:
- Fun = fun(A) -> B
- List1 = [A]
- List2 = [B]
- A = B = term()

Takes a function from A to B, and a list of As and produces a list of Bs by applying the function to every element in the list. This function is used to obtain the return values. The evaluation order is implementation dependent.

mapfoldl(Fun, Acc0, List1) -> {List2, Acc1}

Types:
- Fun = fun(A, AccIn) -> {B, AccOut}
- Acc0 = Acc1 = AccIn = AccOut = term()
- List1 = [A]
- List2 = [B]
- A = B = term()

mapfold combines the operations of map/2 and foldl/3 into one pass. An example, summing the elements in a list and double them at the same time:

> lists:mapfoldl(fun(X, Sum) -> {2*X, X+Sum} end, 0, [{1,2,3,4,5}] + [{2,4,6,8,10},15])
mapfoldr(Fun, Acc0, List1) -> [List2, Acc1]

Types:
- Fun = fun(A, AccIn) -> {B, AccOut}
- Acc0 = Acc1 = AccIn = AccOut = term()
- List1 = [A]
- List2 = [B]
- A = B = term()

mapfold combines the operations of map/2 and foldr/3 into one pass.

max(List) -> Max

Types:
- List = [term()], length(List) > 0
- Max = term()

Returns the first element of List that compares greater than or equal to all other elements of List.

member(Elem, List) -> bool()

Types:
- Elem = term()
- List = [term()]

Returns true if Elem matches some element of List, otherwise false.

merge(ListOfLists) -> List1

Types:
- ListOfLists = [List]
- List = List1 = [term()]

Returns the sorted list formed by merging all the sub-lists of ListOfLists. All sub-lists must be sorted prior to evaluating this function. When two elements compare equal, the element from the sub-list with the lowest position in ListOfLists is picked before the other element.

merge(List1, List2) -> List3

Types:
- List1 = List2 = List3 = [term()]

Returns the sorted list formed by merging List1 and List2. Both List1 and List2 must be sorted prior to evaluating this function. When two elements compare equal, the element from List1 is picked before the element from List2.

merge(Fun, List1, List2) -> List3

Types:
- Fun = fun(A, B) -> bool()
- List1 = [A]
- List2 = [B]
- List3 = [A | B]
- A = B = \text{term}()

Returns the sorted list formed by merging List1 and List2. Both List1 and List2 must be sorted according to the ordering function [page 266] \text{Fun} prior to evaluating this function. \text{Fun}(A, B) should return true if A comes before B in the ordering, false otherwise. When two elements compare equal, the element from List1 is picked before the element from List2.

\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 List1, List2 and List3. All of List1, List2 and List3 must be sorted prior to evaluating this function. When two elements compare equal, the element from List1, if there is such an element, is picked before the other element, otherwise the element from List2 is picked before the element from List3.

\text{min}(\text{List}) \rightarrow \text{Min}

Types:
- \text{List} = \text{[term]}, \text{length(}\text{List}\text{)} > 0
- \text{Min} = \text{[term]}

Returns the first element of List that compares less than or equal to all other elements of List.

\text{nth}(\text{N}, \text{List}) \rightarrow \text{Elem}

Types:
- \text{N} = 1..\text{length(}\text{List}\text{)}
- \text{List} = \text{[term]}
- \text{Elem} = \text{[term]}

Returns the \text{N}th element of List. For example:

\begin{verbatim}
> \text{lists:}\text{nth}(3, [a, b, c, d, e]).
\text{c}
\end{verbatim}

\text{nthtail}(\text{N}, \text{List}) \rightarrow \text{Tail}

Types:
- \text{N} = 0..\text{length(}\text{List}\text{)}
- \text{List} = \text{Tail} = \text{[term]}

Returns the \text{N}th tail of List, that is, the sublist of List starting at \text{N}+1 and continuing up to the end of the list. For example:

\begin{verbatim}
> \text{lists:}\text{nthtail}(3, [a, b, c, d, e]).
[d, e]
> \text{tl(}\text{tl(}\text{tl([a, b, c, d, e])})\text{).}
[d, e]
> \text{lists:}\text{nthtail}(0, [a, b, c, d, e]).
[a, b, c, d, e]
> \text{lists:}\text{nthtail}(5, [a, b, c, d, e]).
[]
\end{verbatim}
**partition**(*Pred*, *List*) -> \{Satisfying, NonSatisfying\)

Types:
- *Pred = fun(Elem) -> bool()*
  - *Elem = term()*
- *List = Satisfying = NonSatisfying = \[term()\]*

Partitions *List* into two lists, where the first list contains all elements for which *Pred(Elem)* returns `true`, and the second list contains all elements for which *Pred(Elem)* returns `false`.

Examples:
- `>` lists:partition(fun(A) -> A rem 2 == 1 end, \[1,2,3,4,5,6,7\]).
  \[1,3,5,7\],\[2,4,6\]
- `>` lists:partition(fun(A) -> is_atom(A) end, \[a,b,1,c,d,2,3,4,e\]).
  \[a,b,c,d,e\],\[1,2,3,4\]

See also `splitwith/2` for a different way to partition a list.

**prefix**(*List1*, *List2*) -> bool()

Types:
- *List1 = List2 = \[term()\]*

Returns `true` if *List1* is a prefix of *List2*, otherwise `false`.

**reverse**(*List1*) -> *List2*

Types:
- *List1 = List2 = \[term()\]*

Returns a list with the top level elements in *List1* in reverse order.

**reverse**(*List1*, *Tail*) -> *List2*

Types:
- *List1 = Tail = List2 = \[term()\]*

Returns a list with the top level elements in *List1* in reverse order, with the tail *Tail* appended. For example:
- `>` lists:reverse([1, 2, 3, 4], \[a, b, c\]).
  \[4,3,2,1,a,b,c\]

**seq**(*From*, *To*) -> *Seq*

**seq**(*From*, *To*, *Incr*) -> *Seq*

Types:
- *From = To = Incr = int()*
  - *Seq = \[int()\]*
Returns a sequence of integers which starts with From and contains the successive results of adding Incr to the previous element, until To has been reached or passed (in the latter case, To is not an element of the sequence). Incr defaults to 1.

Failure: If To<From-Incr and Incr is positive, or if To>From-Incr and Incr is negative, or if Incr==0 and From=To.

The following equalities hold for all sequences:

\[
\text{length(lists:seq(From, To))} = \text{To-From+1}
\]
\[
\text{length(lists:seq(From, To, Incr))} = (\text{To-From+Incr}) \div \text{Incr}
\]

Examples:

\[
> \text{lists:seq(1, 10)}.
[1,2,3,4,5,6,7,8,9,10]
\]
\[
> \text{lists:seq(1, 20, 3)}.
[1,4,7,10,13,16,19]
\]
\[
> \text{lists:seq(1, 0, 1)}.
[]
\]
\[
> \text{lists:seq(10, 6, 4)}.
[]
\]
\[
> \text{lists:seq(1, 1, 0)}.
[1]
\]

\[
\text{sort(List1)} -> \text{List2}
\]

Types:
- List1 = List2 = [term()]

Returns a list containing the sorted elements of List1.

\[
\text{sort(Fun, List1)} -> \text{List2}
\]

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 [page 266] Fun. Fun(A, B) should return true if A comes before B in the ordering, false otherwise.

\[
\text{split(N, List1)} -> \{\text{List2, List3}\}
\]

Types:
- N = 0..\text{length(List1)}
- List1 = List2 = List3 = [term()]

Splits List1 into List2 and List3. List2 contains the first N elements and List3 the rest of the elements (the Nth tail).

\[
\text{splitwith(Pred, List)} -> \{\text{List1, List2}\}
\]

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:

```erlang
splitwith(Pred, List) ->
    {takewhile(Pred, List), dropwhile(Pred, List)}.
```

Examples:

```erlang
> lists:splitwith(fun(A) -> A rem 2 == 1 end, [1,2,3,4,5,6,7]).
[[1],[2,3,4,5,6,7]]
> lists:splitwith(fun(A) -> is_atom(A) end, [a,b,1,c,d,2,3,4,e]).
[[a,b],[1,c,d,2,3,4,e]]
```

See also partition/2 for a different way to partition a list.

sublist(List1, Len) -> List2

Types:

• List1 = List2 = [term()]
• Len = int()

Returns the sub-list of List1 starting at position 1 and with (max) Len elements. It is not an error for Len to exceed the length of the list – in that case the whole list is returned.

sublist(List1, Start, Len) -> List2

Types:

• List1 = List2 = [term()]
• Start = 1..(length(List1)+1)
• Len = int()

Returns the sub-list of List1 starting at Start and with (max) Len elements. It is not an error for Start+Len to exceed the length of the list.

```erlang
> lists:sublist([1,2,3,4], 2, 2).
[2,3]
> lists:sublist([1,2,3,4], 2, 5).
[2,3,4]
> lists:sublist([1,2,3,4], 5, 2).
[]
```

subtract(List1, List2) -> List3

Types:

• List1 = List2 = List3 = [term()]

Returns a new list List3 which is a copy of List1, subjected to the following procedure: for each element in List2, its first occurrence in List1 is deleted. For example:

```erlang
> lists:subtract("123212", "212").
"312".
```
lists:subtract(A, B) is equivalent to A -- B.

**Warning:**
The complexity of lists:subtract(A, B) is proportional to length(A)*length(B), meaning that it will be very slow if both A and B are long lists. (Using ordered lists and ordsets:subtract/2 [page 303] is a much better choice if both lists are long.)

- **suffix(List1, List2)**
  - bool()
  - Returns true if List1 is a suffix of List2, otherwise false.

- **sum(List)**
  - number()
  - Types:
    - List = [number()]
  - Returns the sum of the elements in List.

- **takewhile(Pred, List1)**
  - List2
  - Types:
    - Pred = fun(Elem) -> bool()
    - Elem = term()
    - List1 = List2 = [term()]
  - Takes elements Elem from List1 while Pred(Elem) returns true, that is, the function returns the longest prefix of the list for which all elements satisfy the predicate.

- **ukeymerge(N, TupleList1, TupleList2)**
  - TupleList3
  - Types:
    - N = 1..tuple
    - TupleList1 = TupleList2 = TupleList3 = [Tuple]
    - Tuple = tuple()
  - Returns the sorted list formed by merging TupleList1 and TupleList2. The merge is performed on the Nth element of each tuple. Both TupleList1 and TupleList2 must be key-sorted without duplicates prior to evaluating this function. When two tuples compare equal, the tuple from TupleList1 is picked and the one from TupleList2 deleted.

- **ukeysort(N, TupleList1)**
  - TupleList2
  - Types:
    - N = 1..tuple
    - TupleList1 = [Tuple]
    - Tuple = tuple()
  - Returns a list containing the sorted elements of the list TupleList1 where all but the first tuple of the tuples comparing equal have been deleted. Sorting is performed on the 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. All sub-lists must be sorted and contain no duplicates prior to evaluating this function. When two elements compare equal, the element from the sub-list with the lowest position in ListOfLists is picked and the other one deleted.

umerge(List1, List2) -> List3

Types:
- List1 = List2 = List3 = [term()]

Returns the sorted list formed by merging List1 and List2. Both List1 and List2 must be sorted and contain no duplicates prior to evaluating this function. When two elements compare equal, the element from List1 is picked and the one from List2 deleted.

umerge(Fun, List1, List2) -> List3

Types:
- Fun = fun(A, B) -> bool()
- List1 = [A]
- List2 = [B]
- List3 = [A | B]
- A = B = term()

Returns the sorted list formed by merging List1 and List2. Both List1 and List2 must be sorted according to the ordering function [page 266] Fun and contain no duplicates prior to evaluating this function. Fun(A, B) should return true if A comes before B in the ordering, false otherwise. When two elements compare equal, the element from List1 is picked and the one from List2 deleted.

umerge3(List1, List2, List3) -> List4

Types:
- List1 = List2 = List3 = List4 = [term()]

Returns the sorted list formed by merging List1, List2 and List3. All of List1, List2 and List3 must be sorted and contain no duplicates prior to evaluating this function. When two elements compare equal, the element from List1 is picked if there is such an element, otherwise the element from List2 is picked, and the other one deleted.

unzip(List1) -> {List2, List3}

Types:
- List1 = [{X, Y}]
- List2 = [X]
- List3 = [Y]
- X = Y = term()
"Unzips" a list of two-tuples into two lists, where the first list contains the first element of each tuple, and the second list contains the second element of each tuple.

unzip3(List1) -> {List2, List3, List4}

Types:
- List1 = [[X, Y, Z]]
- List2 = [X]
- List3 = [Y]
- List4 = [Z]
- X = Y = Z = term()

"Unzips" a list of three-tuples into three lists, where the first list contains the first element of each tuple, the second list contains the second element of each tuple, and the third list contains the third element of each tuple.

usort(List1) -> List2

Types:
- List1 = List2 = [term()]

Returns a list containing the sorted elements of List1 where all but the first element of the elements comparing equal have been deleted.

usort(Fun, List1) -> List2

Types:
- Fun = fun(Elem1, Elem2) -> bool()
- Elem1 = Elem2 = term()
- List1 = List2 = [term()]

Returns a list which contains the sorted elements of List1 where all but the first element of the elements comparing equal according to the ordering function [page 266] Fun have been deleted. Fun(A, B) should return true if A comes before B in the ordering, false otherwise.

zip(List1, List2) -> List3

Types:
- List1 = [X]
- List2 = [Y]
- List3 = [[X, Y]]
- X = Y = term()

"Zips" two lists of equal length into one list of two-tuples, where the first element of each tuple is taken from the first list and the second element is taken from corresponding element in the second list.

zip3(List1, List2, List3) -> List4

Types:
- List1 = [X]
- List2 = [Y]
- List3 = [Z]
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 \( \text{Combine}(X, Y) \).

\[
\text{zipwith}(\text{fun}(X, Y) \rightarrow X + Y \text{ end}, \text{List1}, \text{List2}) \text{ is equivalent to } \text{zip} (\text{List1}, \text{List2}).
\]

Example:
> lists:zipwith(fun(X, Y) -> X + Y end, [1, 2, 3], [4, 5, 6]).
> [5, 7, 9]

zipwith3(Combine, List1, List2, List3) -> List4

Types:
- Combine = fun(X, Y, Z) -> T
- List1 = [X]
- List2 = [Y]
- List3 = [Z]
- List4 = [T]
- X = Y = Z = T = term()

Combine the elements of three lists of equal length into one list. For each triple \( X, Y, Z \) of list elements from the three lists, the element in the result list will be \( \text{Combine}(X, Y, Z) \).

\[
\text{zipwith3}(\text{fun}(X, Y, Z) \rightarrow X + Y + Z \text{ end}, \text{List1}, \text{List2}, \text{List3}) \text{ is equivalent to } \text{zip3} (\text{List1}, \text{List2}, \text{List3}).
\]

Examples:
> lists:zipwith3(fun(X, Y, Z) -> X + Y + Z end, [1, 2, 3], [4, 5, 6], [7, 8, 9]).
> [12, 15, 18]
> lists:zipwith3(fun(X, Y, Z) -> [X, Y, Z] end, [a, b, c], [x, y, z], [1, 2, 3]).
> [[a, x, 1], [b, y, 2], [c, z, 3]]
The log_mf_h is a gen_event handler module which can be installed in any gen_event process. It logs onto disk all events which are sent to an event manager. Each event is written as a binary which makes the logging very fast. However, a tool such as the Report Browser (rb) must be used in order to read the files. The events are written to multiple files. When all files have been used, the first one is re-used and overwritten. The directory location, the number of files, and the size of each file are configurable. The directory will include one file called index, and report files 1, 2, ... .

Exports

init(Dir, MaxBytes, MaxFiles)
init(Dir, MaxBytes, MaxFiles, Pred) -> Args

Types:
  • Dir = string()
  • MaxBytes = integer()
  • MaxFiles = 0 < integer() < 256
  • Pred = fun(Event) -> boolean()
  • Event = term()
  • Args = args()

Initiates the event handler. This function returns Args, which should be used in a call to gen_event:add_handler(EventMgr, log_mf_h, Args).

Dir specifies which directory to use for the log files. MaxBytes specifies the size of each individual file. MaxFiles specifies how many files are used. Pred is a predicate function used to filter the events. If no predicate function is specified, all events are logged.

See Also

gen_event(3) [page 215], rb(3)
math

Erlang Module

This module provides an interface to a number of mathematical functions.

**Note:**
Not all functions are implemented on all platforms. In particular, the \texttt{erf/1} and \texttt{erfc/1} functions are not implemented on Windows.

Exports

\texttt{pi()} \rightarrow \texttt{float()}

A useful number.

\texttt{sin(X)}
\texttt{cos(X)}
\texttt{tan(X)}
\texttt{asin(X)}
\texttt{acos(X)}
\texttt{atan(X)}
\texttt{atan2(Y, X)}
\texttt{sinh(X)}
\texttt{cosh(X)}
\texttt{tanh(X)}
\texttt{asinh(X)}
\texttt{acosh(X)}
\texttt{atanh(X)}
\texttt{exp(X)}
\texttt{log(X)}
\texttt{log10(X)}
\texttt{pow(X, Y)}
\texttt{sqrt(X)}

Types:
- \texttt{X = Y = number()}

A collection of math functions which return floats. Arguments are numbers.
erf(X) -> float()

Types:
- X = number()

Returns the error function of X, where

\[ \text{erf}(X) = \frac{2}{\sqrt{\pi}} \int_0^X e^{-t^2} \, dt \].

erfc(X) -> float()

Types:
- X = number()

\[ \text{erfc}(X) = 1.0 - \text{erf}(X) \], computed by methods that avoid cancellation for large X.

Bugs

As these are the C library, the bugs are the same.
ms_transform

Erlang Module

This module implements the parse_transform that makes calls to ets and dbg:fun2ms/1 translate into literal match specifications. It also implements the back end for the same functions when called from the Erlang shell.

The translations from fun's to match_specs is accessed through the two "pseudo functions" ets:fun2ms/1 and dbg:fun2ms/1.

Actually this introduction is more or less an introduction to the whole concept of match specifications. Since everyone trying to use ets:select or dbg seems to end up reading this page, it seems in good place to explain a little more than just what this module does.

There are some caveats one should be aware of, please read through the whole manual page if it’s the first time you’re using the transformations.

Match specifications are used more or less as filters. They resemble usual Erlang matching in a list comprehension or in a fun used in conjunction with lists:foldl etc.

The syntax of pure match specifications is somewhat awkward though, as they are made up purely by Erlang terms and there is no syntax in the language to make the match specifications more readable.

As the match specifications execution and structure is quite like that of a fun, it would for most programmers be more straight forward to simply write it using the familiar fun syntax and having that translated into a match specification automatically. Of course a real fun is more powerful than the match specifications allow, but bearing the match specifications in mind, and what they can do, it's still more convenient to write it all as a fun. This module contains the code that simply translates the fun syntax into match_spec terms.

Let’s start with an ets example. Using ets:select and a match specification, one can filter out rows of a table and construct a list of tuples containing relevant parts of the data in these rows. Of course one could use ets:foldl instead, but the select call is far more efficient. Without the translation, one has to struggle with writing match specifications terms to accommodate this, or one has to resort to the less powerful ets:match(object) calls, or simply give up and use the more inefficient method of ets:foldl. Using the ets:fun2ms transformation, an ets:select call is at least as easy to write as any of the alternatives.

As an example, consider a simple table of employees:

-record(emp, {empno, %Employee number as a string, the key
  surname, %Surname of the employee
givenname, %Given name of employee
dept, %Department one of {dev,sales,prod,adm}
empyear}). %Year the employee was employed

We create the table using:

ets:new(emp_tab,[[keypos,#emp.empno],named_table,ordered_set]).
Let's also fill it with some randomly chosen data for the examples:

```
[{emp,"011103","Black","Alfred",sales,2000},
 {emp,"041231","Doe","John",prod,2001},
 {emp,"052341","Smith","John",dev,1997},
 {emp,"076324","Smith","Ella",sales,1995},
 {emp,"122334","Weston","Anna",prod,2002},
 {emp,"535216","Chalker","Samuel",adm,1998},
 {emp,"789789","Harrysson","Joe",adm,1996},
 {emp,"963721","Scott","Juliana",dev,2003},
 {emp,"989891","Brown","Gabriel",prod,1999}]
```

Now, the amount of data in the table is of course too small to justify complicated ets searches, but on real tables, using `select` to get exactly the data you want will increase efficiency remarkably.

Let's say for example that we'd want the employee numbers of everyone in the sales department. One might use `ets:match` in such a situation:

```
1> ets:match(emp_tab, [{_, '$1', _, _, sales, _}]).
["011103"],["076324"]
```

Even though `ets:match` does not require a full match specification, but a simpler type, it's still somewhat unreadable, and one has little control over the returned result, it's always a list of lists. OK, one might use `ets:foldl` or `ets:foldr` instead:

```
ets:foldr(fun(#emp{empno = E, dept = sales},Acc) -> [E | Acc];
   (_,Acc) -> Acc
end,
[],
emp_tab).
```

Running that would result in `"011103","076324"`, which at least gets rid of the extra lists. The function is also quite straightforward, so the only problem is that all the data from the table has to be transferred from the table to the calling process for filtering.

That's inefficient compared to the `ets:match` call where the filtering can be done "inside" the emulator and only the result is transferred to the process. Remember that ets tables are all about efficiency, if it wasn't for efficiency all of ets could be implemented in Erlang, as a process receiving requests and sending answers back. One uses ets because one wants performance, and therefore one wouldn't want all of the table transferred to the process for filtering. OK, let's look at a pure `ets:select` call that does what the `ets:foldr` does:

```
ets:select(emp_tab,[#{empno = '$1', dept = sales, _='_'},[],['$1']]).
```

Even though the record syntax is used, it's still somewhat hard to read and even harder to write. The first element of the tuple, `#emp{empno = '$1', dept = sales, _='_'}`, tells what to match, elements not matching this will not be returned at all, as in the `ets:match` example. The second element, the empty list is a list of guard expressions, which we need none, and the third element is the list of expressions constructing the return value (in ets this almost always is a list containing one single term). In our case `$1` is bound to the employee number in the head (first element of tuple), and hence it is the employee number that is returned. The result is `"011103","076324"`, just as in the `ets:foldr` example, but the result is retrieved much more efficiently in terms of execution speed and memory consumption.

We have one efficient but hardly readable way of doing it and one inefficient but fairly readable (at least to the skilled Erlang programmer) way of doing it. With the use of
ets:fun2ms, one could have something that is as efficient as possible but still is written as a filter using the fun syntax:

```
-include_lib("stdlib/include/ms_transform.hrl").

% ...  

ets:select(emp_tab, ets:fun2ms(
    fun(#emp{empno = E, dept = sales}) ->
        E
    end)).
```

This may not be the shortest of the expressions, but it requires no special knowledge of match specifications to read. The fun's head should simply match what you want to filter out and the body returns what you want returned. As long as the fun can be kept within the limits of the match specifications, there is no need to transfer all data of the table to the process for filtering as in the ets:foldr example. In fact it's even easier to read then the ets:foldr example, as the select call in itself discards anything that doesn't match, while the fun of the foldr call needs to handle both the elements matching and the ones not matching.

It's worth noting in the above ets:fun2ms example that one needs to include ms_transform.hrl in the source code, as this is what triggers the parse transformation of the ets:fun2ms call to a valid match specification. This also implies that the transformation is done at compile time (except when called from the shell of course) and therefore will take no resources at all in runtime. So although you use the more intuitive fun syntax, it gets as efficient in runtime as writing match specifications by hand.

Let's look at some more ets examples. Let's say one wants to get all the employee numbers of any employee hired before the year 2000. Using ets:match isn't an alternative here as relational operators cannot be expressed there. Once again, an ets:foldr could do it (slowly, but correct):

```
ets:foldr(fun(#emp{empno = E, empyear = Y},Acc) when Y < 2000 -> [E | Acc];
          (_,Acc) -> Acc
       end,
       [],
       emp_tab).
```

The result will be `"052341", "076324", "535216", "789789", "989891"]`, as expected. Now the equivalent expression using a handwritten match specification would look something like this:

```
ets:select(emp_tab, [{#emp{empno = '$1', empyear = '$2', _='_'},
                      [{'<', '$2', 2000}],
                      ['$1']}).
```

This gives the same result, the `[['', '$2', 2000]]` is in the guard part and therefore discards anything that does not have a empyear (bound to '$2' in the head) less than 2000, just as the guard in the foldl example. Let's jump on to writing it using ets:fun2ms
% Obviously readability is gained by using the parse transformation.
I'll show some more examples without the tiresome comparing-to-alternatives stuff.
Let's say we'd want the whole object matching instead of only one element. We could
of course assign a variable to every part of the record and build it up once again in the
body of the fun, but it's easier to do like this:

```
ets:select(emp_tab, ets:fun2ms(
  fun(Obj = #emp{empno = E, empyear = Y})
    when Y < 2000 ->
      Obj
    end)).
```

Just as in ordinary Erlang matching, you can bind a variable to the whole matched
object using a “match in then match”, i.e. a =. Unfortunately this is not general in fun’s
translated to match specifications, only on the “top level”, i.e. matching the whole object
arriving to be matched into a separate variable, is it allowed. For the one’s used to
writing match specifications by hand, I’ll have to mention that the variable A will
simply be translated into ‘$’. It’s not general, but it has very common usage, why it is
handled as a special, but useful, case. If this bothers you, the pseudo function object
also returns the whole matched object, see the part about caveats and limitations below.
Let’s do something in the fun’s body too: Let’s say that someone realizes that there are
a few people having an employee number beginning with a zero (0), which shouldn’t be
allowed. All those should have their numbers changed to begin with a one (1) instead
and one wants the list [[<Old empno>,<New empno>]] created:

```
ets:select(emp_tab, ets:fun2ms(
  fun(#emp{empno = [0 | Rest] }) ->
    {{0|Rest},[1|Rest]}
  end)).
```

As a matter of fact, this query hit’s the feature of partially bound keys in the table type
ordered_set, so that not the whole table need be searched, only the part of the table
containing keys beginning with 0 is in fact looked into.
The fun of course can have several clauses, so that if one could do the following: For
each employee, if he or she is hired prior to 1997, return the tuple {inventory,
<employee number>}, for each hired 1997 or later, but before 2001, return {rookie,
<employee number>}, for all others return {newbie, <employee number>}. All
except for the ones named Smith as they would be affronted by anything other than the
tag guru and that is also what’s returned for their numbers; {guru, <employee
number>}:

```
ets:select(emp_tab, ets:fun2ms(
  fun(#emp{empno = E, surname = "Smith" }) ->
    {guru,E};
    (#emp{empno = E, empyear = Y}) when Y < 1997 ->
      {inventory, E};
```

```
% ...%
The result will be:

```erlang
[{rookie,"011103"},
 {rookie,"041231"},
 {guru,"052341"},
 {guru,"076324"},
 {newbie,"122334"},
 {rookie,"535216"},
 {inventory,"789789"},
 {newbie,"963721"},
 {rookie,"989891"}]
```

and so the Smith's will be happy...

So, what more can you do? Well, the simple answer would be: look in the documentation of match specifications in ERTS users guide. However let's briefly go through the most useful “built in functions” that you can use when the fun is to be translated into a match specification by ets:fun2ms (it’s worth mentioning, although it might be obvious to some, that calling other functions than the one’s allowed in match specifications cannot be done. No “usual” Erlang code can be executed by the fun being translated by fun2ms, the fun is after all limited exactly to the power of the match specifications, which is unfortunate, but the price one has to pay for the execution speed of an ets:select compared to ets:foldl/foldr).

The head of the fun is obviously a head matching (or mismatching) one parameter, one object of the table we select from. The object is always a single variable (can be _ or a tuple, as that’s what’s in ets, dets and mnesia tables (the match specification returned by ets:fun2ms can of course be used with dets:select and mnesia:select as well as with ets:select). The use of _ in the head is allowed (and encouraged) on the top level.

The guard section can contain any guard expression of Erlang. Even the “old” type test are allowed on the toplevel of the guard (is_integer(X) instead of is_integer(X)). As the new type tests (the is_tests) are in practice just guard bif’s they can also be called from within the body of the fun, but so they can in ordinary Erlang code. Also arithmetics is allowed, as well as ordinary guard bif’s. Here’s a list of bif’s and expressions:

- The type tests: is_atom, is_constant, is_float, is_integer, is_list, is_number, is_pid, is_port, is_reference, is_tuple, is_binary, is_function, is_record
- The boolean operators: not, and, or, andalso, orelse
- The relational operators: >, >=, <, <=, =:=, ==, /=, /=
- Arithmetics: +, -, *, div, rem
- Bitwise operators: band, bor, bxor, bnot, bsl, bsr
- The guard bif’s: abs, element, hd, length, node, round, size, tl, trunc, self
- The obsolete type test (only in guards): atom, constant, float, integer, list, number, pid, port, reference, tuple, binary, function, record
Contrary to the fact with “handwritten” match specifications, the is_record guard works as in ordinary Erlang code.

Semicolons (;) in guards are allowed, the result will be (as expected) one “match_spec-clause” for each semicolon-separated part of the guard. The semantics being identical to the Erlang semantics.

The body of the fun is used to construct the resulting value. When selecting from tables one usually just construct a suiting term here, using ordinary Erlang term construction, like tuple parentheses, list brackets and variables matched out in the head, possibly in conjunction with the occasional constant. Whatever expressions are allowed in guards are also allowed here, but there are no special functions except object and bindings (see further down), which returns the whole matched object and all known variable bindings respectively.

The dbg variants of match specifications have an imperative approach to the match specification body, the ets dialect hasn’t. The fun body for ets:fun2ms returns the result without side effects, and as matching (=) in the body of the match specifications is not allowed (for performance reasons) the only thing left, more or less, is term construction...

Let’s move on to the dbg dialect, the slightly different match specifications translated by dbg:fun2ms.

The same reasons for using the parse transformation applies to dbg, maybe even more so as filtering using Erlang code is simply not a good idea when tracing (except afterwards, if you trace to file). The concept is similar to that of ets:fun2ms except that you usually use it directly from the shell (which can also be done with ets:fun2ms).

Let’s manufacture a toy module to trace on

-module(toy).
-export([start/1, store/2, retrieve/1]).

start(Args) ->
  toy_table = ets:new(toy_table,Args).
store(Key, Value) ->
  ets:insert(toy_table,{Key,Value}).
retrieve(Key) ->
  [{Key, Value}] = ets:lookup(toy_table,Key),
  Value.

During model testing, the first test bails out with a {badmatch,16} in {toy,start,1}, why?

We suspect the ets call, as we match hard on the return value, but want only the particular new call with toy_table as first parameter. So we start a default tracer on the node:

1> dbg:tracer().
{ok,<0.88.0>}

And so we turn on call tracing for all processes, we are going to make a pretty restrictive trace pattern, so there’s no need to call trace only a few processes (it usually isn’t):

2> dbg:p(all,call).
{ok,[[matched,nonode@nohost,25]]}
It's time to specify the filter. We want to view calls that resemble `ets:new(toy_table,<something>):`

3> dbg:tp(ets,new,dbg:fun2ms(fun([toy_table,_] -> true end))).
   {ok,[[matched,nonode@nohost,1],[saved,1]]}

As can be seen, the fun's used with `dbg:fun2ms` takes a single list as parameter instead of a single tuple. The list matches a list of the parameters to the traced function. A single variable may also be used of course. The body of the fun expresses in a more imperative way actions to be taken if the fun head (and the guards) matches. I return `true` here, but it's only because the body of a fun cannot be empty, the return value will be discarded.

When we run the test of our module now, we get the following trace output:

(0.86.0) call ets:new(toy_table,[ordered_set])

Let's play we haven't spotted the problem yet, and want to see what `ets:new` returns. We do a slightly different trace pattern:

4> dbg:tp(ets,new,dbg:fun2ms(fun([toy_table,_] -> return_trace() end))).

Resulting in the following trace output when we run the test:

(0.86.0) call ets:new(toy_table,[ordered_set])
(0.86.0) returned from ets:new/2 -> 24

The call to `return_trace`, makes a trace message appear when the function returns. It applies only to the specific function call triggering the match specification (and matching the head/guards of the match specification). This is the by far the most common call in the body of a `dbg` match specification.

As the test now fails with `{badmatch,24}`, it's obvious that the badmatch is because the atom `toy_table` does not match the number returned for an unnamed table. So we spotted the problem, the table should be named and the arguments supplied by our test program does not include `named_table`. We rewrite the start function to:

```erlang
call(Args) ->
   toy_table = ets:new(toy_table,[named_table |Args]).
```

And with the same tracing turned on, we get the following trace output:

(0.86.0) call ets:new(toy_table,[named_table,ordered_set])
(0.86.0) returned from ets:new/2 -> toy_table

Very well. Let's say the module now passes all testing and goes into the system. After a while someone realizes that the table `toy_table` grows while the system is running and that for some reason there are a lot of elements with atom's as keys. You had expected only integer keys and so does the rest of the system. Well, obviously not all of the system. You turn on call tracing and try to see calls to your module with an atom as the key:

1> dbg:tracer().
   {ok,<0.88.0>}
2> dbg:p(all,call).
   {ok,[[matched,nonode@nohost,25]]}
3> dbg:tpl(toy,store,dbg:fun2ms(fun([_,_] when is_atom(A) -> true end))).
   {ok,[[matched,nonode@nohost,1],[saved,1]]}
We use `dbg:tpl` here to make sure to catch local calls (let’s say the module has grown since the smaller version and we’re not sure this inserting of atoms is not done locally...). When in doubt always use local call tracing.

Let’s say nothing happens when we trace in this way. Our function is never called with these parameters. We make the conclusion that someone else (some other module) is doing it and we realize that we must trace on `ets:insert` and want to see the calling function. The calling function may be retrieved using the match specification function `caller` and to get it into the trace message, one has to use the match spec function `message`. The filter call looks like this (looking for calls to `ets:insert`):

```erlang
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:

```erlang
(<0.86.0>) call ets:insert(toy_table,\([\text{garbage,can}]\)) \{(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

\[ is_list(X) \rightarrow ['is_list', '$1'] \] ('$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

\[ \{ \{ '$1', '$2' \}, \{ \text{'is_atom', '$1'} \}, \{ $2 \} \} \]

Variables that are not appearing in the head are imported from the environment and made into match_spec const expressions. Example from the shell:

1> X = 25.
25
2> ets:fun2ms(fun([A,B]) when A > X -> B end).
\[ \{ \{ '$1', '$2' \}, \{ \text{'}>', '$1', \{ \text{'}const, 25\} \}, \{ $2 \} \} \]

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).
\[ \{ \{ '$1', \{ '$2', '$3' \}, \{ '>' , '$1', \{ '$2' \} \}, \{ $2 \} \} \]
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]] = D) when A > B -> D = [B|C], D end).
Error: fun with body matching ('=' in body) is illegal as match_spec {error,transform_error}

All variables are bound in the head of a match_spec, so the translator can not allow multiple bindings. The special case when matching is done on the top level makes the variable bind to '$_' in the resulting match_spec, it is to allow a more natural access to the whole matched object. The pseudo function object() could be used instead, see below. The following expressions are translated equally:

ets:fun2ms(fun({a,_} = A) -> A end).
ets:fun2ms(fun({a,_}) -> object() end).

The special match_spec variables '$.' and '$*' can be accessed through the pseudo functions object() (for '$.') and bindings() (for '$*'). as an example, one could translate the following ets:match_object/2 call to a ets:select call:

ets:match_object(Table, \{ '$1', test, '$2' \}).

...is the same as...

ets:select(Table, ets:fun2ms(fun([A,test,B]) -> object() end)).

(This was just an example, in this simple case the former expression is probably preferable in terms of readability). The ets:select/2 call will conceptually look like this in the resulting code:
ets:select(Table, [[{'$1', 'test', '$2'}, [], ['$_']])

Matching on the top level of the fun head might feel like a more natural way to access '$_' see above.

- Term constructions/literals are translated as much as is needed to get them into valid match specs, so that tuples are made into match_spec tuple constructions (a one element tuple containing the tuple) and constant expressions are used when importing variables from the environment. Records are also translated into plain tuple constructions, calls to element etc. The guard test is_record/2 is translated into match_spec code using the three parameter version that's built into match_specs so that is_record(A,t) is translated into {is_record,'$1',t,5} given that the record size of record type t is 5.

- Language constructions like case, if, catch etc that are not present in match specs are not allowed.

- If the header file ms_transform.hrl is not included, the fun won't be translated, which may result in a runtime error (depending on if the fun is valid in a pure Erlang context). Be absolutely sure that the header is included when using ets and dbg:fun2ms/1 in compiled code.

- If the pseudo function triggering the translation is ets:fun2ms/1, the fun's head must contain a single variable or a single tuple. If the pseudo function is dbg:fun2ms/1 the fun's head must contain a single variable or a single list.

The translation from fun's to match_specs is done at compile time, so runtime performance is not affected by using these pseudo functions. The compile time might be somewhat longer though.

For more information about match_specs please read about them in ERTS users guide.

Exports

parse_transform(Forms, Options) -> Forms

Types:
- Forms = Erlang abstract code format, see the erl_parse module description
- Options = Option list, required but not used

Implements the actual transformation at compile time. This function is called by the compiler to do the source code transformation if and when the ms_transform.hrl header file is included in your source code. See the ets and dbg:fun2ms/1 function manual pages for documentation on how to use this parse_transform, see the match_spec chapter in ERTS users guide for a description of match specifications.

transform_from_shell(Dialect, Clauses, BoundEnvironment) -> term()

Types:
- Dialect = ets | dbg
- Clauses = Erlang abstract form for a single fun
- BoundEnvironment = [{atom(), term()}, ...], list of variable bindings in the shell environment
Implements the actual transformation when the fun2ms functions are called from the shell. In this case the abstract form is for one single fun (parsed by the Erlang shell), and all imported variables should be in the key-value list passed as BoundEnvironment. The result is a term, normalized, i.e. not in abstract format.

format_error(Errcode) -> ErrMessage

Types:
- Errcode = term()
- ErrMessage = string()

Takes an error code returned by one of the other functions in the module and creates a textual description of the error. Fairly uninteresting function actually.
orddict

Erlang Module

Orddict implements a Key-Value dictionary. An orddict is a representation of a
dictionary, where a list of pairs is used to store the keys and values. The list is ordered
after the keys.

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

DATA TYPES

ordered_dictionary()
  as returned by new/0

Exports

append(Key, Value, Orddict1) -> Orddict2
  Types:
  • Key = Value = term()
  • Orddict1 = Orddict2 = ordered_dictionary()
  This function appends a new Value to the current list of values associated with Key. An
  exception is generated if the initial value associated with Key is not a list of values.

append_list(Key, ValList, Orddict1) -> Orddict2
  Types:
  • ValList = [Value]
  • Key = Value = term()
  • Orddict1 = Orddict2 = ordered_dictionary()
  This function appends a list of values ValList to the current list of values associated
  with Key. An exception is generated if the initial value associated with Key is not a list
  of values.

erase(Key, Orddict1) -> Orddict2
  Types:
  • Key = term()
  • Orddict1 = Orddict2 = ordered_dictionary()
This function erases all items with a given key from a dictionary.

```erlang
fetch(Key, Orddict) -> Value
```

**Types:**
- `Key` = `Value` = `term()`
- `Orddict` = `ordered_dictionary()`

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

```erlang
fetch_keys(Orddict) -> Keys
```

**Types:**
- `Orddict` = `ordered_dictionary()`
- `Keys` = `[term()]`

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

```erlang
filter(Pred, Orddict1) -> Orddict2
```

**Types:**
- `Pred` = `fun(Key, Value) -> bool()`
- `Key` = `Value` = `term()`
- `Orddict1` = `Orddict2` = `ordered_dictionary()`

`Orddict2` is a dictionary of all keys and values in `Orddict1` for which `Pred(Key, Value)` is true.

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

**Types:**
- `Key` = `Value` = `term()`
- `Orddict` = `ordered_dictionary()`

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

```erlang
fold(Fun, Acc0, Orddict) -> Acc1
```

**Types:**
- `Fun` = `fun(Key, Value, AccIn) -> AccOut`
- `Key` = `Value` = `term()`
- `Acc0` = `Acc1` = `AccIn` = `AccOut` = `term()`
- `Orddict` = `ordered_dictionary()`

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

```erlang
from_list(List) -> Orddict
```

**Types:**
- `List` = `[{Key, Value}]`
**orddict**

- `O r d d i c t = o r d e r e d _ d i c t i o n a r y ()`
  
  This function converts the key/value list `List` to a dictionary.

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

  Types:
  - `Key = term ()`
  - `Orddict = ordered_dictionary ()`

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

`map(Fun, Orddict1) -> Orddict2`

  Types:
  - `Fun = fun(Key, Value1) -> Value2`
  - `Key = Value1 = Value2 = term ()`
  - `Orddict1 = Orddict2 = ordered_dictionary ()`

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

`merge(Fun, Orddict1, Orddict2) -> Orddict3`

  Types:
  - `Fun = fun(Key, Value1, Value2) -> Value`
  - `Key = Value1 = Value2 = Value3 = term ()`
  - `Orddict1 = Orddict2 = Orddict3 = ordered_dictionary ()`

  `merge` merges two dictionaries, `Orddict1` and `Orddict2`, to create a new dictionary. All the `Key`-`Value` pairs from both dictionaries are included in the new dictionary. If a key occurs in both dictionaries then `Fun` is called with the key and both values to return a new value. `merge` could be defined as:

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

  but is faster.

`new () -> ordered_dictionary ()`

  This function creates a new dictionary.

`size(Orddict) -> int ()`

  Types:
  - `Orddict = ordered_dictionary ()`

  Returns the number of elements in an `Orddict`.

`store(Key, Value, Orddict1) -> Orddict2`

  Types:
  - `Key = Value = term ()`
to_list(Orddict) -> List

Types:
- Orddict = ordered_dictionary()
- List = [{Key, Value}]

This function converts the dictionary to a list representation.

update(Key, Fun, Orddict1) -> Orddict2

Types:
- Key = term()
- Fun = fun(Value1) -> Value2
- Value1 = Value2 = term()
- Orddict1 = Orddict2 = ordered_dictionary()

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

update(Key, Fun, Initial, Orddict1) -> Orddict2

Types:
- Key = Initial = term()
- Fun = fun(Value1) -> Value2
- Value1 = Value2 = term()
- Orddict1 = Orddict2 = ordered_dictionary()

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

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

update_counter(Key, Increment, Orddict1) -> Orddict2

Types:
- Key = term()
- Increment = number()
- Orddict1 = Orddict2 = ordered_dictionary()

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

This could be defined as:

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

but is faster.
Notes

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

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

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

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

See Also

dict(3) [page 120], gb_trees(3) [page 210]
ordsets

Erlang Module

Sets are collections of elements with no duplicate elements. An ordset is a representation of a set, where an ordered list is used to store the elements of the set. An ordered list is more efficient than an unordered list.

This module provides exactly the same interface as the module sets but with a defined representation. One difference is that while sets considers two elements as different if they do not match (=:=), this module considers two elements as different if and only if they do not compare equal (==).

DATA TYPES

ordered_set()
    as returned by new/0

Exports

call(new) -> Ordset
    Types:
    • Ordset = ordered_set()
    Returns a new empty ordered set.

call(is_set(Ordset)) -> bool()
    Types:
    • Ordset = term()
    Returns true if Ordset is an ordered set of elements, otherwise false.

call(size(Ordset)) -> int()
    Types:
    • Ordset = term()
    Returns the number of elements in Ordset.

call(to_list(Ordset)) -> List
    Types:
    • Ordset = ordered_set()
    • List = [term()]
Returns the elements of Ordset as a list.

`from_list(List) -> Ordset`

Types:
- `List = [term()]`
- `Ordset = ordered_set()`

Returns an ordered set of the elements in List.

`is_element(Element, Ordset) -> bool()`

Types:
- `Element = term()`
- `Ordset = ordered_set()`

Returns true if Element is an element of Ordset, otherwise false.

`add_element(Element, Ordset1) -> Ordset2`

Types:
- `Element = term()`
- `Ordset1 = Ordset2 = ordered_set()`

Returns a new ordered set formed from Ordset1 with Element inserted.

`del_element(Element, Ordset1) -> Ordset2`

Types:
- `Element = term()`
- `Ordset1 = Ordset2 = ordered_set()`

Returns Ordset1, but with Element removed.

`union(Ordset1, Ordset2) -> Ordset3`

Types:
- `Ordset1 = Ordset2 = Ordset3 = ordered_set()`

Returns the merged (union) set of Ordset1 and Ordset2.

`union(OrdsetList) -> Ordset`

Types:
- `OrdsetList = [ordered_set()]`
- `Ordset = ordered_set()`

Returns the merged (union) set of the list of sets.

`intersection(Ordset1, Ordset2) -> Ordset3`

Types:
- `Ordset1 = Ordset2 = Ordset3 = ordered_set()`

Returns the intersection of Ordset1 and Ordset2.
intersection(OrdsetList) -> Ordset

Types:
- OrdsetList = [ordered_set()]
- Ordset = ordered_set()

Returns the intersection of the non-empty list of sets.

is_disjoint(Ordset1, Ordset2) -> bool()

Types:
- Ordset1 = Ordset2 = ordered_set()

Returns true if Ordset1 and Ordset2 are disjoint (have no elements in common), and false otherwise.

subtract(Ordset1, Ordset2) -> Ordset3

Types:
- Ordset1 = Ordset2 = Ordset3 = ordered_set()

Returns only the elements of Ordset1 which are not also elements of Ordset2.

is_subset(Ordset1, Ordset2) -> bool()

Types:
- Ordset1 = Ordset2 = ordered_set()

Returns true when every element of Ordset1 is also a member of Ordset2, otherwise false.

fold(Function, Acc0, Ordset) -> Acc1

Types:
- Function = fun (E, AccIn) -> AccOut
- Acc0 = Acc1 = AccIn = AccOut = term()
- Ordset = ordered_set()

Fold Function over every element in Ordset returning the final value of the accumulator.

filter(Pred, Ordset1) -> Set2

Types:
- Pred = fun (E) -> bool()
- Set1 = Set2 = ordered_set()

Filter elements in Set1 with boolean function Fun.

See Also

gb_sets(3) [page 204], sets(3) [page 397]
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 \( \text{Pid} \) to the process group \( \text{PgName} \). Returns a list of all old members of the group.

\[ \text{send}(\text{PgName}, \text{Msg}) \rightarrow \text{void()} \]

\textbf{Types:}

- \( \text{PgName} = \text{Msg} = \text{term()} \)

Sends the tuple \( \{ \text{pg.message, From, PgName, Msg} \} \) to all members of the process group \( \text{PgName} \).

\textbf{Failure:} \( \text{badarg, \{PgName, Msg\}} \) if \( \text{PgName} \) is not a process group (a globally registered name).

\[ \text{eseend}(\text{PgName}, \text{Msg}) \rightarrow \text{void()} \]

\textbf{Types:}

- \( \text{PgName} = \text{Msg} = \text{term()} \)

Sends the tuple \( \{ \text{pg.message, From, PgName, Msg} \} \) to all members of the process group \( \text{PgName} \), except ourselves.

\textbf{Failure:} \( \text{badarg, \{PgName, Msg\}} \) if \( \text{PgName} \) is not a process group (a globally registered name).

\[ \text{members}(\text{PgName}) \rightarrow \text{Members} \]

\textbf{Types:}

- \( \text{PgName} = \text{term()} \)
- \( \text{Members} = [\text{pid()}] \)

Returns a list of all members of the process group \( \text{PgName} \).
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) ->
```

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 307] 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 412].

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) ->
```

Types:
- `Node` = node()
This function ensures that a pool master is running and includes Node in the pool master's pool of nodes.

stop() -> stopped

Stops the pool and kills all the slave nodes.

get_nodes() -> Nodes

Types:
- Nodes = [node()]
Returns a list of the current member nodes of the pool.

pspawn(Mod, Fun, Args) -> pid()

Types:
- Mod = Fun = atom()
- Args = [term()]
Spawns a process on the pool node which is expected to have the lowest future load.

pspawn_link(Mod, Fun, Args) -> pid()

Types:
- Mod = Fun = atom()
- Args = [term()]
Spawn links a process on the pool node which is expected to have the lowest future load.

get_node() -> node()

Returns the node with the expected lowest future load.

Files

.hosts.erlang is used to pick hosts where nodes can be started. See [net_adm(3)] for information about format and location of this file.

$HOME/.erlang_slave.out.HOST is used for all additional IO that may come from the slave nodes on standard IO. If the start-up procedure does not work, this file may indicate the reason.
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 or {shutdown,Term}. shutdown 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, shutdown, or {shutdown,Term} - 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. Spawn the process and waits for it to start. When the process has started, it must call `init_ack(Parent, Ret)` [page 310] or `init_ack(Ret)` [page 310], 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`.

```erlang
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 309] 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 error_logger:report(error_report(CrashReport)). That is, the event to be handled is of the format {error_report, GL, {Pid, crash_report, CrashReport}} where GL is the group leader pid of the process Pid which sent the crash report.

initial_call(Process) -> {Module,Function,Args} | false

Types:
- Process = pid() | {X,Y,Z} | ProcInfo
- X = Y = Z = int()
- ProcInfo = term()
- Module = Function = atom()
- Args = [atom()]

Extracts the initial call of a process that was started using one of the spawn or start functions described above. Process can either be a pid, an integer tuple (from which a pid can be created), or the process information of a process Pid fetched through an erlang:process_info(Pid) function call.

Note:
The list Args no longer contains the actual arguments, but the same number of atoms as the number of arguments; the first atom is always 'Argument_1', the second 'Argument_2', and so on. The reason is that the argument list could waste a significant amount of memory, and if the argument list contained funs, it could be impossible to upgrade the code for the module.

If the process was spawned using a fun, initial_call/1 no longer returns the actual fun, but the module, function for the local function implementing the fun, and the arity, for instance {some_module,-work/3-fun-0-,0} (meaning that the fun was created in the function some_module:work/3). The reason is that keeping the fun would prevent code upgrade for the module, and that a significant amount of memory could be wasted.

translate_initial_call(Process) -> {Module,Function,Arity} | Fun

Types:
- Process = pid() | {X,Y,Z} | ProcInfo
- X = Y = Z = int()
- ProcInfo = term()
- Module = Function = atom()
- Arity = int()
- Fun = fun() -> void()
This function is used by the c:i/0 and c:regs/0 functions in order to present process information.

Extracts the initial call of a process that was started using one of the spawn or start functions described above, and translates it to more useful information. Process can either be a pid, an integer tuple (from which a pid can be created), or the process information of a process Pid fetched through an erlang:process_info(Pid) function call.

If the initial call is to one of the system defined behaviors such as gen_server or gen_event, it is translated to more useful information. If a gen_server is spawned, the returned Module is the name of the callback module and Function is init (the function that initiates the new server).

A supervisor and a supervisor_bridge are also gen_server processes. In order to return information that this process is a supervisor and the name of the call-back module, Module is supervisor and Function is the name of the supervisor callback module. Arity is 1 since the init/1 function is called initially in the callback module.

By default, {proc_lib, init, P, 5} is returned if no information about the initial call can be found. It is assumed that the caller knows that the process has been spawned with the proc_lib module.

hibernate(Module, Function, Args)

Types:
- Module = Function = atom()
- Args = [term()]

This function does the same as (and does call) the BIF [hibernate/3], but ensures that exception handling and logging continues to work as expected when the process wakes up. Always use this function instead of the BIF for processes started using proc_lib functions.

SEE ALSO

[error_logger(3)]
proplists

Erlang Module

Property lists are ordinary lists containing entries in the form of either tuples, whose first elements are keys used for lookup and insertion, or atoms, which work as shorthand for tuples (\texttt{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

\texttt{append\_values(Key, List) -> List}

\textbf{Types:}
\begin{itemize}
  \item \texttt{Key = term()}
  \item \texttt{List = [term()]}\end{itemize}

Similar to \texttt{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., \texttt{append\_values(a, [{a, [1,2]}, {b, 0}, {a, 3}, {c, -1}, {a, [4]}])} will return the list \[1,2,3,4\].

\texttt{compact(List) -> List}

\textbf{Types:}
\begin{itemize}
  \item \texttt{List = [term()]}\end{itemize}

Minimizes the representation of all entries in the list. This is equivalent to \[\text{[property(P) || P <- List]}\].

See also: property/1, unfold/1.

\texttt{delete(Key, List) -> List}

\textbf{Types:}
\begin{itemize}
  \item \texttt{Key = term()}
  \item \texttt{List = [term()]}\end{itemize}

Deletes all entries associated with \texttt{Key} from \texttt{List}.

\texttt{expand(Expansions, List) -> List}

\textbf{Types:}
Key = term()
Expansions = [{Property, term()}]
Property = atom() | tuple()

Expands particular properties to corresponding sets of properties (or other terms). For each pair `{Property, Expansion}` in `Expansions`, if `E` is the first entry in `List` with the same key as `Property`, and `E` and `Property` have equivalent normal forms, then `E` is replaced with the terms in `Expansion`, and any following entries with the same key are deleted from `List`.

For example, the following expressions all return `[fie, bar, baz, fum]`:

```
expand([{{foo, [bar, baz]}}], [fie, foo, fum])
expand([{{foo, true}, [bar, baz]}}], [fie, foo, fum])
expand([{{foo, false}, [bar, baz]}}], [fie, {foo, false}, fum])
```

However, no expansion is done in the following call:

```
expand([{{foo, true}, [bar, baz]}}], [{foo, false}, fie, foo, fum])
```

because `{foo, false}` shadows `foo`.

Note that if the original property term is to be preserved in the result when expanded, it must be included in the expansion list. The inserted terms are not expanded recursively. If `Expansions` contains more than one property with the same key, only the first occurrence is used.

See also: normalize/2.

get_all_values(Key, List) -> [term()]

Types:
- Key = term()
- List = [term()]

Similar to get_value/2, but returns the list of values for all entries `{Key, Value}` in `List`. If no such entry exists, the result is the empty list.

See also: get_value/2.

get_bool(Key, List) -> bool()

Types:
- Key = term()
- List = [term()]

Returns the value of a boolean key/value option. If `lookup(Key, List)` would yield `{Key, true}`, this function returns `true`; otherwise `false` is returned.

See also: get_value/2, lookup/2.

get_keys(List) -> [term()]

Types:
- List = [term()]
Returns an unordered list of the keys used in List, not containing duplicates.

\[
\text{get\_value(Key, List) -> term()}
\]

Types:
- \(\text{Key} = \text{term()}\)
- \(\text{List} = [\text{term()}]\)

Equivalent to get\_value(Key, List, undefined).

\[
\text{get\_value(Key, List, Default) -> term()}
\]

Types:
- \(\text{Key} = \text{term()}\)
- \(\text{Default} = \text{term()}\)
- \(\text{List} = [\text{term()}]\)

Returns the value of a simple key/value property in List. If \(\text{lookup(Key, List)}\) would yield \{Key, Value\}, this function returns the corresponding Value, otherwise Default is returned.

See also: get\_all\_values/2, get\_bool/2, get\_value/2, lookup/2.

\[
\text{is\_defined(Key, List) -> bool()}
\]

Types:
- \(\text{Key} = \text{term()}\)
- \(\text{List} = [\text{term()}]\)

Returns true if List contains at least one entry associated with Key, otherwise false is returned.

\[
\text{lookup(Key, List) -> none | tuple()}
\]

Types:
- \(\text{Key} = \text{term()}\)
- \(\text{List} = [\text{term()}]\)

Returns the first entry associated with Key in List, if one exists, otherwise returns none. For an atom A in the list, the tuple \{A, true\} is the entry associated with A.

See also: get\_bool/2, get\_value/2, lookup\_all/2.

\[
\text{lookup\_all(Key, List) -> [tuple()]}\]

Types:
- \(\text{Key} = \text{term()}\)
- \(\text{List} = [\text{term()}]\)

Returns the list of all entries associated with Key in List. If no such entry exists, the result is the empty list.

See also: lookup/2.

\[
\text{normalize(List, Stages) -> List}
\]

Types:
Passes List through a sequence of substitution/expansion stages. For an \texttt{aliases} operation, the function \texttt{substitute} is applied using the given list of aliases; for a \texttt{negations} operation, \texttt{substitute} is applied using the given negation list; for an \texttt{expand} operation, the function \texttt{expand} is applied using the given list of expansions. The final result is automatically compacted (cf. \texttt{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: \texttt{compact/1}, \texttt{expand/2}, \texttt{substitute}.

\begin{verbatim}
property(Property) -> Property
    Types:
    • Property = atom() | tuple()

Creates a normal form (minimal) representation of a property. If \texttt{Property} is \texttt{Key}, \text{true} where \texttt{Key} is an atom, this returns \texttt{Key}, otherwise the whole term \texttt{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 \texttt{Key} if \texttt{Value} is \texttt{true} and \texttt{Key} is an atom, otherwise a tuple \texttt{[Key, Value]} is returned.

See also: property/1.

split(List, Keys) -> \{Lists, Rest\}
    Types:
    • List = [term()]  
    • Keys = [term()]  
    • Lists = [[term()]] 
    • Rest = [term()]  
\end{verbatim}
Partitions List into a list of sublists and a remainder. Lists contains one sublist for each key in Keys, in the corresponding order. The relative order of the elements in each sublist is preserved from the original List. Rest contains the elements in List that are not associated with any of the given keys, also with their original relative order preserved.

Example: split([[c, 2], {e, 1}, a, {c, 3, 4}, d, {b, 5}, b], [a, b, c]) returns

$$[[a], [{b, 5}, b], [{c, 2}, {c, 3, 4}]], [{e, 1}, d]]$$

**substituteAliases(Aliases, List)** -> List

Types:
- Aliases = [{Key, Key}]
- Key = term()
- List = [term()]

Substitutes keys of properties. For each entry in List, if it is associated with some key K1 such that {K1, K2} occurs in Aliases, the key of the entry is changed to Key2. If the same K1 occurs more than once in Aliases, only the first occurrence is used.

Example: substituteAliases([{color, colour}], L) will replace all tuples {color, ...} in L with {colour, ...}, and all atoms color with colour.

See also: normalize/2, substituteNegations/2.

**substituteNegations(Negations, List)** -> List

Types:
- Negations = [{Key, Key}]
- Key = term()
- List = [term()]

Substitutes keys of boolean-valued properties and simultaneously negates their values. For each entry in List, if it is associated with some key K1 such that {K1, true} occurs in Negations, then if the entry was {K1, true} it will be replaced with {K2, false}, otherwise it will be replaced with {K2, true}, thus changing the name of the option and simultaneously negating the value given by getBool(List). If the same K1 occurs more than once in Negations, only the first occurrence is used.

Example: substituteNegations([{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: getBool/2, normalize/2, substituteAliases/2.

**unfold(List)** -> List

Types:
- List = [term()]

Unfolds all occurrences of atoms in List to tuples {Atom, true}.
The \texttt{qlc} module provides a query interface to \texttt{Mnesia}, \texttt{ETS}, \texttt{Dets} and other data structures that implement an iterator style traversal of objects.

**Overview**

The \texttt{qlc} module implements a query interface to QLC tables. Typical QLC tables are \texttt{ETS}, \texttt{Dets}, and \texttt{Mnesia} tables. There is also support for user defined tables, see the Implementing a QLC table \cite{page-323} section. A query is stated using QLC List Comprehensions (QLCs). The answers to a query are determined by data in QLC tables that fulfill the constraints expressed by the QLCs of the query. QLCs are similar to ordinary list comprehensions as described in the Erlang Reference Manual and Programming Examples except that variables introduced in patterns cannot be used in list expressions. In fact, in the absence of optimizations and options such as \texttt{cache} and \texttt{unique} (see below), every QLC free of QLC tables evaluates to the same list of answers as the identical ordinary list comprehension.

While ordinary list comprehensions evaluate to lists, calling \texttt{qlc:q/1,2} \cite{page-330} returns a Query Handle. To obtain all the answers to a query, \texttt{qlc:eval/1,2} \cite{page-327} should be called with the query handle as first argument. Query handles are essentially functional objects ("funs") created in the module calling \texttt{q/1,2}. As the funs refer to the module's code, one should be careful not to keep query handles too long if the module's code is to be replaced. Code replacement is described in the \cite{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 \texttt{qlc:cursor/1,2} \cite{page-327} 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:

\[ \text{Expression} \mid |\ \text{Qualifier1, Qualifier2, ...} \]

Expression (the template) is an arbitrary Erlang expression. Qualifiers are either filters or generators. Filters are Erlang expressions returning \texttt{bool()}. Generators have the form \texttt{Pattern \textless - ListExpression}, where \texttt{ListExpression} is an expression evaluating to a query handle or a list. Query handles are returned from \texttt{qlc:table/2, qlc:append/1,2, qlc:sort/1,2, qlc:keysort/2,3, qlc:q/1,2, and qlc:string_to_handle/1,2,3}. 


Evaluation

The evaluation of a query handle begins by the inspection of options and the collection of information about tables. As a result, qualifiers are modified during the optimization phase. Next, all list expressions are evaluated. If a cursor has been created, evaluation takes place in the cursor process. For those list expressions that are QLCs, the list expressions of the QLCs' generators are evaluated as well. One has to be careful if list expressions have side effects since the order in which list expressions are evaluated is unspecified. Finally, the answers are found by evaluating the qualifiers from left to right, backtracking when some filter returns false, or collecting the template when all filters return true.

Filters that do not return bool() but fail are handled differently depending on their syntax: if the filter is a guard it returns false, otherwise the query evaluation fails. This behavior makes it possible for the qlc module to do some optimizations without affecting the meaning of a query. For example, when testing some position of a table and one or more constants for equality, only the objects with equal values are candidates for further evaluation. The other objects are guaranteed to make the filter return false, but never fail. The (small) set of candidate objects can often be found by looking up some key values of the table or by traversing the table using a match specification. It is necessary to place the guard filters immediately after the table's generator, otherwise the candidate objects will not be restricted to a small set. The reason is that objects that could make the query evaluation fail must not be excluded by looking up a key or running a match specification.

Join

The qlc module supports fast join of two query handles. Fast join is possible if some position P1 of one query handler and some position P2 of another query handler are tested for equality. 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 or compare equal. The qlc module does not create any indices but looks up values using the key position and the indexed positions of the QLC table.
- **Merge join** sorts the objects of each query handle if necessary and filters out objects where the values at P1 and P2 do not compare equal. If there are many objects with the same value of P2 a temporary file will be used for the equivalence classes.

The qlc module warns at compile time if a QLC combines query handles in such a way that more than one join is possible. In other words, there is no query planner that can choose a good order between possible join operations. It is up to the user to order the joins by introducing query handles.

The join is to be expressed as a guard filter. The filter must be placed immediately after the two joined generators, possibly after guard filters that use variables from no other generators but the two joined generators. The qlc module inspects the operands of =:=/2, ==/2, is_record/2, element/2, and logical operators (and/2, or/2, andalso/2, orelse/2, xor/2) when determining which joins to consider.
Common options

The following options are accepted by cursor/2, eval/2, fold/4, and info/2:

- \{cache,all\}, Cache where Cache is equal to ets or list adds a \{cache,Cache\} option to every list expression of the query except tables and lists. Default is \{cache,all,no\}. The option cache,all is equivalent to \{cache,all,ets\}.

- \{max,limit\}, MaxListSize where MaxListSize is the size in bytes of terms on the external format. If the accumulated size of collected objects exceeds MaxListSize the objects are written onto a temporary file. This option is used by the \{cache,list\} option as well as by the merge join method. Default is 512*1024 bytes.

- \{tmpdir_usage\}, TmpFileUsage determines the action taken when qlc is about to create temporary files on the directory set by the tmpdir option. If the value is not allowed an error tuple is returned, otherwise temporary files are created as needed. Default is allowed which means that no further action is taken. The values info_msg, warning_msg, and error_msg mean that the function with the corresponding name in the module error_logger is called for printing some information (currently the stacktrace).

- \{tmpdir\}, TempDirectory sets the directory used by merge join for temporary files and by the \{cache,list\} option. The option also overrides the tmpdir option of keysort/3 and sort/2. The default value is "" which means that the directory returned by file:get_cwd() is used.

- \{unique,all, true\} adds a \{unique,true\} option to every list expression of the query. Default is \{unique,all,false\}. The option unique,all is equivalent to \{unique,all,true\}.

Common data types

- QueryCursor = \{qlc_cursor, term()\}
- QueryHandle = \{qlc_handle, term()\}
- QueryHandleOrList = QueryHandle | list()
- Answers = [Answer]
- Answer = term()
- AbstractExpression = -parse trees for Erlang expressions, see the [abstract format] documentation in the ERTS User’s Guide
- MatchExpression = -match specifications, see the [match specification] documentation in the ERTS User's Guide and ms_transform(3) [page 285]
- SpawnOptions = default | spawn_options()
- SortOptions = [SortOption] | SortOption
- SortOption = \{compressed, bool()\} | \{no_files, NoFiles\} | \{order, Order\} | \{size, Size\} | \{tmpdir, TempDirectory\} | \{unique, bool()\}
- TempDirectory = "" | filename()
• Size = int() > 0
• NoFiles = int() > 1
• KeyPos = int() > 0 | [int() > 0]
• MaxListSize = int() >= 0
• bool() = true | false
• Cache = ets | list | no
• TmpFileUsage = allowed | not_allowed | info_msg | warning_msg | error_msg
• filename() = -see filename(3) [page 198]-
• spawn_options() = -see [erlang(3)]-

Getting started

As already mentioned queries are stated in the list comprehension syntax as described in the [Erlang Reference Manual]. In the following some familiarity with list comprehensions is assumed. There are examples in [Programming Examples] that can get you started. It should be stressed that list comprehensions do not add any computational power to the language; anything that can be done with list comprehensions can also be done without them. But they add a syntax for expressing simple search problems which is compact and clear once you get used to it.

Many list comprehension expressions can be evaluated by the qlc module. Exceptions are expressions such that variables introduced in patterns (or filters) are used in some generator later in the list comprehension. As an example consider an implementation of lists:append(L): [X ||Y <- L ,X <- Y ].

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 several 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 323] 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:
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. Both alternatives find duplicates by comparing answers, that is, if A1 and A2 are answers found in that order, then A2 is a removed if A1 == A2.

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])
```

The `qlc` module will evaluate this differently depending on the query handles QH1 and QH2. If, for example, X2 is matched against the key of a QLC table the lookup join method will traverse the objects of QH2 while looking up key values in the table. On the other hand, if neither X2 nor Y2 is matched against the key or an indexed position of a QLC table, the merge join method will make sure that QH1 and QH2 are both sorted on position 2 and next do the join by traversing the objects one by one.

The `join` option can be used to force the `qlc` module to use a certain join method. For the rest of this section it is assumed that the excessively slow join method called “nested loop” has been chosen:

```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 323] section, gb_table:table/1, the iterator for the gb-tree will be initiated for each answer to QH1 after which the objects of the gb-tree will be returned one by one. This is probably the most efficient way of traversing the table in that case.
since it takes minimal computational power to get the following object. But if \( QH2 \) is
not a table but a more complicated QLC, it can be more efficient use some RAM
memory for collecting the answers in a cache, particularly if there are only a few
answers. It must then be assumed that evaluating \( QH2 \) has no side effects so that the
meaning of the query does not change if \( QH2 \) is evaluated only once. One way of
caching the answers is to evaluate \( QH2 \) first of all and substitute the list of answers for
\( QH2 \) in the query. Another way is to use the cache option. It is stated like this:

\[
QH2' = \text{qlc:q}([X \mid X \leftarrow QH2], \{\text{cache}, \text{ets}\})
\]

or just

\[
QH2' = \text{qlc:q}([X \mid X \leftarrow QH2], \text{cache})
\]

The effect of the cache option is that when the generator \( QH2' \) is run the first time
every answer is stored in an ETS table. When next answer of \( QH1 \) is tried, answers to
\( QH2' \) are copied from the ETS table which is very fast. As for the unique option the
cost is a possibly substantial amount of RAM memory. The \{cache, list\} option offers
the possibility to store the answers in a list on the process heap. While this has the
potential of being faster than ETS tables since there is no need to copy answers from the
table it can often result in slower evaluation due to more garbage collections of the
process' heap as well as increased RAM memory consumption due to larger heaps.
Another drawback with cache lists is that if the size of the list exceeds a limit a
temporary file will be used. Reading the answers from a file is very much slower than
copying them from an ETS table. But if the available RAM memory is scarce setting the
limit [page 320] to some low value is an alternative.

There is an option cache_all that can be set to ets or list when evaluating a query. It
adds a cache or \{cache, list\} option to every list expression except QLC tables and
lists on all levels of the query. This can be used for testing if caching would improve
efficiency at all. If the answer is yes further testing is needed to pinpoint the generators
that should be cached.

Implementing a QLC table

As an example of how to use the qlc:table/2 [page 330] function the implementation of
a QLC table for the gb_trees [page 210] module is given:

```erlang
-module(gb_table).
-export([table/1]).

table(T) ->
    TF = fun() -> qlc_next(gb_trees:next(gb_trees:iterator(T))) end,
    InfoFun = fun(num_of_objects) -> gb_trees:size(T);
        (keypos) -> 1;
        (is_sorted_key) -> true;
        (is_unique_objects) -> true;
        (_) -> undefined
    end,
    LookupFun =
        fun(1, Ks) ->
            lists:flatmap(fun(K) ->
                case gb_trees:lookup(K, T) of
                    {value, V} -> {[K,V]};
                end,
                Ks)
        end,
    InfoFun(TF, InfoFun, LookupFun).
```


FormatFun =
  fun({all, NElements, ElementFun}) ->
    ValsS = io_lib:format("gb_trees:from_orddict(~w)",
                        [gb_nodes(T, NElements, ElementFun)]),
    io_lib:format("gb_table:table(~s)", [ValsS]);
  (lookup, 1, KeyValues, _NElements, ElementFun) ->
    ValsS = io_lib:format("gb_trees:from_orddict(~w)",
                        [gb_nodes(T, infinity, ElementFun)]),
    io_lib:format("lists:flatmap(fun(K) -> "
                  "case gb_trees:lookup(K, ~s) of "
                  "{value, V} -> [{K,V}];none -> [] end "
                  "end, ~w)",
                        [ValsS, [ElementFun(KV) || KV <- KeyValues]])
end,
qlc_table(TF, [{info_fun, InfoFun}, {format_fun, FormatFun},
               {lookup_fun, LookupFun}, {key_equality, '=='}]).

qlc_next({X, V, S}) ->
  [{X,V} | fun() -> qlc_next(gb_trees:next(S)) end];
qlc_next(none) ->
  [].

gb_nodes(T, infinity, ElementFun) ->
  gb_nodes(T, -1, ElementFun);
gb_nodes(T, NElements, ElementFun) ->
  gb_iter(gb_trees:iterator(T), NElements, ElementFun).

gb_iter(_, I, 0, _EFun) ->
  '...';
gb_iter(IO, N, EFun) ->
  case gb_trees:next(IO) of
    {X, V, I} ->
      [EFun({X,V}) | gb_iter(I, N-1, EFun)];
    none ->
      []
  end.

TF is the traversal function. The qlc module requires that there is a way of traversing all
objects of the data structure; in gb_trees there is an iterator function suitable for that
purpose. Note that for each object returned a new fun is created. As long as the list is
not terminated by [] it is assumed that the tail of the list is a nullary function and that
calling the function returns further objects (and functions).

The lookup function is optional. It is assumed that the lookup function always finds
values much faster than it would take to traverse the table. The first argument is the
position of the key. Since qlc_next returns the objects as {Key,Value} pairs the position
is 1. Note that the lookup function should return {Key,Value} pairs, just as the traversal
function does.

The format function is also optional. It is called by qlc_info to give feedback at
runtime of how the query will be evaluated. One should try to give as good feedback as
possible without showing too much details. In the example at most 7 objects of the table are shown. The format function handles two cases: all means that all objects of the table will be traversed; {lookup,1,KeyValues} means that the lookup function will be used for looking up key values.

Whether the whole table will be traversed or just some keys looked up depends on how the query is stated. If the query has the form

qlc:q([T || P <- LE, F])

and P is a tuple, the qlc module analyzes P and F in compile time to find positions of the tuple P that are tested for equality to constants. If such a position at runtime turns out to be the key position, the lookup function can be used, otherwise all objects of the table have to be traversed. It is the info function InfoFun that returns the key position. There can be indexed positions as well, also returned by the info function. An index is an extra table that makes lookup on some position fast. Mnesia maintains indices upon request, thereby introducing so called secondary keys. The qlc module prefers to look up objects using the key before secondary keys regardless of the number of constants to look up.

Key equality

In Erlang there are two operators for testing term equality, namely ==/2 and =:=/2. The difference between them is all about the integers that can be represented by floats. For instance, 2 == 2.0 evaluates to true while 2 =:= 2.0 evaluates to false. Normally this is a minor issue, but the qlc module cannot ignore the difference, which affects the user’s choice of operators in QLCs.

If the qlc module can find out at compile time that some constant is free of integers, it does not matter which one of ==/2 or =:=/2 is used:

1> E1 = ets:new(t, [set]), % uses =:=/2 for key equality
   Q1 = qlc:q([K || K <- ets:table(E1),
               K == 2.71 orelse K == a]),
   io:format("~s~n", [qlc:info(Q1)]).

In the example the ==/2 operator has been handled exactly as =:=/2 would have been handled. On the other hand, if it cannot be determined at compile time that some constant is free of integers and the table uses =:=/2 when comparing keys for equality (see the option key_equality [page 336]), the qlc module will not try to look up the constant. The reason is that there is in the general case no upper limit on the number of key values that can compare equal to such a constant; every combination of integers and floats has to be looked up:

2> E2 = ets:new(t, [set]),
   true = ets:insert(E2, [[2,2],a],[[2,2.0],b],[[2.0,2],c]],
   F2 = fun(I) ->
           qlc:q([V || {K,V} <- ets:table(E2), K == I])
       end,
Q2 = F2(\{2,2\}),
io:format("\"s\"n", [qlc:info(Q2)]).
ets:table(53264,
  [{traverse,
    {select,[[\{'$1','$2'\},[\'=','=','$1',\{const,\{2,2\}\}],[\'$2'\}]]}]})
3> lists:sort(qlc:e(Q2)).
a,b,c

Looking up just \{2,2\} would not return b and c.
If the table uses \=/2 when comparing keys for equality, the qlc module will look up the constant regardless of which operator is used in the QLC. However, \=/2 is to be preferred:

4> E3 = ets:new(t, [ordered_set]), % uses \=/2 for key equality
tue = ets:insert(E3, [[\{2,2.0\},b]],
F3 = fun(I) ->
qlc:q([[V ||
  {K,V} <- ets:table(E3), K == I]])
end,
Q3 = F3(\{2,2\}),
io:format("\"s\"n", [qlc:info(Q3)]).
ets:match_spec_run(ets:lookup(86033, \{2,2\}),
  ets:match_spec_compile([[\{'$1','$2'\},[],[\'$2'\}]]))
5> qlc:e(Q3).
b
Lookup join is handled analogously to lookup of constants in a table: if the join operator is \=/2 and the table where constants are to be looked up uses \=:=/2 when testing keys for equality, the qlc module will not consider lookup join for that table.

Exports

append(QHL) :- QH
Types:
  • QHL = [QueryHandleOrList]
  • QH = QueryHandle
Returns a query handle. When evaluating the query handle QH all answers to the first query handle in QHL is returned followed by all answers to the rest of the query handles in QHL.

append(QH1, QH2) :- QH3
Types:
  • QH1 = QH2 = QueryHandleOrList
  • QH3 = QueryHandle
Returns a query handle. When evaluating the query handle QH3 all answers to QH1 are returned followed by all answers to QH2.
append(QH1, QH2) is equivalent to append([QH1, QH2]).

cursor(QueryHandleOrList [, Options]) :- QueryCursor
Types:
-  Options = [Options] | Option
-  Option = {cache_all, Cache} | cache_all | {max_list_size, MaxListSize} | {spawn_options, SpawnOptions} | {tmpdir_usage, TmpFileUsage} | {tmpdir, TempDirectory} | {unique_all, bool()} | unique_all

Creates a query cursor and makes the calling process the owner of the cursor. The cursor is to be used as argument to next_answers/1,2 and (eventually) delete_cursor/1. Calls erlang:spawn_opt to spawn and link a process which will evaluate the query handle. The value of the option spawn_options is used as last argument when calling spawn_opt. The default value is [link].

```
1> QH = qlc:q([[X,Y] || X <- [a,b], Y <- [1,2]]),
   QC = qlc:cursor(QH),
   qlc:next_answers(QC, 1).
   [(a,1)]
2> qlc:next_answers(QC, 1).
   [(a,2)]
3> qlc:next_answers(QC, all_remaining).
   [(b,1), (b,2)]
4> qlc:delete_cursor(QC).
   ok
```

`delete_cursor(QueryCursor) -> ok`

Deletes a query cursor. Only the owner of the cursor can delete the cursor.

eval(QueueHandleOrList [, Options]) -> Answers | Error
ere(QueueHandleOrList [, Options]) -> Answers

Types:
-  Options = [Options] | Option
-  Option = {cache_all, Cache} | cache_all | {max_list_size, MaxListSize} | {tmpdir_usage, TmpFileUsage} | {tmpdir, TempDirectory} | {unique_all, bool()} | unique_all
-  Error = {error, module(), Reason}
-  Reason = as returned by file_sorter/3

Evaluates a query handle in the calling process and collects all answers in a list.

```
1> QH = qlc:q([[X,Y] || X <- [a,b], Y <- [1,2]]),
   qlc:eval(QH).
   [{a,1},{a,2},{b,1},{b,2}]
```

fold(Function, Acc0, QueueHandleOrList [, Options]) -> Acc1 | Error

Types:
-  Function = fun(Answer, AccIn) -> AccOut
-  Acc0 = Acc1 = AccOut = term()
-  Options = [Options] | Option
-  Option = {cache_all, Cache} | cache_all | {max_list_size, MaxListSize} | {tmpdir_usage, TmpFileUsage} | {tmpdir, TempDirectory} | {unique_all, bool()} | unique_all
qlc STDLIB Reference Manual

- Error = {error, module(), Reason}
- Reason = as returned by file_sorter(3)

Calls Function on successive answers to the query handle together with an extra argument AccIn. The query handle and the function are evaluated in the calling process. Function must return a new accumulator which is passed to the next call. Acc0 is returned if there are no answers to the query handle.

```
1> QH = [1,2,3,4,5,6],
   qlc:fold(fun(X, Sum) -> X + Sum end, 0, QH).
21
```

format_error(Error) -> Chars

Types:
- Error = {error, module(), term()}
- Chars = [char() | Chars]

Returns a descriptive string in English of an error tuple returned by some of the functions of the qlc module or the parse transform. This function is mainly used by the compiler invoking the parse transform.

info(QueryHandleOrList [, Options]) -> Info

Types:
- Options = [Option] | Option
  - Option = EvalOption | ReturnOption
    - EvalOption = {cache_all, Cache} | cache_all | {max_list_size, MaxListSize} |
      {tmpdir_usage, TempFileUsage} | {tmpdir, TempDirectory} | {unique_all, bool()} | unique_all
    - ReturnOption = {depth, Depth} | {flat, bool()} | {format, Format} | {n_elements, NElements}
  - Depth = infinity | int() > = 0
  - Format = abstract_code | string
  - NElements = infinity | int() > 0
  - Info = AbstractExpression | string()

Returns information about a query handle. The information describes the simplifications and optimizations that are the results of preparing the query for evaluation. This function is probably useful mostly during debugging.

The information has the form of an Erlang expression where QLCs most likely occur. Depending on the format functions of mentioned QLC tables it may not be absolutely accurate.

The default is to return a sequence of QLCs in a block, but if the option {flat, false} is given, one single QLC is returned. The default is to return a string, but if the option {format, abstract_code} is given, abstract code is returned instead. In the abstract code port identifiers, references, and pids are represented by strings. The default is to return all elements in lists, but if the {n_elements, NElements} option is given, only a limited number of elements are returned. The default is to show all of objects and match specifications, but if the {depth, Depth} option is given, parts of terms below a certain depth are replaced by ‘...’.
1> QH = qlc:q([[X,Y] || X <- [x,y], Y <- [a,b]]),
io:format("s\n", [qlc:info(QH, unique, all)]).
begin
  V1 =
    qlc:q([[SQV ||
          SQV <- [x,y],
          [unique, true]]),
    V2 =
    qlc:q([[SQV ||
          SQV <- [a,b],
          [unique, true]]),
    qlc:q([[X,Y] ||
          X <- V1,
          Y <- V2],
          [unique, true]))
end

In this example two simple QLCs have been inserted just to hold the {unique, true} option.

1> E1 = ets:new(e1, []),
   E2 = ets:new(e2, []),
   true = ets:insert(E1, [[1,a],[2,b]]),
   true = ets:insert(E2, [[a,1],[b,2]]),
   Q = qlc:q([[X,Z,W] ||
             [X,Z] <- ets:table(E1),
             [W,Y] <- ets:table(E2),
             X =:= Y]),
io:format("s\n", [qlc:info(Q)]).
begin
  V1 =
    qlc:q([[P0 ||
            P0 = [W,Y] <- ets:table(17)]),
  V2 =
    qlc:q([[G1|G2] ||
            G2 <- V1,
            G1 <- ets:table(16),
            element(2, G1) =:= element(1, G2),
            [join, lookup]]),
    qlc:q([[X,Z,W] ||
            [[X,Z]|{W,Y}] <- V2])
end

In this example the query list comprehension V2 has been inserted to show the joined generators and the join method chosen. A convention is used for lookup join: the first generator (G2) is the one traversed, the second one (G1) is the table where constants are looked up.

keysort(KeyPos, QH1 [, SortOptions]) -> QH2

Types:
- QH1 = QueryHandleOrList
- QH2 = QueryHandle
Returns a query handle. When evaluating the query handle \texttt{QH2} the answers to the query handle \texttt{QH1} are sorted by \texttt{file:sorter:keysort/4} according to the options.

The sorter will use temporary files only if \texttt{QH1} does not evaluate to a list and the size of the binary representation of the answers exceeds \texttt{Size} bytes, where \texttt{Size} is the value of the \texttt{size} option.

\begin{verbatim}
next_answers(QueryCursor [, NumberOfAnswers]) -> Answers | Error

Types:
- NumberOfAnswers = all_remaining | int() > 0
- Error = {error, module(), Reason}
- Reason = as returned by \texttt{file:sorter(3)}

Returns some or all of the remaining answers to a query cursor. Only the owner of \texttt{Cursor} can retrieve answers.

The optional argument \texttt{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 \texttt{next_answers} will return [].
\end{verbatim}

\begin{verbatim}
q(QueryListComprehension [, Options]) -> QueryHandle

Types:
- QueryListComprehension = -literal query listcomprehension-
- Options = [Option] | Option
  - Option = \{max_lookup, MaxLookup\} | \{cache, Cache\} | cache | \{join, Join\} | lookup, Lookup
  - MaxLookup = int() >= 0 | infinity
  - Join = any | lookup | merge | nested_loop
  - Lookup = bool() | any

Returns a query handle for a query list comprehension. The query list comprehension must be the first argument to \texttt{qlc:q/1,2} or it will be evaluated as an ordinary list comprehension. It is also necessary to add the line
\begin{verbatim}
-includelib("stdlib/include/qlc.hrl")
\end{verbatim}
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 \texttt{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 \texttt{erl:eval(3)}. This is also true when expressions are evaluated by means of \texttt{file:eval/1,2} or in the debugger.

To be very explicit, this will not work:
\begin{verbatim}
... A = [X || X <- [{1},{2}]],
QH = qlc:q(A),
...\end{verbatim}
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.

```prolog
1> Q = qlc:q([[A,X,Z,W] ||
  A <- [a,b,c],
  [X,Z] <- [[a,1],[b,4],[c,6]],
  [W,Y] <- [[2,a],[3,b],[4,c]],
  X =:= Y],
  \{cache, list\}),
io:format("s\n", [qlc:info(Q)])).
```

```
begin
  V1 = qlc:q([P0 ||
    P0 = {X,Z} <- qlc:keysort(1, [[a,1],[b,4],[c,6]], []),
  ],
  V2 = qlc:q([P0 ||
    P0 = {W,Y} <- qlc:keysort(2, [[2,a],[3,b],[4,c]], []),
  ],
  V3 = qlc:q([[G1|G2] ||
    G1 <- V1,
    G2 <- V2,
    element(1, G1) == element(2, G2)],
```
In this example the cached results of the merge join are traversed for each value of A. Note that without the cache option the join would have been carried out three times, once for each value of A.

sort/1,2 and keysort/2,3 can also be used for caching answers and for removing duplicates. When sorting answers are cached in a list, possibly stored on a temporary file, and no ETS tables are used.

Sometimes (see qlc:table/2 [page 335] below) traversal of tables can be done by looking up key values, which is assumed to be fast. Under certain (rare) circumstances it could happen that there are too many key values to look up. The {max_lookup,MaxLookup} option can then be used to limit the number of lookups: if more than MaxLookup lookups would be required no lookups are done but the table traversed instead. The default value is infinity which means that there is no limit on the number of keys to look up.

1> T = gb_trees:empty(),
   QH = qlc:q([[X || {X,Y},_] <- gb_table:table(T),
                 (X == 1) or (X == 2)) andalso
                 ((Y == a) or (Y == b) or (Y == c))],
               io:format(""a"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,

   [[1,a],[1,b],[1,c],[2,a],[2,b],[2,c]],
   ets:match_spec_compile([{{'$1'','$2'','$3'},{[]},['$1']}])])

In this example using the gb_table module from the Implementing a QLC table [page 323] 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 compared separately.

The {lookup,true} option can be used to ensure that the qlc module will look up constants in some QLC table. If there are more than one QLC table among the generators' list expressions, constants have to be looked up in at least one of the tables. The evaluation of the query fails if there are no constants to look up. This option is useful in situations when it would be unacceptable to traverse all objects in some table. Setting the lookup option to false ensures that no constants will be looked up ({max_lookup,0} has the same effect). The default value is any which means that constants will be looked up whenever possible.
The \{join,Join\} option can be used to ensure that a certain join method will be used: \{join,lookup\} invokes the lookup join method; \{join,merge\} invokes the merge join method; and \{join,nested_loop\} invokes the method of matching every pair of objects from two handles. The last method is mostly very slow. The evaluation of the query fails if the qlc module cannot carry out the chosen join method. The default value is any which means that some fast join method will be used if possible.

\[
\text{sort(QH1 [, SortOptions]] } \rightarrow \text{ QH2}
\]

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 \text{file_sorter:sort/3} \text{[page 190]} 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 \text{size} option.

\[
\text{string_to_handle(QueryString [, Options [, Bindings]]]) } \rightarrow \text{ QueryHandle | Error}
\]

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 \text{erl_eval:bindings/1}-
- Error = \{error, module(), Reason\}
- Reason = -\text{ErrorInfo} as returned by \text{erl_scan:string/1} or \text{erl_parse:parse_exprs/1}-

A string version of qlc:q/1,2. When the query handle is evaluated the fun created by the parse transform is interpreted by \text{erl_eval(3)}. The query string is to be one single query list comprehension terminated by a period.

\[
1> L = [1,2,3],
Bs = \text{erl_eval:add_binding('L', L, erl_eval:new_bindings())},
QH = \text{qlc:string_to_handle("[X+1 || X <- L].", [], Bs)},
\text{qlc:eval(QH)}.
[2,3,4]
\]

This function is probably useful mostly when called from outside of Erlang, for instance from a driver written in C.

\[
\text{table(TraverseFun, Options)} } \rightarrow \text{ QueryHandle}
\]

Types:
- TraverseFun = TraverseFun0 | TraverseFun1
- TraverseFun0 = fun() \rightarrow TraverseResult
- TraverseFun1 = fun(MatchExpression) \rightarrow TraverseResult
- TraverseResult = Objects | term()
Returns a query handle for a QLC table. In Erlang/OTP there is support for ETS, Dets and Mnesia tables, but it is also possible to turn many other data structures into QLC tables. The way to accomplish this is to let function(s) in the module implementing the data structure create a query handle by calling `qlc:table/2`. The different ways to traverse the table as well as properties of the table are handled by callback functions provided as options to `qlc:table/2`.

The callback function `TraverseFun` is used for traversing the table. It is to return a list of objects terminated by either `[]` or a nullary fun to be used for traversing the not yet traversed objects of the table. Any other return value is immediately returned as value of the query evaluation. Unary `TraverseFuns` are to accept a match specification as argument. The match specification is created by the parse transform by analyzing the pattern of the generator calling `qlc:table/2` and filters using variables introduced in the pattern. If the parse transform cannot find a match specification equivalent to the pattern and filters, `TraverseFun` will be called with a match specification returning every object. Modules that can utilize match specifications for optimized traversal of tables should call `qlc:table/2` with a unary `TraverseFun` while other modules can provide a nullary `TraverseFun`. `ets:table/2` is an example of the former; `gb_table:table/1` in the Implementing a QLC table [page 323] section is an example of the latter.
PreFun is a unary callback function that is called once before the table is read for the first time. If the call fails, the query evaluation fails. Similarly, the nullary callback function PostFun is called once after the table was last read. The return value, which is caught, is ignored. If PreFun has been called for a table, PostFun is guaranteed to be called for that table, even if the evaluation of the query fails for some reason. The order in which pre (post) functions for different tables are evaluated is not specified. Other table access than reading, such as calling InfoFun, is assumed to be OK at any time.

The argument PreArgs is a list of tagged values. Currently there are two tags, parent_value and stop_fun, used by Mnesia for managing transactions. The value of parent_value is the value returned by ParentFun, or undefined if there is no ParentFun. ParentFun is called once just before the call of PreFun in the context of the process calling eval, fold, or cursor. The value of stop_fun is a nullary fun that deletes the cursor if called from the parent, or undefined if there is no cursor.

The binary callback function LookupFun is used for looking up objects in the table. The first argument Position is the key position or an indexed position and the second argument Keys is a sorted list of unique values. The return value is to be a list of all objects (tuples) such that the element at Position is a member of Keys. Any other return value is immediately returned as value of the query evaluation. LookupFun is called instead of traversing the table if the parse transform at compile time can find out that the filters match and compare the element at Position in such a way that only Keys need to be looked up in order to find all potential answers. The key position is obtained by calling InfoFun(keypos) and the indexed positions by calling InfoFun(indices). If the key position can be used for lookup it is always chosen, otherwise the indexed position requiring the least number of lookups is chosen. If there is a tie between two indexed positions the one occurring first in the list returned by InfoFun is chosen. Positions requiring more than max_lookup [page 332] lookups are ignored.

The unary callback function InfoFun is to return information about the table. undefined should be returned if the value of some tag is unknown:

- indices. Returns a list of indexed positions, a list of positive integers.
- is_unique_objects. Returns true if the objects returned by TraverseFun are unique.
- keypos. Returns the position of the table’s key, a positive integer.
- is_sorted_key. Returns true if the objects returned by TraverseFun are sorted on the key.
- num_of_objects. Returns the number of objects in the table, a non-negative integer.

The unary callback function FormatFun is used by qlc:info/1,2 [page 328] for displaying the call that created the table’s query handle. The default value, undefined, means that info/1,2 displays a call to 'MOD': 'FUN'/0. It is up to FormatFun to present the selected objects of the table in a suitable way. However, if a character list is chosen for presentation it must be an Erlang expression that can be scanned and parsed (a trailing dot will be added by qlc:info though). FormatFun is called with an argument that describes the selected objects based on optimizations done as a result of analyzing the filters of the QLC where the call to qlc:table/2 occurs. The possible values of the argument are:

- [lookup, Position, Keys, NElements, DepthFun]. LookupFun is used for looking up objects in the table.
• \{match spec, MatchExpression\}. No way of finding all possible answers by looking up keys was found, but the filters could be transformed into a match specification. All answers are found by calling TraverseFun(MatchExpression).

• \{all, NElements, DepthFun\}. No optimization was found. A match specification matching all objects will be used if TraverseFun is unary.

NElements is the value of the info/1,2 option nElements, and DepthFun is a function that can be used for limiting the size of terms; calling DepthFun(Term) substitutes '...' for parts of Term below the depth specified by the info/1,2 option depth. If calling FormatFun with an argument including NElements and DepthFun fails, FormatFun is called once again with an argument excluding NElements and DepthFun \{lookup,Position,Keys\} or all.

The value of key_equality is to be '===' if the table considers two keys equal if they match, and to be '===' if two keys are equal if they compare equal. The default is '==='. See ets(3) [page 186], dets(3) [page 118] and [mnesia(3)] for the various options recognized by table/1,2 in respective module.

See Also

dets(3) [page 104], [Erlang Reference Manual], erl_eval(3) [page 139], [erlang(3)], ets(3) [page 166], [file(3)], [error_logger(3)], file_sorter(3) [page 190], [mnesia(3)], [Programming Examples], shell(3) [page 400]
queue

Erlang Module

This module implements (double ended) FIFO queues in an efficient manner.

All functions fail with reason \texttt{badarg} if arguments are of wrong type, for example queue arguments are not queues, indexes are not integers, list arguments are not lists. Improper lists cause internal crashes. A n index out of range for a queue also causes a failure with reason \texttt{badarg}.

Some functions, where noted, fail with reason \texttt{empty} for an empty queue.

The data representing a queue as used by this module should be regarded as opaque by other modules. Any code assuming knowledge of the format is running on thin ice.

All operations has an amortized \(O(1)\) running time, except \texttt{len/1}, \texttt{join/2}, \texttt{split/2}, \texttt{filter/2} and \texttt{member/2} that have \(O(n)\). To minimize the size of a queue minimizing the amount of garbage built by queue operations, the queues do not contain explicit length information, and that is why \texttt{len/1} is \(O(n)\). If better performance for this particular operation is essential, it is easy for the caller to keep track of the length.

Queues are double ended. The mental picture of a queue is a line of people (items) waiting for their turn. The queue front is the end with the item that has waited the longest. The queue rear is the end an item enters when it starts to wait. If instead using the mental picture of a list, the front is called head and the rear is called tail.

Entering at the front and exiting at the rear are reverse operations on the queue.

The module has several sets of interface functions. The “Original API”, the “Extended API” and the “Okasaki API”.

The “Original API” and the “Extended API” both use the mental picture of a waiting line of items. Both also have reverse operations suffixed “\texttt{r}”.

The “Original API” item removal functions return compound terms with both the removed item and the resulting queue. The “Extended API” contain alternative functions that build less garbage as well as functions for just inspecting the queue ends. Also the “Okasaki API” functions build less garbage.

The “Okasaki API” is inspired by “Purely Functional Data structures” by Chris Okasaki. It regards queues as lists. The API is by many regarded as strange and avoidable. For example many reverse operations have lexically reversed names, some with more readable but perhaps less understandable aliases.
Original API

Exports

\texttt{new()} \rightarrow \texttt{Q}

Types:
- \texttt{Q} = \texttt{queue()}

Returns an empty queue.

\texttt{is\_queue(Term)} \rightarrow \texttt{true} \mid \texttt{false}

Types:
- \texttt{Term} = \texttt{term()}

Tests if \texttt{Q} is a queue and returns \texttt{true} if so and \texttt{false} otherwise.

\texttt{is\_empty(Q)} \rightarrow \texttt{true} \mid \texttt{false}

Types:
- \texttt{Q} = \texttt{queue()}

Tests if \texttt{Q} is empty and returns \texttt{true} if so and \texttt{false} otherwise.

\texttt{len(Q)} \rightarrow \texttt{N}

Types:
- \texttt{Q} = \texttt{queue()}
- \texttt{N} = \texttt{integer()}

Calculates and returns the length of queue \texttt{Q}.

\texttt{in(Item, Q1)} \rightarrow \texttt{Q2}

Types:
- \texttt{Item} = \texttt{term()}
- \texttt{Q1} = \texttt{Q2} = \texttt{queue()}

Inserts \texttt{Item} at the rear of queue \texttt{Q1}. Returns the resulting queue \texttt{Q2}.

\texttt{in\_r(Item, Q1)} \rightarrow \texttt{Q2}

Types:
- \texttt{Item} = \texttt{term()}
- \texttt{Q1} = \texttt{Q2} = \texttt{queue()}

Inserts \texttt{Item} at the front of queue \texttt{Q1}. Returns the resulting queue \texttt{Q2}.

\texttt{out(Q1)} \rightarrow \texttt{Result}

Types:
- \texttt{Result} = \{\{\texttt{value, Item}, Q2\} \mid \{\texttt{empty, Q1}\}
- \texttt{Q1} = \texttt{Q2} = \texttt{queue()}

Removes the item at the front of queue \( Q1 \). Returns the tuple \( \{ \text{value}, \text{Item} \}, Q2 \) \( \), where \( \text{Item} \) is the item removed and \( Q2 \) is the resulting queue. If \( Q1 \) is empty, the tuple \( \{ \text{empty}, Q1 \} \) is returned.

\[
\text{out}\_r(Q1) \rightarrow \text{Result}
\]

Types:
- \( \text{Result} = \{ \{ \text{value}, \text{Item} \}, Q2 \} \mid \{ \text{empty}, Q1 \} \)
- \( Q1 = Q2 = \text{queue()} \)

Removes the item at the rear of the queue \( Q1 \). Returns the tuple \( \{ \{ \text{value}, \text{Item} \}, Q2 \} \), where \( \text{Item} \) is the item removed and \( Q2 \) is the new queue. If \( Q1 \) is empty, the tuple \( \{ \text{empty}, Q1 \} \) is returned.

\[
\text{from}\_\text{list}(L) \rightarrow \text{queue()}
\]

Types:
- \( L = \text{list()} \)

Returns a queue containing the items in \( L \) in the same order; the head item of the list will become the front item of the queue.

\[
\text{to}\_\text{list}(Q) \rightarrow \text{list()}
\]

Types:
- \( Q = \text{queue()} \)

Returns a list of the items in the queue in the same order; the front item of the queue will become the head of the list.

\[
\text{reverse}(Q1) \rightarrow Q2
\]

Types:
- \( Q1 = Q2 = \text{queue()} \)

Returns a queue \( Q2 \) that contains the items of \( Q1 \) in the reverse order.

\[
\text{split}(N, Q1) \rightarrow \{ Q2, Q3 \}
\]

Types:
- \( N = \text{integer()} \)
- \( Q1 = Q2 = Q3 = \text{queue()} \)

Splits \( Q1 \) in two. The \( N \) front items are put in \( Q2 \) and the rest in \( Q3 \).

\[
\text{join}(Q1, Q2) \rightarrow Q3
\]

Types:
- \( Q1 = Q2 = Q3 = \text{queue()} \)

Returns a queue \( Q3 \) that is the result of joining \( Q1 \) and \( Q2 \) with \( Q1 \) in front of \( Q2 \).

\[
\text{filter}(\text{Fun}, Q1) \rightarrow Q2
\]

Types:
- \( \text{Fun} = \text{fun(\text{Item})} \rightarrow \text{bool()} \mid \text{list()} \)
• \(Q1 = Q2 = \text{queue}()\)

Returns a queue \(Q2\) that is the result of calling \(\text{Fun}(\text{Item})\) on all items in \(Q1\), in order from front to rear.

If \(\text{Fun}(\text{Item})\) returns true, \(\text{Item}\) is copied to the result queue. If it returns false, \(\text{Item}\) is not copied. If it returns a list the list elements are inserted instead of \(\text{Item}\) in the result queue.

So, \(\text{Fun}(\text{Item})\) returning \([\text{Item}]\) is thereby semantically equivalent to returning true, just as returning \([\]\) is semantically equivalent to returning false. But returning a list builds more garbage than returning an atom.

\[
\text{member(}\text{Item, Q}\text{)} \rightarrow \text{bool()}
\]

Types:
• \(\text{Item} = \text{term()}\)
• \(\text{Q} = \text{queue()}\)

Returns true if \(\text{Item}\) matches some element in \(\text{Q}\), otherwise false.

### Extended API

### Exports

\[
\text{get(}\text{Q}\text{)} \rightarrow \text{Item}
\]

Types:
• \(\text{Item} = \text{term()}\)
• \(\text{Q} = \text{queue()}\)

Returns \(\text{Item}\) at the front of queue \(\text{Q}\).

Fails with reason empty if \(\text{Q}\) is empty.

\[
\text{get_r(}\text{Q}\text{)} \rightarrow \text{Item}
\]

Types:
• \(\text{Item} = \text{term()}\)
• \(\text{Q} = \text{queue()}\)

Returns \(\text{Item}\) at the rear of queue \(\text{Q}\).

Fails with reason empty if \(\text{Q}\) is empty.

\[
\text{drop(}\text{Q1}\text{)} \rightarrow \text{Q2}
\]

Types:
• \(\text{Item} = \text{term()}\)
• \(\text{Q1} = \text{Q2} = \text{queue()}\)

Returns a queue \(Q2\) that is the result of removing the front item from \(Q1\).

Fails with reason empty if \(Q1\) is empty.

\[
\text{drop_r(}\text{Q1}\text{)} \rightarrow \text{Q2}
\]
Types:
- \text{Item} = \text{term}()
- Q_1 = Q_2 = \text{queue}()

Returns a queue \( Q_2 \) that is the result of removing the rear item from \( Q_1 \).
Fails with reason \text{empty} if \( Q_1 \) is empty.

\[
\text{peek}(Q) \rightarrow \{\text{value, Item} | \text{empty}\}
\]

Types:
- \text{Item} = \text{term}()
- Q = \text{queue}()

Returns the tuple \( \{\text{value, Item}\} \) where \text{Item} is the front item of \( Q \), or \text{empty} if \( Q_1 \) is empty.

\[
\text{peek}_r(Q) \rightarrow \{\text{value, Item} | \text{empty}\}
\]

Types:
- \text{Item} = \text{term}()
- Q = \text{queue}()

Returns the tuple \( \{\text{value, Item}\} \) where \text{Item} is the rear item of \( Q \), or \text{empty} if \( Q_1 \) is empty.

\textbf{Okasaki API}

\textbf{Exports}

\[
\text{cons}(\text{Item}, Q_1) \rightarrow Q_2
\]

Types:
- \text{Item} = \text{term}()
- Q_1 = Q_2 = \text{queue}()

Inserts \text{Item} at the head of queue \( Q_1 \). Returns the new queue \( Q_2 \).

\[
\text{head}(Q) \rightarrow \text{Item}
\]

Types:
- \text{Item} = \text{term}()
- Q = \text{queue}()

Returns \text{Item} from the head of queue \( Q \).
Fails with reason \text{empty} if \( Q \) is empty.

\[
\text{tail}(Q_1) \rightarrow Q_2
\]

Types:
- \text{Item} = \text{term}()
- Q_1 = Q_2 = \text{queue}()
Returns a queue \( Q_2 \) that is the result of removing the head item from \( Q_1 \).
Fails with reason empty if \( Q_1 \) is empty.

\[ \text{snoc}(Q_1, \text{Item}) \rightarrow Q_2 \]

Types:
- \( \text{Item} = \text{term()} \)
- \( Q_1 = Q_2 = \text{queue()} \)

Inserts \( \text{Item} \) as the tail item of queue \( Q_1 \). Returns the new queue \( Q_2 \).

\[ \text{dash}(Q) \rightarrow \text{Item} \]
\[ \text{last}(Q) \rightarrow \text{Item} \]

Types:
- \( \text{Item} = \text{term()} \)
- \( Q = \text{queue()} \)

Returns the tail item of queue \( Q \).
Fails with reason empty if \( Q \) is empty.

\[ \text{liat}(Q_1) \rightarrow Q_2 \]
\[ \text{init}(Q_1) \rightarrow Q_2 \]
\[ \text{lait}(Q_1) \rightarrow Q_2 \]

Types:
- \( \text{Item} = \text{term()} \)
- \( Q_1 = Q_2 = \text{queue()} \)

Returns a queue \( Q_2 \) that is the result of removing the tail item from \( Q_1 \).
Fails with reason empty if \( Q_1 \) is empty.
The name lait/1 is a misspelling - do not use it anymore.
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.

One way of obtaining a seed is to use the BIF `now/0`:

...{A1,A2,A3} = now(),
random:seed(A1, A2, A3),
...

seed0() -> ran()

Returns the default state.

uniform() -> float()

Returns a random float uniformly distributed between 0.0 and 1.0, updating the state in the process dictionary.

uniform(N) -> int()


Types:
- $N = \text{int}()$

Given an integer $N \geq 1$, `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:
- $N = \text{int}()$
- State0 = State1 = ran()

Given an integer $N \geq 1$ and a state, `uniform_s/2` returns a random integer uniformly distributed between 1 and $N$, and a new state.

**Note**

Some of the functions use the process dictionary variable `random_seed` to remember the current seed.

If a process calls `uniform/0` or `uniform/1` without setting a seed first, `seed/0` is called automatically.
This module contains functions for regular expression matching for strings and binaries. The regular expression syntax and semantics resemble that of Perl. This library in many ways replaces the old regexp library written purely in Erlang, as it has a richer syntax as well as many more options. The library is also faster than the older regexp implementation.

Although the library's matching algorithms are currently based on the PCRE library, it is not to be viewed as an Erlang to PCRE mapping. Only parts of the PCRE library is interfaced and the re library in some ways extend PCRE. The PCRE documentation contains many parts of no interest to the Erlang programmer, why only the relevant part of the documentation is included here. There should bee no need to go directly to the PCRE library documentation.

**Note:**
The Erlang literal syntax for strings give special meaning to the "\" (backslash) character. To literally write a regular expression or a replacement string containing a backslash in your code or in the shell, two backslashes have to be written: "\\".

### DATA TYPES

- **iodata()** = iolist() | binary()
- **iolist()** = [char() | binary() | iolist()]
  - a binary is allowed as the tail of the list
- **unicode_binary()** = binary() with characters encoded in UTF-8 coding standard
- **unicode_char()** = integer() representing valid unicode codepoint
- **chardata()** = charlist() | unicode_binary()
- **charlist()** = [unicode_char() | unicode_binary() | charlist()]
  - a unicode_binary is allowed as the tail of the list
- **mp()** = Opaque datatype containing a compiled regular expression.
Exports

\[
\text{compile}(\text{Regexp}) \to \{\text{ok}, \text{MP}\} \mid \{\text{error}, \text{ErrSpec}\}
\]

Types:
- \text{Regexp} = \text{iodata()}
- The same as \text{compile}(\text{Regexp}, [])

\[
\text{compile}(\text{Regexp}, \text{Options}) \to \{\text{ok}, \text{MP}\} \mid \{\text{error}, \text{ErrSpec}\}
\]

Types:
- \text{Regexp} = \text{iodata()} \mid \text{charlist()}
- \text{Options} = [ \text{Option} ]
- \text{Option} = \text{unicode} \mid \text{anchored} \mid \text{caseless} \mid \text{dollar_endonly} \mid \text{dotall} \mid \text{extended} \mid \text{firstline} \mid \text{multiline} \mid \text{no_auto_capture} \mid \text{dupnames} \mid \text{ungreedy} \mid \{\text{newline, NLSpec}\} \mid \text{bsr_anycrlf} \mid \text{bsr_unicode}
- \text{NLSpec} = \text{cr} \mid \text{crlf} \mid \text{lf} \mid \text{anycrlf} \mid \text{any}
- \text{MP} = \text{mp()}
- \text{ErrSpec} = \{\text{ErrString, Position}\}
- \text{ErrString} = \text{string()}
- \text{Position} = \text{int()}

This function compiles a regular expression with the syntax described below into an internal format to be used later as a parameter to the \text{run/2,3} functions.

Compiling the regular expression before matching is useful if the same expression is to be used in matching against multiple subjects during the program’s lifetime. Compiling once and executing many times is far more efficient than compiling each time one wants to match.

When the unicode option is given, the regular expression should be given as a valid unicode \text{charlist()}, otherwise as any valid \text{iodata()}.

The options have the following meanings:

- **unicode**: The regular expression is given as a unicode \text{charlist()} and the resulting regular expression code is to be run against a valid unicode \text{charlist()} subject.
- **anchored**: The pattern is forced to be “anchored”, that is, it is constrained to match only at the first matching point in the string that is being searched (the “subject string”). This effect can also be achieved by appropriate constructs in the pattern itself.
- **caseless**: Letters in the pattern match both upper and lower case letters. It is equivalent to Perl’s /i option, and it can be changed within a pattern by a (?i) option setting. Uppercase and lowercase letters are defined as in the ISO-8859-1 character set.
- **dollar_endonly**: A dollar metacharacter in the pattern matches only at the end of the subject string. Without this option, a dollar also matches immediately before a newline at the end of the string (but not before any other newlines). The dollar_endonly option is ignored if multiline is given. There is no equivalent option in Perl, and no way to set it within a pattern.
dotall A dot maturate in the pattern matches all characters, including those that indicate newline. Without it, a dot does not match when the current position is at a newline. This option is equivalent to Perl’s /s option, and it can be changed within a pattern by a (?s) option setting. A negative class such as [^a] always matches newline characters, independent of the setting of this option.

extended Whitespace data characters in the pattern are ignored except when escaped or inside a character class. Whitespace does not include the VT character (ASCII 11). In addition, characters between an unescaped # outside a character class and the next newline, inclusive, are also ignored. This is equivalent to Perl’s /x option, and it can be changed within a pattern by a (?x) option setting. This option makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. Whitespace characters may never appear within special character sequences in a pattern, for example within the sequence (? which introduces a conditional subpattern.

firstline An unanchored pattern is required to match before or at the first newline in the subject string, though the matched text may continue over the newline.

multiline By default, PCRE treats the subject string as consisting of a single line of characters (even if it actually contains newlines). The “start of line” metacharacter (^) matches only at the start of the string, while the “end of line” metacharacter ($) matches only at the end of the string, or before a terminating newline (unless dollar_endonly is given). This is the same as Perl. When multiline is given, the “start of line” and “end of line” constructs match immediately following or immediately before internal newlines in the subject string, respectively, as well as at the very start and end. This is equivalent to Perl’s /m option, and it can be changed within a pattern by a (?m) option setting. If there are no newlines in a subject string, or no occurrences of ^ or $ in a pattern, setting multiline has no effect.

no_auto_capture Disables the use of numbered capturing parentheses in the pattern. Any opening parenthesis that is not followed by ? behaves as if it were followed by ?; but named parentheses can still be used for capturing (and they acquire numbers in the usual way). There is no equivalent of this option in Perl.

dupnames Names used to identify capturing subpatterns need not be unique. This can be helpful for certain types of pattern when it is known that only one instance of the named subpattern can ever be matched. There are more details of named subpatterns below.

ungreedy This option inverts the “greediness” of the quantifiers so that they are not greedy by default, but become greedy if followed by ?. It is not compatible with Perl. It can also be set by a (?U) option setting within the pattern.

{newline, NLSpec} Override the default definition of a newline in the subject string, which is LF (ASCII 10) in Erlang.

cr Newline is indicated by a single character CR (ASCII 13)
lf Newline is indicated by a single character LF (ASCII 10), the default
crlf Newline is indicated by the two-character CRLF (ASCII 13 followed by ASCII 10) sequence
anycrlf Any of the three preceding sequences should be recognized.
anyc Newline sequences above, plus the Unicode sequences VT (vertical tab, U+0008), FF (formfeed, U+000C), NEL (next line, U+0085), LS (line separator, U+2028), and PS (paragraph separator, U+2029).

bsr_anycrlf Specifies specifically that \R is to match only the cr, lf or crlf sequences, not the Unicode specific newline characters.
bsr_unicode Specifies specifically that \R is to match all the Unicode newline
characters (including crlf etc, the default).

run(Subject, RE) -> {match, Captured} | nomatch

Types:
- Subject = iodata() | charlist()
- RE = mp() | iodata() | charlist()
- Captured = [ CaptureData ]
- CaptureData = [int(),int()]

The same as run(Subject, RE, []).

run(Subject, RE, Options) -> {match, Captured} | match | nomatch

Types:
- Subject = iodata() | charlist()
- RE = mp() | iodata() | charlist()
- Options = [Option]
- Option = anchored | global | notbol | noteol | notempty | {offset, int()} | {newline, NLSpec} | bsr_anycrlf | bsr_unicode | {capture, ValueSpec} | {capture, ValueSpec, Type} | CompileOpt
- Type = index | list | binary
- ValueSpec = all | all but first | first | ValueList
- ValueList = [ ValueID ]
- ValueID = int() | string() | atom()
- CompileOpt = see compile/2 above
- NLSpec = cr | crlf | lf | anycril | any
- Captured = [ CaptureData ] | [ [ CaptureData ] ... ]
- CaptureData = [int(),int()] | ListConversionData | binary()
- ListConversionData = string() | {error, string(), binary()} | {incomplete, string(),
  binary()}

Executes a regexp matching, returning match/{match, Captured} or nomatch. The
regular expression can be given either as iodata() in which case it is automatically
compiled (as by re:compile/2) and executed, or as a pre compiled mp() in which case
it is executed against the subject directly.

When compilation is involved, the exception badarg is thrown if a compilation error
occurs. To locate the error in the regular expression, use the function re:compile/2 to
get more information.

If the regular expression is previously compiled, the option list can only contain the
options anchored, global, notbol, noteol, notempty, {offset, int()}, {newline,
NLSpec} and {capture, ValueSpec}/ {capture, ValueSpec, Type}. Otherwise all
options valid for the re:compile/2 function are allowed as well. Options allowed both
for compilation and execution of a match, namely anchored and {newline, NLSpec},
will affect both the compilation and execution if present together with a non
pre-compiled regular expression.

If the regular expression was previously compiled with the option unicode, the
Subject should be provided as a valid Unicode charlist(), otherwise any iodata()
will do. If compilation is involved and the option unicode is given, both the Subject and the regular expression should be given as valid Unicode charlists().

The {capture, ValueSpec} / {capture, ValueSpec, Type} defines what to return from the function upon successful matching. The capture tuple may contain both a value specification telling which of the captured substrings are to be returned, and a type specification, telling how captured substrings are to be returned (as index tuples, lists or binaries). The capture option makes the function quite flexible and powerful.

The different options are described in detail below.

If the capture options describe that no substring capturing at all is to be done (({capture, none}), the function will return the single atom match upon successful matching, otherwise the tuple {match, ValueList} is returned. Disabling capturing can be done either by specifying none or an empty list as ValueSpec.

A description of all the options relevant for execution follows:

**anchored** Limits re:run/3 to matching at the first matching position. If a pattern was compiled with anchored, or turned out to be anchored by virtue of its contents, it cannot be made unanchored at matching time, hence there is no unanchored option.

**global** Implements global (repetitive) search as the g flag in i.e. Perl. Each match found is returned as a separate list() containing the specific match as well as any matching subexpressions (or as specified by the capture option). The Captured part of the return value will hence be a list() of list()’s when this option is given.

When the regular expression matches an empty string, the behaviour might seem non-intuitive, why the behaviour requites some clarifying. With the global option, re:run/3 handles empty matches in the same way as Perl, meaning that a match at any point giving an empty string (with length 0) will be retried with the options [anchored, notempty] as well. If that search gives a result of length > 0, the result is included. An example:

```
re:run("cat","(|at)",[global]).
```

The matching will be performed as following:

**At offset 0** The regexp (|at) will first match at the initial position of the string cat, giving the result set [[0,0],[0,0]] (the second [0,0] is due to the subexpression marked by the parentheses). As the length of the match is 0, we don’t advance to the next position yet.

**At offset 0 with [anchored, notempty]** The search is retried with the options [anchored, notempty] at the same position, which does not give any interesting result of longer length, why the search position is now advanced to the next character (a).

**At offset 1** Now the search results in [[1,0],[1,0]] meaning this search will also be repeated with the extra options.

**At offset 1 with [anchored, notempty]** Now the ab alternative is found and the result will be [[1,2],[1,2]]. The result is added to the list of results and the position in the search string is advanced two steps.

**At offset 3** The search now once again matches the empty string, giving [[3,0],[3,0]].

**At offset 1 with [anchored, notempty]** This will give no result of length > 0 and we are at the last position, so the global search is complete.

The result of the call is:
notempty

An empty string is not considered to be a valid match if this option is given.
If there are alternatives in the pattern, they are tried. If all the alternatives match
the empty string, the entire match fails. For example, if the pattern

```
a?b?
```

is applied to a string not beginning with "a" or "b", it matches the empty string at
the start of the subject. With notempty given, this match is not valid, so re:run/3
searches further into the string for occurrences of "a" or "b".
Perl has no direct equivalent of notempty, but it does make a special case of a
pattern match of the empty string within its split() function, and when using the /g
modifier. It is possible to emulate Perl's behavior after matching a null string by
first trying the match again at the same offset with notempty and anchored, and
then if that fails by advancing the starting offset (see below) and trying an ordinary
match again.

notbol

This option specifies that the first character of the subject string is not the
beginning of a line, so the circumflex metacharacter should not match before it.
Setting this without multiline (at compile time) causes circumflex never to
match. This option affects only the behavior of the circumflex metacharacter. It
does not affect \A.

noteol

This option specifies that the end of the subject string is not the end of a line,
so the dollar metacharacter should not match it nor (except in multiline mode) a
newline immediately before it. Setting this without multiline (at compile time)
causes dollar never to match. This option affects only the behavior of the dollar
metacharacter. It does not affect \Z or \z.

{offset, int()} Start matching at the offset (position) given in the subject string.
The offset is zero-based, so that the default is {offset, 0} (all of the subject
string).

{newline, NLSpec} Override the default definition of a newline in the subject string,
which is LF (ASCII 10) in Erlang.

\cr Newline is indicated by a single character CR (ASCII 13)
\lf Newline is indicated by a single character LF (ASCII 10), the default
\crlf Newline is indicated by the two-character CRLF (ASCII 13 followed by
ASCII 10) sequence.
\any\crlf Any of the three preceding sequences should be recognized.
\any Any of the newline sequences above, plus the Unicode sequences VT (vertical
\tab, U+0008), FF (formfeed, U+000C), NEL (next line, U+0085), LS (line
separator, U+2028), and PS (paragraph separator, U+2029).
\bsr_\any\crlf Specifies specifically that \R is to match only the cr, lf or crlf sequences,
not the Unicode specific newline characters.(overrides compilation option)
\bsr_unidece Specifies specifically that \R is to match all the Unicode newline
characters (including crlf etc, the default).(overrides compilation option)

{capture, ValueSpec}|{capture, ValueSpec, Type} Specifies which captured
substrings are returned and in what format. By default, re:run/3 captures all of
the matching part of the substring as well as all capturing subpatterns (all of the
pattern is automatically captured). The default return type is (zero-based) indexes
of the captured parts of the string, given as {Offset,Length} pairs (the index
Type of capturing).
As an example of the default behavior, the following call:
re:run("ABCabcdABC","abcd",[]).

returns, as first and only captured string the matching part of the subject ("abcd" in the middle) as a index pair \(\{3,4\}\), where character positions are zero based, just as in offsets. The return value of the call above would then be:

\{match,\{3,4\}\}

Another (and quite common) case is where the regular expression matches all of the subject, as in:

re:run("ABCabcdABC", ".*abcd.*", []).

where the return value correspondingly will point out all of the string, beginning at index 0 and being 10 characters long:

\{match,\{0,10\}\}

If the regular expression contains capturing subpatterns, like in the following case:

re:run("ABCabcdABC", ".*\(abcd\).*", []).

all of the matched subject is captured, as well as the captured substrings:

\{match,\{0,10\},\{3,4\}\}

the complete matching pattern always giving the first return value in the list and the rest of the subpatterns being added in the order they occurred in the regular expression.

The capture tuple is built up as follows:

ValueSpec Specifies which captured (sub)patterns are to be returned. The ValueSpec can either be an atom describing a predefined set of return values, or a list containing either the indexes or the names of specific subpatterns to return.

The predefined sets of subpatterns are:

all All captured subpatterns including the complete matching string. This is the default.

first Only the first captured subpattern, which is always the complete matching part of the subject. All explicitly captured subpatterns are discarded.

all_but_first All but the first matching subpattern, i.e. all explicitly captured subpatterns, but not the complete matching part of the subject string. This is useful if the regular expression as a whole matches a large part of the subject, but the part you’re interested in is in an explicitly captured subpattern. If the return type is list or binary, not returning subpatterns you’re not interested in is a good way to optimize.

none Do not return matching subpatterns at all, yielding the single atom match as the return value of the function when matching successfully instead of the \{match, list()\} return. Specifying an empty list gives the same behavior.

The value list is a list of indexes for the subpatterns to return, where index 0 is for all of the pattern, and 1 is for the first explicit capturing subpattern in the regular expression, and so forth. When using named captured subpatterns (see below) in the regular expression, one can use atom()’s or string()’s to specify the subpatterns to be returned. This deserves an example, consider the following regular expression:
matched against the string "ABCabcdABC", capturing only the "abcd" part (the first explicit subpattern):

```erlang
re:run("ABCabcdABC",".*(abcd).*",[{capture,[1]}]).
```

The call will yield the following result:

```erlang
{match,[[3,4]]}
```

as the first explicitly captured subpattern is "(abcd)", matching "abcd" in the subject, at (zero-based) position 3, of length 4.

Now consider the same regular expression, but with the subpattern explicitly named 'FOO':

```
.*(<?FOO>abcd).*
```

With this expression, we could still give the index of the subpattern with the following call:

```erlang
re:run("ABCabcdABC",".*(<?FOO>abcd).*",[{capture,[1]}]).
```

giving the same result as before. But as the subpattern is named, we can also give its name in the value list:

```erlang
re:run("ABCabcdABC",".*(<?FOO>abcd).*",[{capture,'FOO'}]).
```

which would yield the same result as the earlier examples, namely:

```erlang
{match,[[3,4]]}
```

The values list might specify indexes or names not present in the regular expression, in which case the return values vary depending on the type. If the type is index, the tuple \(-1,0\) is returned for values having no corresponding subpattern in the regexp, but for the other types (binary and list), the values are the empty binary or list respectively.

**Type**

Optionally specifies how captured substrings are to be returned. If omitted, the default of index is used. The **Type** can be one of the following:

- **index** Return captured substrings as pairs of byte indexes into the subject string and length of the matching string in the subject (as if the subject string was flattened with iolist_to_binary/1 or unicode:characters_to_binary/2 prior to matching). Note that the unicode option results in byte-oriented indexes in a (possibly imagined) UTF-8 encoded binary. A byte index tuple \(0,2\) might therefore represent one or two characters when unicode is in effect. This might seem contra-intuitive, but has been deemed the most effective and useful way to do it. To return lists instead might result in simpler code if that is desired. This return type is the default.

- **list** Return matching substrings as lists of characters (Erlang string()'s). It the unicode option is used in combination with the \(\backslash C\) sequence in the regular expression, a captured subpattern can contain bytes that has is not valid UTF-8 (\(\backslash C\) matches bytes regardless of character encoding). In that case the list capturing may result in the same types of tuples that unicode:characters_to_list/2 can return, namely three-tuples with the tag incomplete or error, the successfully converted characters and the invalid UTF-8 tail of the conversion as a binary. The best strategy is to avoid using the\(\backslash C\) sequence when capturing lists.

- **binary** Return matching substrings as binaries. If the unicode option is used, these binaries is in UTF-8. If the \(\backslash C\) sequence is used together with unicode the binaries may be invalid UTF-8.
In general, subpatterns that got assigned no value in the match are returned as the tuple \([-1, 0]\) when \texttt{type} is \texttt{index}. Unassigned subpatterns are returned as the empty binary or list respectively for other return types. Consider the regular expression:

\[ \cdot((?<\text{FOO}\text{abdd})\mid a(.d))\cdot \]

There are three explicitly capturing subpatterns, where the opening parenthesis position determines the order in the result, hence \([(?<\text{FOO}\text{abdd})\mid a(.d))\] is subpattern index 1, \((?<\text{FOO}\text{abdd})\) is subpattern index 2 and \((.d)\) is subpattern index 3. When matched against the following string:

"ABCabcdABC"

the subpattern at index 2 won’t match, as “abdd” is not present in the string, but the complete pattern matches (due to the alternative a(.d). The subpattern at index 2 is therefore unassigned and the default return value will be:

\{match,\{[0,10],[3,4],[-1,0],[4,3]\}\}

Setting the capture \texttt{Type} to \texttt{binary} would give the following:

\{match,\{<<ABCabcdABC>>,<<"abcd">>,<<>>,<<"bcd">>\}\}

where the empty binary \(<<<>>\) represents the unassigned subpattern. In the binary case, some information about the matching is therefore lost, the \(<<<>>\) might just as well be an empty string captured.

If differentiation between empty matches and non-existing subpatterns is necessary, use the \texttt{type index} and do the conversion to the final type in Erlang code.

When the option \texttt{global} is given, the capture specification affects each match separately, so that:

\[
\text{re:run("cacb","c(a|b)",[global,{capture,[1],list}]).}
\]

gives the result:

\{match,\["a"],["b"]\}\]

The options solely affecting the compilation step are described in the \texttt{re:compile/2} function.
Option = anchored | global | notbol | noteol | notempty | \{offset, int()\} | \{newline, NLSpec\} | bsr|anycrlf | bsr|unicode | \{return, ReturnType\} | CompileOpt
ReturnType = iodata | list | binary
CompileOpt = see compile/2 above
NLSpec = cr | crlf | lf | anycrlf | any

Replaces the matched part of the Subject string with the content of Replacement. Options are given as to the re:run/3 function except that the capture option of re:run/3 is not allowed. Instead a \{return, ReturnType\} is present. The default return type is iodata, constructed in a way to minimize copying. The iodata result can be used directly in many i/o-operations. If a flat list() is desired, specify \{return, list\} and if a binary is preferred, specify \{return, binary\}.

As in the re:run/3 function, an mp() compiled with the unicode option requires the Subject to be a Unicode charlist(). If compilation is done implicitly and the unicode compilation option is given to this function, both the regular expression and the Subject should be given as valid Unicode charlist()'s.

The replacement string can contain the special character &, which inserts the whole matching expression in the result, and the special sequence \N (where N is an integer > 0), resulting in the subexpression number N will be inserted in the result. If no subexpression with that number is generated by the regular expression, nothing is inserted.

To insert an & or \ in the result, precede it with a \. Note that Erlang already gives a special meaning to \ in literal strings, why a single \ has to be written as "\\" and therefore a double \ as "\\\\". Example:

```
re:replace("abcd","c","[&]",\{"return,list\}).
```
gives

"ab[c]d"

while

```
re:replace("abcd","c","[\\&]",\{"return,list\}).
```
gives

"ab[&]d"

As with re:run/3, compilation errors raise the badarg exception, re:compile/2 can be used to get more information about the error.

**split(Subject,RE) -> SplitList**

Types:
- Subject = iodata() | charlist()
- RE = mp() | iodata()
- SplitList = [ iodata() | charlist() ]

The same as split(Subject,RE,[]).

**split(Subject,RE,Options) -> SplitList**

Types:
- Subject = iodata() | charlist()
- RE = mp() | iodata() | charlist()
This function splits the input into parts by finding tokens according to the regular expression supplied.

The splitting is done basically by running a global regexp match and dividing the initial string wherever a match occurs. The matching part of the string is removed from the output.

As in the re:run/3 function, an mp() compiled with the unicode option requires the Subject to be a Unicode charlist(). If compilation is done implicitly and the unicode compilation option is given to this function, both the regular expression and the Subject should be given as valid Unicode charlist()'s.

The result is given as a list of “strings”, the preferred datatype given in the return option (default iodata).

If subexpressions are given in the regular expression, the matching subexpressions are returned in the resulting list as well. An example:

re:split("Erlang","[ln]",[{return,list}]).

will yield the result:

["Er","a","g"]

while

re:split("Erlang","([ln])",[{return,list}]).

will yield

["Er","l","a","n","g"]

The text matching the subexpression (marked by the parantheses in the regexp) is inserted in the result list where it was found. In effect this means that concatenating the result of a split where the whole regexp is a single subexpression (as in the example above) will always result in the original string.

As there is no matching subexpression for the last part in the example (the “g”), there is nothing inserted after that. To make the group of strings and the parts matching the subexpressions more obvious, one might use the group option, which groups together the part of the subject string with the parts matching the subexpressions when the string was split:

re:split("Erlang","([ln])",[{return,list},group]).

gives:

[["Er","l"],["a","n"],["g"]]

Options = [ Option ]
Option = anchored | global | notbol | noteol | notempty | {offset, int()} |
{newline, NLSpec} | bsr anycrlf | bsr unicode | {return, ReturnType} | {parts, NumParts} | group | trim | CompileOpt
NumParts = int() | infinity
ReturnType = iodata | list | binary
CompileOpt = see compile/2 above
NLSpec = cr | crlf | if | anycrlf | any
SplitList = [ RetData ] | [ GroupedRetData ]
GroupedRetData = [ RetData ]
RetData = iodata() charlist() | binary() | list()
Here the regular expression matched first the “l”, causing “Er” to be the first part in the result. When the regular expression matched, the (only) subexpression was bound to the “l”, why the “l” is inserted in the group together with “Er”. The next match is of the “n”, making “a” the next part to be returned. As the subexpression is bound to the substring “n” in this case, the “n” is inserted into this group. The last group consists of the rest of the string, as no more matches are found.

By default, all parts of the string, including the empty strings are returned from the function. As an example:

\[
\text{re:split("Erlang","[lg]",\{return,list\})}.
\]

The result will be:

\[
["Er","an",[]]
\]

as the matching of the “g” in the end of the string leaves an empty rest which is also returned. This behaviour differs from the default behaviour of the split function in Perl, where empty strings at the end are by default removed. To get the “trimming” default behavior of Perl, specify \text{trim} as an option:

\[
\text{re:split("Erlang","[lg]",\{return,list,trim\})}.
\]

The result will be:

\[
["Er","an"]
\]

The “trim” option in effect says; “give me as many parts as possible except the empty ones”, which might be useful in some circumstances. You can also specify how many parts you want, by specifying \{parts,N\}:

\[
\text{re:split("Erlang","[lg]",\{return,list,\{parts,2\}\})}.
\]

This will give:

\[
["Er","ang"]
\]

Note that the last part is “ang”, not “an”, as we only specified splitting into two parts, and the splitting stops when enough parts are given, why the result differs from that of \text{trim}.

More than three parts are not possible with this indata, why

\[
\text{re:split("Erlang","[lg]",\{return,list,\{parts,4\}\})}.
\]

will give the same result as the default, which is to be viewed as “an infinite number of parts”.

Specifying 0 as the number of parts gives the same effect as the option \text{trim}. If subexpressions are captured, empty subexpression matches at the end are also stripped from the result if \text{trim} or \{parts,0\} is specified.

If you are familiar with Perl, the \text{trim} behaviour corresponds exactly to the Perl default, the \{parts,N\} where N is a positive integer corresponds exactly to the Perl behaviour with a positive numerical third parameter and the default behaviour of \text{re:split/3} corresponds to that when the Perl routine is given a negative integer as the third parameter.

Summary of options not previously described for the \text{re:run/3} function:

\{\text{return},\text{ReturnType}\}  \text{Specifies how the parts of the original string are presented in the result list. The possible types are:}
iodata The variant of iodata() that gives the least copying of data with the current implementation (often a binary, but don’t depend on it).

binary All parts returned as binaries.
list All parts returned as lists of characters (“strings”).

group Groups together the part of the string with the parts of the string matching the subexpressions of the regexp.
   The return value from the function will in this case be a list() of list()’s. Each sublist begins with the string picked out of the subject string, followed by the parts matching each of the subexpressions in order of occurrence in the regular expression.

{parts,N} Specifies the number of parts the subject string is to be split into.
   The number of parts should be a positive integer for a specific maximum on the number of parts and infinity for the maximum number of parts possible (the default). Specifying {parts,0} gives as many parts as possible disregarding empty parts at the end, the same as specifying trim

trim Specifies that empty parts at the end of the result list are to be disregarded. The same as specifying {parts,0}. This corresponds to the default behaviour of the split built in function in Perl.

PERL LIKE REGULAR EXPRESSIONS SYNTAX

The following sections contain reference material for the regular expressions used by this module. The regular expression reference is taken from the PCRE documentation, but converted as needed.
The documentation is altered where appropriate and where the re module behaves differently than the PCRE library.

PCRE regular expression details

The syntax and semantics of the regular expressions that are supported by PCRE are described in detail below. Perl’s regular expressions are described in its own documentation, and regular expressions in general are covered in a number of books, some of which have copious examples. Jeffrey Friedl’s “Mastering Regular Expressions”, published by O’Reilly, covers regular expressions in great detail. This description of PCRE’s regular expressions is intended as reference material.
The reference material is divided into the following sections:

- Newline conventions [page 358]
- Characters and metacharacters [page 359]
- Backslash [page 360]
- Circumflex and dollar [page 369]
- Full stop (period, dot) [page 369]
- Matching a single byte [page 370]
- Square brackets and character classes [page 370]
- Posix character classes [page 371]
- Vertical bar [page 372]
Newline conventions

PCRE supports five different conventions for indicating line breaks in strings: a single CR (carriage return) character, a single LF (linefeed) character, the two-character sequence CRLF, any of the three preceding, or any Unicode newline sequence.

It is also possible to specify a newline convention by starting a pattern string with one of the following five sequences:

(*CR) carriage return
(*LF) linefeed
(*CRLF) carriage return, followed by linefeed
(*ANYCRLF) any of the three above
(*ANY) all Unicode newline sequences

These override the default and the options given to `re:compile/2`. For example, the pattern:

(*CR)a.b

changes the convention to CR. That pattern matches “a\nb” because LF is no longer a newline. Note that these special settings, which are not Perl-compatible, are recognized only at the very start of a pattern, and that they must be in upper case. If more than one of them is present, the last one is used.

The newline convention does not affect what the \R escape sequence matches. By default, this is any Unicode newline sequence, for Perl compatibility. However, this can be changed; see the description of \R in the section entitled “Newline sequences” below. A change of \R setting can be combined with a change of newline convention.
Characters and metacharacters

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

```
The quick brown fox
```

matches a portion of a subject string that is identical to itself. When caseless matching is specified (the `caseless` option), letters are matched independently of case.

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of metacharacters, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of metacharacters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized within square brackets. Outside square brackets, the metacharacters are as follows:

- `\` general escape character with several uses
- `^` assert start of string (or line, in multiline mode)
- `$` assert end of string (or line, in multiline mode)
- `.` match any character except newline (by default)
- `[` start character class definition
- `|` start of alternative branch
- `{` start subpattern
- `}` end subpattern
- `?` extends the meaning of `, also 0 or 1 quantifier, also quantifier minimizer
- `*` 0 or more quantifier
- `+` 1 or more quantifier, also "possessive quantifier"
- `/` start min/max quantifier

Part of a pattern that is in square brackets is called a “character class”. In a character class the only metacharacters are:

- `\` general escape character
- `^` negate the class, but only if the first character
- `-` indicates character range
- `[` POSIX character class (only if followed by POSIX syntax)
- `]` terminates the character class

The following sections describe the use of each of the metacharacters.
Backslash

The backslash character has several uses. Firstly, if it is followed by a non-alphanumeric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a * character, you write \* in the pattern. This escaping action applies whether or not the following character would otherwise be interpreted as a metacharacter, so it is always safe to precede a non-alphanumeric with backslash to specify that it stands for itself. In particular, if you want to match a backslash, you write \\
.

If a pattern is compiled with the extended option, whitespace in the pattern (other than in a character class) and characters between a # outside a character class and the next newline are ignored. An escaping backslash can be used to include a whitespace or # character as part of the pattern.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between \Q and \E. This is different from Perl in that $ and @ are handled as literals in \Q...\E sequences in PCRE, whereas in Perl, $ and @ cause variable interpolation. Note the following examples:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>PCRE matches</th>
<th>Perl matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>\Qabc$xyz\E</td>
<td>abc$xyz</td>
<td>abc followed by the contents of $xyz</td>
</tr>
<tr>
<td>\Qabc\$xyz\E</td>
<td>abc$xyz</td>
<td>abc$xyz</td>
</tr>
<tr>
<td>\Qabc\E$\Qxyz\E</td>
<td>abc$xyz</td>
<td>abc$xyz</td>
</tr>
</tbody>
</table>

The \Q...\E sequence is recognized both inside and outside character classes.

Non-printing characters

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

\a alarm, that is, the BEL character (hex 07)
\cx “control-x”, where x is any character
\e escape (hex 1B)
\f formfeed (hex 0C)
\n linefeed (hex 0A)
\r carriage return (hex 0D)
\t tab (hex 09)
\ddd character with octal code ddd, or backreference
\xhh character with hex code hh
\x{hhh..} character with hex code hhh..
The precise effect of \cx is as follows: if x is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus \c2 becomes hex 1A, but \c1 becomes hex 3B, while \c; becomes hex 7B.

After \cx, from zero to two hexadecimal digits are read (letters can be in upper or lower case). Any number of hexadecimal digits may appear between \x{ and }, but the value of the character code must be less than 256 in non-UTF-8 mode, and less than 2**31 in UTF-8 mode. That is, the maximum value in hexadecimal is 7FFFFFF. Note that this is bigger than the largest Unicode code point, which is 10FFFF.

If characters other than hexadecimal digits appear between \x{ and }, or if there is no terminating }, this form of escape is not recognized. Instead, the initial \x will be interpreted as a basic hexadecimal escape, with no following digits, giving a character whose value is zero.

Characters whose value is less than 256 can be defined by either of the two syntaxes for \x. There is no difference in the way they are handled. For example, \xdc is exactly the same as \x{dc}.

After \O up to two further octal digits are read. If there are fewer than two digits, just those that are present are used. Thus the sequence \O\x07 specifies two binary zeros followed by a BEL character (code value 7). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a back reference. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and uses them to generate a data character. Any subsequent digits stand for themselves. The value of a character specified in octal must be less than \400. In non-UTF-8 mode, the value of a character specified in octal must be less than \400. In UTF-8 mode, values up to \777 are permitted. For example:

- \040 is another way of writing a space
- \40 is the same, provided there are fewer than 40 previous capturing subpatterns
- \7 is always a back reference
- \11 might be a back reference, or another way of writing a tab
- \011 is always a tab
- \0113 is a tab followed by the character “3”
- \113 might be a back reference, otherwise the character with octal code 113
- \377 might be a back reference, otherwise the byte consisting entirely of 1 bits
- \81 is either a back reference, or a binary zero followed by the two characters “8” and “1”

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single character value can be used both inside and outside character classes. In addition, inside a character class, the sequence \b is interpreted as the backspace character (hex 08), and the sequences \R and \X are...
interpreted as the characters “R” and “X”, respectively. Outside a character class, these sequences have different meanings (see below).

Absolute and relative back references

The sequence \g followed by an unsigned or a negative number, optionally enclosed in braces, is an absolute or relative back reference. A named back reference can be coded as \g{name}. Back references are discussed later, following the discussion of parenthesized subpatterns.

Generic character types

Another use of backslash is for specifying generic character types. The following are always recognized:

\d any decimal digit
\D any character that is not a decimal digit
\h any horizontal whitespace character
\H any character that is not a horizontal whitespace character
\s any whitespace character
\S any character that is not a whitespace character
\v any vertical whitespace character
\V any character that is not a vertical whitespace character
\w any “word” character
\W any “non-word” character

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

For compatibility with Perl, \s does not match the VT character (code 11). This makes it different from the the POSIX “space” class. The \s characters are HT (9), LF (10), FF (12), CR (13), and space (32). If “use locale;” is included in a Perl script, \s may match the VT character. In PCRE, it never does.

In UTF-8 mode, characters with values greater than 128 never match \d, \s, or \w, and always match \D, \S, and \W. This is true even when Unicode character property support is available. These sequences retain their original meanings from before UTF-8 support was available, mainly for efficiency reasons.

The sequences \h, \H, \v, and \V are Perl 5.10 features. In contrast to the other sequences, these do match certain high-valued codepoints in UTF-8 mode. The horizontal space characters are:

U+0009 Horizontal tab
U+0020 Space
U+00A0 Non-break space
U+1680 Ogham space mark
U+180E Mongolian vowel separator
U+2000 En quad
U+2001 Em quad
STDLIB Reference Manual

U+2002 En space
U+2003 Em space
U+2004 Three-per-em space
U+2005 Four-per-em space
U+2006 Six-per-em space
U+2007 Figure space
U+2008 Punctuation space
U+2009 Thin space
U+200A Hair space
U+202F Narrow no-break space
U+205F Medium mathematical space
U+3000 Ideographic space

The vertical space characters are:

U+000A Linefeed
U+000B Vertical tab
U+000C Formfeed
U+000D Carriage return
U+0085 Next line
U+2028 Line separator
U+2029 Paragraph separator

A “word” character is an underscore or any character less than 256 that is a letter or
digit. The definition of letters and digits is controlled by PCRE’s low-valued character
tables, which are always ISO-8859-1.

Newline sequences

Outside a character class, by default, the escape sequence \R matches any Unicode
newline sequence. This is a Perl 5.10 feature. In non-UTF-8 mode \R is equivalent to
the following:

(?<\n|\x0b|\r|\x85)

This is an example of an “atomic group”, details of which are given below.

This particular group matches either the two-character sequence CR followed by LF, or
one of the single characters LF (linefeed, U+000A), VT (vertical tab, U+000B), FF
(formfeed, U+000C), CR (carriage return, U+000D), or NEL (next line, U+0085). The
two-character sequence is treated as a single unit that cannot be split.

In UTF-8 mode, two additional characters whose codepoints are greater than 255 are
added: LS (line separator, U+2028) and PS (paragraph separator, U+2029). Unicode
character property support is not needed for these characters to be recognized.

It is possible to restrict \R to match only CR, LF, or CRLF (instead of the complete set
of Unicode line endings) by setting the option bsr_anycrlf either at compile time or
when the pattern is matched. (BSR is an abbreviation for “backslash R.”) This can be
made the default when PCRE is built; if this is the case, the other behaviour can be
requested via the bsr_unidec option. It is also possible to specify these settings by
starting a pattern string with one of the following sequences:
(*BSR_ANYCRLF) CR, LF, or CRLF only (*BSR_UNICODE) any Unicode newline sequence

These override the default and the options given to \texttt{re:compile/2}, but they can be overridden by options given to \texttt{re:run/3}. Note that these special settings, which are not Perl-compatible, are recognized only at the very start of a pattern, and that they must be in upper case. If more than one of them is present, the last one is used. They can be combined with a change of newline convention, for example, a pattern can start with:

(*ANY)(*BSR_ANYCRLF)

Inside a character class, \texttt{\textbackslash R} matches the letter “R”.

Unicode character properties

When PCRE is built with Unicode character property support, three additional escape sequences that match characters with specific properties are available. When not in UTF-8 mode, these sequences are of course limited to testing characters whose codepoints are less than 256, but they do work in this mode. The extra escape sequences are:

\texttt{\textbackslash p(xx)} a character with the \texttt{xx} property \texttt{\textbackslash P(xx)} a character without the \texttt{xx} property \texttt{\textbackslash X} an extended Unicode sequence

The property names represented by \texttt{xx} above are limited to the Unicode script names, the general category properties, and “Any”, which matches any character (including newline). Other properties such as “InMusicalSymbols” are not currently supported by PCRE. Note that \texttt{\textbackslash P(Any)} does not match any characters, so always causes a match failure.

Sets of Unicode characters are defined as belonging to certain scripts. A character from one of these sets can be matched using a script name. For example:

\texttt{\textbackslash p(Greek)} \texttt{\textbackslash P(Han)}

Those that are not part of an identified script are lumped together as “Common”. The current list of scripts is:

- Arabic
- Armenian
- Balinese
- Bengali
- Bopomofo
- Braille
- Buginese
- Buhid
- Canadian_Aboriginal
- Cherokee
- Common
- Coptic
- Cuneiform
- Cypriot
- Cyrillic
- Deseret
- Devanagari
• Ethiopic
• Georgian
• Glagolitic
• Gothic
• Greek
• Gujarati
• Gurmukhi
• Han
• Hangul
• Hanunoo
• Hebrew
• Hiragana
• Inherited
• Kannada
• Katakana
• Kharoshthi
• Khmer
• Lao
• Latin
• Limbu
• Linear_B
• Malayalam
• Mongolian
• Myanmar
• New_Tai_Lue
• Nko
• Ogham
• Old_Italic
• Old_Persian
• Oriya
• Osmanya
• Phags_Pa
• Phoenician
• Runic
• Shavian
• Sinhala
• Syloti_Nagri
• Syriac
• Tagalog
• Tagbanwa
• Tai_Le
Each character has exactly one general category property, specified by a two-letter abbreviation. For compatibility with Perl, negation can be specified by including a circumflex between the opening brace and the property name. For example, \{^Lu\} is the same as \{L\}.

If only one letter is specified with \p or \P, it includes all the general category properties that start with that letter. In this case, in the absence of negation, the curly brackets in the escape sequence are optional; these two examples have the same effect:

- \{L\}
- \{pL\}

The following general category property codes are supported:

- C Other
- Cc Control
- Cf Format
- Cn Unassigned
- Co Private use
- Cs Surrogate
- L Letter
- LI Lower case letter
- Lm Modifier letter
- Lo Other letter
- Lt Title case letter
- Lu Upper case letter
- M Mark
- Mc Spacing mark
- Me Enclosing mark
- Mn Non-spacing mark
- N Number
- Nd Decimal number
- NI Letter number
- No Other number
- P Punctuation
Pc  Connector punctuation
Pd  Dash punctuation
Pe  Close punctuation
Pf  Final punctuation
Pi  Initial punctuation
Po  Other punctuation
Ps  Open punctuation
S   Symbol
Sc  Currency symbol
Sk  Modifier symbol
Sm  Mathematical symbol
So  Other symbol
Z   Separator
Zl  Line separator
Zp  Paragraph separator
Zs  Space separator

The special property L& is also supported: it matches a character that has the Lu, Ll, or Lt property, in other words, a letter that is not classified as a modifier or “other”.
The Cs (Surrogate) property applies only to characters in the range U+D800 to U+DFFF. Such characters are not valid in UTF-8 strings (see RFC 3629) and so cannot be tested by PCRE, unless UTF-8 validity checking has been turned off (see the discussion of noutf8check in the pcreapi page).
The long synonyms for these properties that Perl supports (such as \p{Letter}) are not supported by PCRE, nor is it permitted to prefix any of these properties with “Is”. No character that is in the Unicode table has the Cn (unassigned) property. Instead, this property is assumed for any code point that is not in the Unicode table.
Specifying caseless matching does not affect these escape sequences. For example, \p{Lu} always matches only upper case letters.
The \X escape matches any number of Unicode characters that form an extended Unicode sequence. \X is equivalent to

(?>\pM\pM*)

That is, it matches a character without the “mark” property, followed by zero or more characters with the “mark” property, and treats the sequence as an atomic group (see below). Characters with the “mark” property are typically accents that affect the preceding character. None of them have codepoints less than 256, so in non-UTF-8 mode \X matches any one character.
Matching characters by Unicode property is not fast, because PCRE has to search a structure that contains data for over fifteen thousand characters. That is why the traditional escape sequences such as \d and \w do not use Unicode properties in PCRE.
Resetting the match start
The escape sequence \K, which is a Perl 5.10 feature, causes any previously matched characters not to be included in the final matched sequence. For example, the pattern:
foo\Kbar

matches “foobar”, but reports that it has matched “bar”. This feature is similar to a
lookbehind assertion (described below). However, in this case, the part of the subject
before the real match does not have to be of fixed length, as lookbehind assertions do.
The use of \K does not interfere with the setting of captured substrings. For example, when the pattern

(foo)\Kbar

matches “foobar”, the first substring is still set to “foo”.

Simple assertions

The final use of backslash is for certain simple assertions. An assertion specifies a
condition that has to be met at a particular point in a match, without consuming any
characters from the subject string. The use of subpatterns for more complicated
assertions is described below. The backslashed assertions are:

\b matches at a word boundary
\B matches when not at a word boundary
\A matches at the start of the subject
\Z matches at the end of the subject also matches before a newline at the end of the
subject
\z matches only at the end of the subject
\G matches at the first matching position in the subject

These assertions may not appear in character classes (but note that \b has a different
meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the
previous character do not both match \w or \W (i.e. one matches \w and the other
matches \W), or the start or end of the string if the first or last character matches \w,
respectively.

The \A, \Z, and \z assertions differ from the traditional circumflex and dollar
(described in the next section) in that they only ever match at the very start and end of
the subject string, whatever options are set. Thus, they are independent of multiline
mode. These three assertions are not affected by the notbol or noteol options, which
affect only the behaviour of the circumflex and dollar metacharacters. However, if the
startoffset argument of re:run/3 is non-zero, indicating that matching is to start at a
point other than the beginning of the subject, \A can never match. The difference
between \Z and \z is that \Z matches before a newline at the end of the string as well
as at the very end, whereas \z matches only at the end.

The \G assertion is true only when the current matching position is at the start point of
the match, as specified by the startoffset argument of re:run/3. It differs from \A when
the value of startoffset is non-zero. By calling re:run/3 multiple times with appropriate
arguments, you can mimic Perl’s /g option, and it is in this kind of implementation
where \G can be useful.

Note, however, that PCRE’s interpretation of \G, as the start of the current match, is
subtly different from Perl’s, which defines it as the end of the previous match. In Perl,
these can be different when the previously matched string was empty. Because PCRE
does just one match at a time, it cannot reproduce this behaviour.
If all the alternatives of a pattern begin with \G, the expression is anchored to the starting match position, and the “anchored” flag is set in the compiled regular expression.

Circumflex and dollar

Outside a character class, in the default matching mode, the circumflex character is an assertion that is true only if the current matching point is at the start of the subject string. If the startoffset argument of \re:run/3 is non-zero, circumflex can never match if the multiline option is unset. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an “anchored” pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character is an assertion that is true only if the current matching point is at the end of the subject string, or immediately before a newline at the end of the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meaning of dollar can be changed so that it matches only at the very end of the string, by setting the dollar_endonly option at compile time. This does not affect the \Z assertion.

The meanings of the circumflex and dollar characters are changed if the multiline option is set. When this is the case, a circumflex matches immediately after internal newlines as well as at the start of the subject string. It does not match after a newline that ends the string. A dollar matches before any newlines in the string, as well as at the very end, when multiline is set. When newline is specified as the two-character sequence CRLF, isolated CR and LF characters do not indicate newlines.

For example, the pattern /\^abc$/ matches the subject string “def\nabc” (where \n represents a newline) in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with \^ are not anchored in multiline mode, and a match for circumflex is possible when the startoffset argument of \re:run/3 is non-zero. The dollar_endonly option is ignored if multiline is set.

Note that the sequences \A, \Z, and \z can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with \A it is always anchored, whether or not multiline is set.

Full stop (period, dot)

Outside a character class, a dot in the pattern matches any one character in the subject string except (by default) a character that signifies the end of a line. In UTF-8 mode, the matched character may be more than one byte long.

When a line ending is defined as a single character, dot never matches that character; when the two-character sequence CRLF is used, dot does not match CR if it is immediately followed by LF, but otherwise it matches all characters (including isolated
When any Unicode line endings are being recognized, dot does not match CR or LF or any of the other line ending characters.

The behaviour of dot with regard to newlines can be changed. If the dotall option is set, a dot matches any one character, without exception. If the two-character sequence CRLF is present in the subject string, it takes two dots to match it.

The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newlines. Dot has no special meaning in a character class.

Matching a single byte

Outside a character class, the escape sequence \C matches any one byte, both in and out of UTF-8 mode. Unlike a dot, it always matches any line-ending characters. The feature is provided in Perl in order to match individual bytes in UTF-8 mode. Because it breaks up UTF-8 characters into individual bytes, what remains in the string may be a malformed UTF-8 string. For this reason, the \C escape sequence is best avoided.

PCRE does not allow \C to appear in lookbehind assertions (described below), because in UTF-8 mode this would make it impossible to calculate the length of the lookbehind.

Square brackets and character classes

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject. In UTF-8 mode, the character may occupy more than one byte. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class [aeiou] matches any lower case vowel, while[^aeiou] matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters that are in the class by enumerating those that are not. A class that starts with a circumflex is not an assertion: it still consumes a character from the subject string, and therefore it fails if the current pointer is at the end of the string.

In UTF-8 mode, characters with values greater than 255 can be included in a class as a literal string of bytes, or by using the \x\ escaping mechanism.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless [aeiou] matches “A” as well as “a”, and a caseless[^aeiou] does not match “A”, whereas a caseful version would. In UTF-8 mode, PCRE always understands the concept of case for characters whose values are less than 128, so caseless matching is always possible. For characters with higher values, the concept of case is supported if PCRE is compiled with Unicode property support, but not otherwise. If you want to use caseless matching for characters 128 and above, you must ensure that PCRE is compiled with Unicode property support as well as with UTF-8 support.
Characters that might indicate line breaks are never treated in any special way when matching character classes, whatever line-ending sequence is in use, and whatever setting of the dotall and multiline options is used. A class such as `[^a]` always matches one of these characters.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, `[d-m]` matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character “]” as the end character of a range. A pattern such as `[W-]46]` is interpreted as a class of two characters (“W” and “-”) followed by a literal string “46]”, so it would match “W46]” or “~46]”. However, if the “]” is escaped with a backslash it is interpreted as the end of range, so `[W-\]46]` is interpreted as a class containing a range followed by two other characters. The octal or hexadecimal representation of “]” can also be used to end a range.

Ranges operate in the collating sequence of character values. They can also be used for characters specified numerically, for example `[\000-\037]`. In UTF-8 mode, ranges can include characters whose values are greater than 255, for example `[\x{100}-\x{2ff}]`.

If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, `[W-c]` is equivalent to `[^\w]`, matched caselessly, and in non-UTF-8 mode, if character tables for a French locale are in use, `[\xc8-\xcb]` matches accented E characters in both cases. In UTF-8 mode, PCRE supports the concept of case for characters with values greater than 128 only when it is compiled with Unicode property support.

The character types `d`, `D`, `p`, `P`, `s`, `S`, `w`, and `W` may also appear in a character class, and add the characters that they match to the class. For example, `[\dABCDEF]` matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class `[^\W_]` matches any letter or digit, but not underscore.

The only metacharacters that are recognized in character classes are backslash, hyphen (only where it can be interpreted as specifying a range), circumflex (only at the start), opening square bracket (only when it can be interpreted as introducing a POSIX class name - see the next section), and the terminating closing square bracket. However, escaping other non-alphanumeric characters does no harm.

### Posix character classes

Perl supports the POSIX notation for character classes. This uses names enclosed by `[:` and `:]` within the enclosing square brackets. PCRE also supports this notation. For example,

```
[01[:alpha:]%]
```

matches “0”, “1”, any alphabetic character, or “%”. The supported class names are

- **alnum** letters and digits
- **alpha** letters
- **ascii** character codes 0 - 127
blank  space or tab only
ctrl   control characters
digit  decimal digits (same as \d)
graph  printing characters, excluding space
lower  lower case letters
print  printing characters, including space
punct  printing characters, excluding letters and digits
space  whitespace (not quite the same as \s)
upper  upper case letters
word   "word" characters (same as \w)
xdigit hexadecimal digits

The “space” characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32).
Notice that this list includes the VT character (code 11). This makes “space” different
to \s, which does not include VT (for Perl compatibility).
The name “word” is a Perl extension, and “blank” is a GNU extension from Perl 5.8.
Another Perl extension is negation, which is indicated by a ^ character after the colon.
For example,

    [12[^\d:]]

matches “1”, “2”, or any non-digit. PCRE (and Perl) also recognize the POSIX syntax
[.ch.] and [=ch=] where “ch” is a “collating element”, but these are not supported, and
an error is given if they are encountered.
In UTF-8 mode, characters with values greater than 128 do not match any of the
POSIX character classes.

Vertical bar

Vertical bar characters are used to separate alternative patterns. For example, the pattern

    gilbert|sullivan

matches either “gilbert” or “sullivan”. Any number of alternatives may appear, and an
empty alternative is permitted (matching the empty string). The matching process tries
each alternative in turn, from left to right, and the first one that succeeds is used. If the
alternatives are within a subpattern (defined below), “succeeds” means matching the
rest of the main pattern as well as the alternative in the subpattern.
Internal option setting

The settings of the caseless, multiline, dotall, and extended options (which are Perl-compatible) can be changed from within the pattern by a sequence of Perl option letters enclosed between "(?" and ")". The option letters are

- `i` for caseless
- `m` for multiline
- `s` for dotall
- `x` for extended

For example, (?im) sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as (?im-sx), which sets caseless and multiline while unsetting dotall and extended, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

The PCRE-specific options dupnames, ungreedy, and extra can be changed in the same way as the Perl-compatible options by using the characters J, U and X respectively.

When an option change occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. If the change is placed right at the start of a pattern, PCRE extracts it into the global options.

An option change within a subpattern (see below for a description of subpatterns) affects only that part of the current pattern that follows it, so

```
(a(?i)b)c
```

matches abc and aBc and no other strings (assuming caseless is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

```
(a(?i)b|c)
```

matches “ab”, “aB”, “c”, and “C”, even though when matching “C” the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

Note: There are other PCRE-specific options that can be set by the application when the compile or match functions are called. In some cases the pattern can contain special leading sequences to override what the application has set or what has been defaulted. Details are given in the section entitled “Newline sequences” above.
Subpatterns

Subpatterns are delimited by parentheses (round brackets), which can be nested. Turning part of a pattern into a subpattern does two things:

1. It localizes a set of alternatives. For example, the pattern
   
   \texttt{cat(aract|erpillar)}
   
   matches one of the words “cat”, “cataract”, or “caterpillar”. Without the parentheses, it would match “cataaract”, “erpillar” or an empty string.

2. It sets up the subpattern as a capturing subpattern. This means that, when the complete pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller via the return value of \texttt{re:run/3}. Opening parentheses are counted from left to right (starting from 1) to obtain numbers for the capturing subpatterns.

   For example, if the string “the red king” is matched against the pattern
   
   \texttt{the (\((red|white)\) (king|queen))}
   
   the captured substrings are “red king”, “red”, and “king”, and are numbered 1, 2, and 3, respectively.

   The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by a question mark and a colon, the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string “the white queen” is matched against the pattern
   
   \texttt{the (\((?:red|white)\) (king|queen))}
   
   the captured substrings are “white queen” and “queen”, and are numbered 1 and 2. The maximum number of capturing subpatterns is 65535.

   As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the “?” and the “:”. Thus the two patterns

   \begin{itemize}
   \item \texttt{(?:sat\{urday|sunday\)}
   \item \texttt{(?:sat\{urday|sunday\)}
   \end{itemize}

   match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match “SUNDAY” as well as “Saturday”.

---

374

STDLIB
Duplicate subpattern numbers

Perl 5.10 introduced a feature whereby each alternative in a subpattern uses the same numbers for its capturing parentheses. Such a subpattern starts with (\?) and is itself a non-capturing subpattern. For example, consider this pattern:

```
(?|(Sat)ur|((Sun))day
```

Because the two alternatives are inside a (\?) group, both sets of capturing parentheses are numbered one. Thus, when the pattern matches, you can look at captured substring number one, whichever alternative matched. This construct is useful when you want to capture part, but not all, of one of a number of alternatives. Inside a (\?) group, parentheses are numbered as usual, but the number is reset at the start of each branch. The numbers of any capturing buffers that follow the subpattern start after the highest number used in any branch. The following example is taken from the Perl documentation. The numbers underneath show in which buffer the captured content will be stored.

```
# before ----------------branch-reset------------ after
/ (a) ( ? |x ( y ) z | ( p ( q ) r ) | ( t ) u ( v ) ) ( z ) /x
# 1 2 2 3 2 3 4
```

A backreference or a recursive call to a numbered subpattern always refers to the first one in the pattern with the given number.

An alternative approach to using this “branch reset” feature is to use duplicate named subpatterns, as described in the next section.

Named subpatterns

Identifying capturing parentheses by number is simple, but it can be very hard to keep track of the numbers in complicated regular expressions. Furthermore, if an expression is modified, the numbers may change. To help with this difficulty, PCRE supports the naming of subpatterns. This feature was not added to Perl until release 5.10. Python had the feature earlier, and PCRE introduced it at release 4.0, using the Python syntax. PCRE now supports both the Perl and the Python syntax.

In PCRE, a subpattern can be named in one of three ways: (\?<name>...) or (?'name'...) as in Perl, or (?P<name>...) as in Python. References to capturing parentheses from other parts of the pattern, such as backreferences, recursion, and conditions, can be made by name as well as by number.

Names consist of up to 32 alphanumeric characters and underscores. Named capturing parentheses are still allocated numbers as well as names, exactly as if the names were not present. The `capture` specification to `re:run/3` can use named values if they are present in the regular expression.

By default, a name must be unique within a pattern, but it is possible to relax this constraint by setting the `dupnames` option at compile time. This can be useful for patterns where only one instance of the named parentheses can match. Suppose you want to match the name of a weekday, either as a 3-letter abbreviation or as the full name, and in both cases you want to extract the abbreviation. This pattern (ignoring the line breaks) does the job:
There are five capturing substrings, but only one is ever set after a match. (An alternative way of solving this problem is to use a “branch reset” subpattern, as described in the previous section.)

In case of capturing named subpatterns which are not unique, the first occurrence is returned from \texttt{re:exec/3}, if the name is specified in the \texttt{values} part of the \texttt{capture} statement.

### Repetition

Repetition is specified by quantifiers, which can follow any of the following items:

- a literal data character
- the dot metacharacter
- the \texttt{\C} escape sequence
- the \texttt{\X} escape sequence (in UTF-8 mode with Unicode properties)
- the \texttt{\R} escape sequence
- an escape such as \texttt{\d} that matches a single character
- a character class
- a back reference (see next section)
- a parenthesized subpattern (unless it is an assertion)

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

```
z{2,4}
```

matches “zz”, “zzz”, or “zzzz”. A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

```
[aeiou]{3,}
```

matches at least 3 successive vowels, but may match many more, while

```
\d{8}
```
matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, \{,6\} is not a quantifier, but a literal string of four characters.

In UTF-8 mode, quantifiers apply to UTF-8 characters rather than to individual bytes. Thus, for example, \x{100}\{2\} matches two UTF-8 characters, each of which is represented by a two-byte sequence. Similarly, when Unicode property support is available, \X{3} matches three Unicode extended sequences, each of which may be several bytes long (and they may be of different lengths).

The quantifier \{0\} is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience, the three most common quantifiers have single-character abbreviations:

* is equivalent to \{0,\}
+ is equivalent to \{1,\}
? is equivalent to \{0,1\}

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are “greedy”, that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between /* and */ and within the comment, individual *, and / characters may appear. An attempt to match C comments by applying the pattern

```
/\*/.*/\*/
```

to the string

```
/* first comment */ not comment /* second comment */
```

fails, because it matches the entire string owing to the greediness of the .* item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\*/.*/\*/
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d
```
which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the ungready option is set (an option that is not available in Perl), the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more memory is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with .* or . and the dotall option (equivalent to Perl's /s) is set, thus allowing the dot to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE normally treats such a pattern as though it were preceded by \A.

In cases where it is known that the subject string contains no newlines, it is worth setting dotall in order to obtain this optimization, or alternatively using ^ to indicate anchoring explicitly.

However, there is one situation where the optimization cannot be used. When .* is inside capturing parentheses that are the subject of a backreference elsewhere in the pattern, a match at the start may fail where a later one succeeds. Consider, for example:

\(.*abc\1\)

If the subject is “xyz123abc123” the match point is the fourth character. For this reason, such a pattern is not implicitly anchored.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

\((tweedle\[dume\]3\s*)+\)

has matched “tweedledum tweedledee” the value of the captured substring is “tweedledee”. However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

/\(a(b)\)+/  

matches “aba” the value of the second captured substring is “b”.

Atomic grouping and possessive quantifiers

With both maximizing (“greedy”) and minimizing (“ungreedy” or “lazy”) repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern \d+foo when applied to the subject line

123456bar
After matching all 6 digits and then failing to match “foo”, the normal action of the
matcher is to try again with only 5 digits matching the \d+ item, and then with 4, and
so on, before ultimately failing. “Atomic grouping” (a term taken from Jeffrey Friedl’s
book) provides the means for specifying that once a subpattern has matched, it is not to
be re-evaluated in this way.

If we use atomic grouping for the previous example, the matcher gives up immediately
on failing to match “foo” the first time. The notation is a kind of special parenthesis,
starting with (?> as in this example:

(?>\d+)foo

This kind of parenthesis “locks up” the part of the pattern it contains once it has
matched, and a failure further into the pattern is prevented from backtracking into it.
Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of
characters that an identical standalone pattern would match, if anchored at the current
point in the subject string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the
above example can be thought of as a maximizing repeat that must swallow everything
it can. So, while both \d+ and \d+? are prepared to adjust the number of digits they
match in order to make the rest of the pattern match, (?>\d+) can only match an entire
sequence of digits.

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and
can be nested. However, when the subpattern for an atomic group is just a single
repeated item, as in the example above, a simpler notation, called a “possessive
quantifier” can be used. This consists of an additional + character following a quantifier.
Using this notation, the previous example can be rewritten as

\d++foo

Note that a possessive quantifier can be used with an entire group, for example:

(abc|xyz){2,3}+

Possessive quantifiers are always greedy; the setting of the ungreedy option is ignored.
They are a convenient notation for the simpler forms of atomic group. However, there
is no difference in the meaning of a possessive quantifier and the equivalent atomic
group, though there may be a performance difference; possessive quantifiers should be
slightly faster.

The possessive quantifier syntax is an extension to the Perl 5.8 syntax. Jeffrey Friedl
originated the idea (and the name) in the first edition of his book. Mike McCloskey
liked it, so implemented it when he built Sun’s java package, and PCRE copied it from
there. It ultimately found its way into Perl at release 5.10.

PCRE has an optimization that automatically “possessifies” certain simple pattern
constructs. For example, the sequence A +B is treated as A ++B because there is no point
in backtracking into a sequence of A’s when B must follow.

When a pattern contains an unlimited repeat inside a subpattern that can itself be
repeated an unlimited number of times, the use of an atomic group is the only way to
avoid some failing matches taking a very long time indeed. The pattern

(\D+|<\d+>)[!?]
matches an unlimited number of substrings that either consist of non-digits, or digits enclosed in <>, followed by either ! or ?. When it matches, it runs quickly. However, if it is applied to

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```
it takes a long time before reporting failure. This is because the string can be divided between the internal \D+ repeat and the external * repeat in a large number of ways, and all have to be tried. (The example uses [!?] rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed so that it uses an atomic group, like this:

```
((?\D+)|<\d+>)[!?]`
```
sequences of non-digits cannot be broken, and failure happens quickly.

**Back references**

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been that many previous capturing left parentheses. However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. A “forward back reference” of this type can make sense when a repetition is involved and the subpattern to the right has participated in an earlier iteration.

It is not possible to have a numerical “forward back reference” to a subpattern whose number is 10 or more using this syntax because a sequence such as \50 is interpreted as a character defined in octal. See the subsection entitled “Non-printing characters” above for further details of the handling of digits following a backslash. There is no such problem when named parentheses are used. A back reference to any subpattern is possible using named parentheses (see below).

Another way of avoiding the ambiguity inherent in the use of digits following a backslash is to use the \g escape sequence, which is a feature introduced in Perl 5.10. This escape must be followed by an unsigned number or a negative number, optionally enclosed in braces. These examples are all identical:

- `(ring), \1`
- `(ring), \g1`
- `(ring), \g{1}`

An unsigned number specifies an absolute reference without the ambiguity that is present in the older syntax. It is also useful when literal digits follow the reference. A negative number is a relative reference. Consider this example:

```
(abc(def)ghi)\g{-1}
```
The sequence \g{-1} is a reference to the most recently started capturing subpattern before \g, that is, is it equivalent to \2. Similarly, \g{-2} would be equivalent to \1. The use of relative references can be helpful in long patterns, and also in patterns that are created by joining together fragments that contain references within themselves.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (see “Subpatterns as subroutines” below for a way of doing that). So the pattern

\begin{verbatim}
(sens|response) and \1ibility
\end{verbatim}

matches “sense and sensibility” and “response and responsibility”, but not “sense and responsibility”. If caseful matching is in force at the time of the back reference, the case of letters is relevant. For example,

\begin{verbatim}
((?i)rah)\s+\1
\end{verbatim}

matches “rah rah” and “RAH RAH”, but not “RAH rah”, even though the original capturing subpattern is matched caselessly.

There are several different ways of writing back references to named subpatterns. The .NET syntax \k{name} and the Perl syntax \<name> or \k{name} are supported, as is the Python syntax (?P=name). Perl 5.10’s unified back reference syntax, in which \g can be used for both numeric and named references, is also supported. We could rewrite the above example in any of the following ways:

- (?<p1>(?i)rah)\s+\k<p1>
- (?'p1'(?!rah)\s+\k{p1}
- (?P<p1>(?!rah)\s+(?P=p1)
- (?<p1>(?!rah)\s+\g{p1}

A subpattern that is referenced by name may appear in the pattern before or after the reference.

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern

\begin{verbatim}
(a(bc))\2
\end{verbatim}

always fails if it starts to match “a” rather than “bc”. Because there may be many capturing parentheses in a pattern, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the extended option is set, this can be whitespace. Otherwise an empty comment (see “Comments” below) can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern

\begin{verbatim}
(a(b)c)\1+
\end{verbatim}

matches any number of “a”s and also “aba”, “ababaa” etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.
Assertions

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as \b, \B, \A, \G, \Z, \z, ^ and $ are described above.

More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it. An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed.

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Lookahead assertions

Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example,

\w+(?=;)

matches a word followed by a semicolon, but does not include the semicolon in the match, and

foo(?!bar)

matches any occurrence of “foo” that is not followed by “bar”. Note that the apparently similar pattern

(?!foo)bar

does not find an occurrence of “bar” that is preceded by something other than “foo”; it finds any occurrence of “bar” whatsoever, because the assertion (?!foo) is always true when the next three characters are “bar”. A lookbehind assertion is needed to achieve the other effect.

If you want to force a matching failure at some point in a pattern, the most convenient way to do it is with (?!) because an empty string always matches, so an assertion that requires there not to be an empty string must always fail.

Lookbehind assertions

Lookbehind assertions start with (?! for positive assertions and (?! for negative assertions. For example,

(?!<foo)bar

does find an occurrence of “bar” that is not preceded by “foo”. The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several top-level alternatives, they do not all have to have the same fixed length. Thus

(?!<bullock|donkey)

is permitted, but
causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl (at least for 5.8), which requires all branches to match the same length of string. An assertion such as

```regex
(?=<!dogs?|cats?)
```

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches:

```regex
(?=<=ab(c|de))
```

In some cases, the Perl 5.10 escape sequence `\K` (see above) can be used instead of a lookbehind assertion; this is not restricted to a fixed-length.

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed length and then try to match. If there are insufficient characters before the current position, the assertion fails.

PCRE does not allow the `\C` escape (which matches a single byte in UTF-8 mode) to appear in lookbehind assertions, because it makes it impossible to calculate the length of the lookbehind. The `\X` and `\R` escapes, which can match different numbers of bytes, are also not permitted.

Possessive quantifiers can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

```regex
abcd$
```

when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for each “a” in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```regex
.*abcd$
```

the initial `.*` matches the entire string at first, but when this fails (because there is no following “a”), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for “a” covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```regex
.*+(?=abcd)
```

there can be no backtracking for the `.*+` item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

Using multiple assertions

Several assertions (of any sort) may occur in succession. For example,

```regex
(?=<\d(3))(<!\d999)foo
```
matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does not match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn’t match "123abcfoo". A pattern to do that is

```regex
(?<=\d{3})(?<!999)foo
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example,

```regex
(?<=(?!foo)bar)baz
```

matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while

```regex
(?<=\d{3})(?!999)...foo
```

is another pattern that matches "foo" preceded by three digits and any three characters that are not "999".

### Conditional subpatterns

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are

- `(?(condition)yes-pattern)`
- `(?(condition)yes-pattern|no-pattern)`

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are four kinds of condition: references to subpatterns, references to recursion, a pseudo-condition called `DEFINE`, and assertions.

Checking for a used subpattern by number

If the text between the parentheses consists of a sequence of digits, the condition is true if the capturing subpattern of that number has previously matched. An alternative notation is to precede the digits with a plus or minus sign. In this case, the subpattern number is relative rather than absolute. The most recently opened parentheses can be referenced by `(?!-1)`, the next most recent by `(?!-2)`, and so on. In looping constructs it can also make sense to refer to subsequent groups with constructs such as `(?!+2)`.

Consider the following pattern, which contains non-significant whitespace to make it more readable (assume the `extended` option) and to divide it into three parts for ease of discussion:

```regex
( \( )? \[ -()\]+ (?!\(1 \) )\)
```
The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If you were embedding this pattern in a larger one, you could use a relative reference:

```
...other stuff... ( \( ) ? [^()] + (?<-1> \) ) ...
```

This makes the fragment independent of the parentheses in the larger pattern.

Checking for a used subpattern by name

Perl uses the syntax `(?(<name>)...)` or `(?('name')...)` to test for a used subpattern by name. For compatibility with earlier versions of PCRE, which had this facility before Perl, the syntax `(?{name}...)` is also recognized. However, there is a possible ambiguity with this syntax, because subpattern names may consist entirely of digits. PCRE looks first for a named subpattern; if it cannot find one and the name consists entirely of digits, PCRE looks for a subpattern of that number, which must be greater than zero. Using subpattern names that consist entirely of digits is not recommended.

Rewriting the above example to use a named subpattern gives this:

```
( ?<OPEN> \( ) ? [^()] + (?<OPEN> \) )
```

Checking for pattern recursion

If the condition is the string `(?R)...`, and there is no subpattern with the name `R`, the condition is true if a recursive call to the whole pattern or any subpattern has been made. If digits or a name preceded by ampersand follow the letter `R`, for example:

```
( ?(R3)... ) or ( ?(R&name)... )
```

the condition is true if the most recent recursion is into the subpattern whose number or name is given. This condition does not check the entire recursion stack.

At “top level”, all these recursion test conditions are false. Recursive patterns are described below.

Defining subpatterns for use by reference only

If the condition is the string `(?DEFINE)...`, and there is no subpattern with the name `DEFINE`, the condition is always false. In this case, there may be only one alternative in the subpattern. It is always skipped if control reaches this point in the pattern; the idea of `DEFINE` is that it can be used to define “subroutines” that can be referenced from elsewhere. (The use of “subroutines” is described below.) For example, a pattern to match an IPv4 address could be written like this (ignore whitespace and line breaks):

```
( ?(DEFINE) (?<byte> 2[0-4]\d | 25[0-5] | 1\d\d | [1-9]\d) ) \b (?&byte) \b(\.(?&byte))\{3\} \b
```
The first part of the pattern is a DEFINE group inside which another group named “byte” is defined. This matches an individual component of an IPv4 address (a number less than 256). When matching takes place, this part of the pattern is skipped because DEFINE acts like a false condition.

The rest of the pattern uses references to the named group to match the four dot-separated components of an IPv4 address, insisting on a word boundary at each end.

Assertion conditions

If the condition is not in any of the above formats, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant whitespace, and with the two alternatives on the second line:

```
(?(?=[^a-z]*[a-z])
d{2}-[a-z]{3}-d{2} | d{2}-d{2}-d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

Comments

The sequence (?# marks the start of a comment that continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the extended option is set, an unescaped # character outside a character class introduces a comment that continues to immediately after the next newline in the pattern.

Recursive patterns

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth.

For some time, Perl has provided a facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at run time, and the code can refer to the expression itself. A Perl pattern using code interpolation to solve the parentheses problem can be created like this:

```
$re = qr{\( (?: (?=\^[a-z]+) 1 (?p{$re}) \)\)* }x;
```

The (?p{...}) item interpolates Perl code at run time, and in this case refers recursively to the pattern in which it appears.

Obviously, PCRE cannot support the interpolation of Perl code. Instead, it supports special syntax for recursion of the entire pattern, and also for individual subpattern recursion. After its introduction in PCRE and Python, this kind of recursion was introduced into Perl at release 5.10.
A special item that consists of (? followed by a number greater than zero and a closing parenthesis is a recursive call of the subpattern of the given number, provided that it occurs inside that subpattern. (If not, it is a “subroutine” call, which is described in the next section.) The special item (?R) or (?0) is a recursive call of the entire regular expression.

In PCRE (like Python, but unlike Perl), a recursive subpattern call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure.

This PCRE pattern solves the nested parentheses problem (assume the extended option is set so that whitespace is ignored):

```
\( ( (?>[^()]+) | (?R) )* \)
```

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of non-parentheses, or a recursive match of the pattern itself (that is, a correctly parenthesized substring). Finally there is a closing parenthesis.

If this were part of a larger pattern, you would not want to recurse the entire pattern, so instead you could use this:

```
( \( ( (?>[^()]+) | (?1) )* \) )
```

We have put the pattern into parentheses, and caused the recursion to refer to them instead of the whole pattern.

In a larger pattern, keeping track of parenthesis numbers can be tricky. This is made easier by the use of relative references. (A Perl 5.10 feature.) Instead of (?1) in the pattern above you can write (?-2) to refer to the second most recently opened parentheses preceding the recursion. In other words, a negative number counts capturing parentheses leftwards from the point at which it is encountered.

It is also possible to refer to subsequently opened parentheses, by writing references such as (?+2). However, these cannot be recursive because the reference is not inside the parentheses that are referenced. They are always “subroutine” calls, as described in the next section.

An alternative approach is to use named parentheses instead. The Perl syntax for this is (?&name); PCRE’s earlier syntax (?P> name) is also supported. We could rewrite the above example as follows:

```
(?<pn> \( ( (?>[^()]+) | (?&pn) )* \) )
```

If there is more than one subpattern with the same name, the earliest one is used.

This particular example pattern that we have been looking at contains nested unlimited repeats, and so the use of atomic grouping for matching strings of non-parentheses is important when applying the pattern to strings that do not match. For example, when this pattern is applied to

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa)
```
it yields “no match” quickly. However, if atomic grouping is not used, the match runs
for a very long time indeed because there are so many different ways the + and * repeats
can carve up the subject, and all have to be tested before failure can be reported.

At the end of a match, the values set for any capturing subpatterns are those from the
outermost level of the recursion at which the subpattern value is set. If the pattern
above is matched against

(ab(cd)ef)

the value for the capturing parentheses is “ef”, which is the last value taken on at the
top level. If additional parentheses are added, giving

\( ( ( (?>[^()]+) | (?R) )* ) \)

the string they capture is “ab(cd)ef”, the contents of the top level parentheses.

Do not confuse the (?R) item with the condition (R), which tests for recursion.
Consider this pattern, which matches text in angle brackets, allowing for arbitrary
nesting. Only digits are allowed in nested brackets (that is, when recursing), whereas
any characters are permitted at the outer level.

\(< (?: (?R) \d++ | [<>]*+ ) | (?R) )* >\)

In this pattern, (?R) is the start of a conditional subpattern, with two different
alternatives for the recursive and non-recursive cases. The (?R) item is the actual
recursive call.

Subpatterns as subroutines

If the syntax for a recursive subpattern reference (either by number or by name) is used
outside the parentheses to which it refers, it operates like a subroutine in a
programming language. The “called” subpattern may be defined before or after the
reference. A numbered reference can be absolute or relative, as in these examples:

• (...(absolute)...)(?2)...
• (...(relative)...)(?1)...
• (...(?+1)...(relative)...

An earlier example pointed out that the pattern

(sens|response)es and \libility

matches “sense and sensibility” and “response and responsibility”, but not “sense and
responsibility”. If instead the pattern

(sens|response)es and (?1)ibility
is used, it does match “sense and responsibility” as well as the other two strings.

Another example is given in the discussion of DEFINE above.

Like recursive subpatterns, a “subroutine” call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure.

When a subpattern is used as a subroutine, processing options such as case-independence are fixed when the subpattern is defined. They cannot be changed for different calls. For example, consider this pattern:

\[(abc)(?!:(?-1))\]

It matches “abcabc”. It does not match “abcABC” because the change of processing option does not affect the called subpattern.

### Backtracking control

Perl 5.10 introduced a number of “Special Backtracking Control Verbs”, which are described in the Perl documentation as “experimental and subject to change or removal in a future version of Perl”. It goes on to say: “Their usage in production code should be noted to avoid problems during upgrades.” The same remarks apply to the PCRE features described in this section.

The new verbs make use of what was previously invalid syntax: an opening parenthesis followed by an asterisk. In Perl, they are generally of the form (*VERB:ARG) but PCRE does not support the use of arguments, so its general form is just (*VERB). Any number of these verbs may occur in a pattern. There are two kinds:

Verbs that act immediately

The following verbs act as soon as they are encountered:

\[(*\text{ACCEPT})\]

This verb causes the match to end successfully, skipping the remainder of the pattern. When inside a recursion, only the innermost pattern is ended immediately. PCRE differs from Perl in what happens if the (*\text{ACCEPT}) is inside capturing parentheses. In Perl, the data so far is captured: in PCRE no data is captured. For example:

\[A(A|B(*\text{ACCEPT})|C)D\]

This matches “AB”, “AAD”, or “ACD”, but when it matches “AB”, no data is captured.

\[(*\text{FAIL})\text{ or }(*F)\]

This verb causes the match to fail, forcing backtracking to occur. It is equivalent to (?!) but easier to read. The Perl documentation notes that it is probably useful only when combined with (?{ }) or (?{ })]. Those are, of course, Perl features that are not present in PCRE. The nearest equivalent is the callout feature, as for example in this pattern:

\[a+(?C)(*\text{FAIL})\]
A match with the string “aaaa” always fails, but the callout is taken before each backtrack happens (in this example, 10 times).

Verbs that act after backtracking

The following verbs do nothing when they are encountered. Matching continues with what follows, but if there is no subsequent match, a failure is forced. The verbs differ in exactly what kind of failure occurs.

(*COMMIT)

This verb causes the whole match to fail outright if the rest of the pattern does not match. Even if the pattern is unanchored, no further attempts to find a match by advancing the start point take place. Once (*COMMIT) has been passed, re:run/3 is committed to finding a match at the current starting point, or not at all. For example:

}\texttt{a+(*COMMIT)b}

This matches “xxaab” but not “aaacaab”. It can be thought of as a kind of dynamic anchor, or “I’ve started, so I must finish.”

(*PRUNE)

This verb causes the match to fail at the current position if the rest of the pattern does not match. If the pattern is unanchored, the normal “bumpalong” advance to the next starting character then happens. Backtracking can occur as usual to the left of (*PRUNE), or when matching to the right of (*PRUNE), but if there is no match to the right, backtracking cannot cross (*PRUNE). In simple cases, the use of (*PRUNE) is just an alternative to an atomic group or possessive quantifier, but there are some uses of (*PRUNE) that cannot be expressed in any other way.

(*SKIP)

This verb is like (*PRUNE), except that if the pattern is unanchored, the “bumpalong” advance is not to the next character, but to the position in the subject where (*SKIP) was encountered. (*SKIP) signifies that whatever text was matched leading up to it cannot be part of a successful match. Consider:

}\texttt{a+(*SKIP)b}

If the subject is “aaaa...”, after the first match attempt fails (starting at the first character in the string), the starting point skips on to start the next attempt at “c”. Note that a possessive quantifier does not have the same effect in this example; although it would suppress backtracking during the first match attempt, the second attempt would start at the second character instead of skipping on to “c”.

(*THEN)

This verb causes a skip to the next alternation if the rest of the pattern does not match. That is, it cancels pending backtracking, but only within the current alternation. Its name comes from the observation that it can be used for a pattern-based if-then-else block:

}\texttt{(COND1 (*THEN) FOO \mid COND2 (*THEN) BAR \mid COND3 (*THEN) BAZ) ...}
If the COND1 pattern matches, FOO is tried (and possibly further items after the end of the group if FOO succeeds); on failure the matcher skips to the second alternative and tries COND2, without backtracking into COND1. If (*THEN) is used outside of any alternation, it acts exactly like (*PRUNE).
regexp

Erlang Module

Note:
This module has been obsoleted by the re [page 345] module and will be removed in a future release.

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()
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,errmsg()\}
- 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,errmsg()\}

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 = errmsg()
- 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 `r`
- `r*` matches zero or more `r`
- `r?` matches zero or one `r`
- `(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

Erlang Module

Sets are collections of elements with no duplicate elements. The representation of a set is not defined.

This module provides exactly the same interface as the module ordsets but with a defined representation. One difference is that while this module considers two elements as different if they do not match (\(\neq\)), ordsets considers two elements as different if and only if they do not compare equal (\(==\)).

DATA TYPES

set() as returned by new/0

Exports

new() -> Set

Types:
- Set = set()
Returns a new empty set.

is_set(Set) -> bool()

Types:
- Set = term()
Returns true if Set is a set of elements, otherwise false.

size(Set) -> int()

Types:
- Set = term()
Returns the number of elements in Set.

to_list(Set) -> List

Types:
- Set = set()
- List = [term()]
Returns the elements of Set as a list.

from_list(List) -> Set
Types:
- List = [term()]
- Set = set()
Returns an set of the elements in List.

is_element(Element, Set) -> bool()
Types:
- Element = term()
- Set = set()
Returns true if Element is an element of Set, otherwise false.

add_element(Element, Set1) -> Set2
Types:
- Element = term()
- Set1 = Set2 = set()
Returns a new set formed from Set1 with Element inserted.

del_element(Element, Set1) -> Set2
Types:
- Element = term()
- Set1 = Set2 = set()
Returns Set1, but with Element removed.

union(Set1, Set2) -> Set3
Types:
- Set1 = Set2 = Set3 = set()
Returns the merged (union) set of Set1 and Set2.

union(SetList) -> Set
Types:
- SetList = [set()]
- Set = set()
Returns the merged (union) set of the list of sets.

intersection(Set1, Set2) -> Set3
Types:
- Set1 = Set2 = Set3 = set()
Returns the intersection of Set1 and Set2.
intersection(SetList) -> Set

Types:
- \( \text{SetList} = [\text{set()}] \)
- \( \text{Set} = \text{set()} \)

Returns the intersection of the non-empty list of sets.

is_disjoint(Set1, Set2) -> bool()

Types:
- \( \text{Set1} = \text{Set2} = \text{set()} \)

Returns true if Set1 and Set2 are disjoint (have no elements in common), and false otherwise.

subtract(Set1, Set2) -> Set3

Types:
- \( \text{Set1} = \text{Set2} = \text{Set3} = \text{set()} \)

Returns only the elements of Set1 which are not also elements of Set2.

is_subset(Set1, Set2) -> bool()

Types:
- \( \text{Set1} = \text{Set2} = \text{set()} \)

Returns true when every element of Set1 is also a member of Set2, otherwise false.

fold(Function, Acc0, Set) -> Acc1

Types:
- \( \text{Function} = \text{fun (E, AccIn)} \rightarrow \text{AccOut} \)
- \( \text{Acc0} = \text{Acc1} = \text{AccIn} = \text{AccOut} = \text{term()} \)
- \( \text{Set} = \text{set()} \)

Fold Function over every element in Set returning the final value of the accumulator.

filter(Pred, Set1) -> Set2

Types:
- \( \text{Pred} = \text{fun (E)} \rightarrow \text{bool()} \)
- \( \text{Set1} = \text{Set2} = \text{set()} \)

Filter elements in Set1 with boolean function Fun.

See Also

ordsets(3) [page 301], gb_sets(3) [page 204]
The module `shell` implements an Erlang shell. The shell is a user interface program for entering expression sequences. The expressions are evaluated and a value is returned. A history mechanism saves previous commands and their values, which can then be incorporated in later commands. How many commands and results to save can be determined by the user, either interactively, by calling `shell:history/1` and `shell:results/1`, or by setting the application configuration parameters `shell_history_length` and `shell_saved_results` for the application `STDLIB`.

The shell uses a helper process for evaluating commands in order to protect the history mechanism from exceptions. By default the evaluator process is killed when an exception occurs, but by calling `shell:catch_exception/1` or by setting the application configuration parameter `shell_catch_exception` for the application `STDLIB` this behavior can be changed. See also the example below.

Variable bindings, and local process dictionary changes which are generated in user expressions are preserved, and the variables can be used in later commands to access their values. The bindings can also be forgotten so the variables can be re-used.

The special shell commands all have the syntax of (local) function calls. They are evaluated as normal function calls and many commands can be used in one expression sequence.

If a command (local function call) is not recognized by the shell, an attempt is first made to find the function in the module `user_default`, where customized local commands can be placed. If found, then the function is evaluated. Otherwise, an attempt is made to evaluate the function in the module `shell_default`. The module `user_default` must be explicitly loaded.

The shell also permits the user to start multiple concurrent jobs. A job can be regarded as a set of processes which can communicate with the shell.

There is some support for reading and printing records in the shell. During compilation record expressions are translated to tuple expressions. In runtime it is not known whether a tuple actually represents a record. Nor are the record definitions used by compiler available at runtime. So in order to read the record syntax and print tuples as records when possible, record definitions have to be maintained by the shell itself. The shell commands for reading, defining, forgetting, listing, and printing records are described below. Note that each job has its own set of record definitions. To facilitate matters record definitions in the modules `shell_default` and `user_default` (if loaded) are read each time a new job is started. For instance, adding the line

```erlang
-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.

\( \text{history}(N) \) Sets the number of previous commands to keep in the history list to \( N \). The previous number is returned. The default number is 20.

\( \text{results}(N) \) 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 \( N \)th 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 \( N \)th previous command is used (i.e., \( v(-1) \) uses the value of the previous command).

\( \text{help}() \) Evaluates \( \text{shell.default:help}() \).

\( \text{c(File)} \) Evaluates \( \text{shell.default:c(File)} \). 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.

\( \text{catch\_exception}(\text{Bool}) \) Sets the exception handling of the evaluator process. The previous exception handling is returned. The default (false) is to kill the evaluator process when an exception occurs, which causes the shell to create a new evaluator process. When the exception handling is set to true the evaluator process lives on which means that for instance ports and ETS tables as well as processes linked to the evaluator process survive the exception.

\( \text{rd(RecordName, RecordDefinition)} \) Defines a record in the shell. \( \text{RecordName} \) is an atom and \( \text{RecordDefinition} \) lists the field names and the default values. Usually record definitions are made known to the shell by use of the \( \text{rr} \) commands described below, but sometimes it is handy to define records on the fly.

\( \text{rf()} \) Removes all record definitions, then reads record definitions from the modules \( \text{shell.default and user.default (if loaded)} \). Returns the names of the records defined.

\( \text{rf(RecordNames)} \) Removes selected record definitions. \( \text{RecordNames} \) is a record name or a list of record names. Use ‘-’ to remove all record definitions.

\( \text{rl()} \) Prints all record definitions.

\( \text{rl(RecordNames)} \) Prints selected record definitions. \( \text{RecordNames} \) is a record name or a list of record names.

\( \text{rp(Term)} \) Prints a term using the record definitions known to the shell. All of \( \text{Term} \) 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
text are explained at the end of the dialogue.

strider 1> erl
Erlang (BEAM) emulator version 5.3 [hipe] [threads:0]

Eshell V5.3  (abort with ^G)
1> Str = "abcd".
"abcd"
2> L = length(Str).
4
3> Descriptor = {L, list_to_atom(Str)}.
{4,abcd}
4> L
4
5> b()
Descriptor = {4,abcd}
L = 4
Str = "abcd"
ok
6> f(L).
ok
7> b()
Descriptor = {4,abcd}
Str = "abcd"
ok
8> f(L).
ok
9> {L, _} = Descriptor.
{4,abcd}
10> L
4
11> {P, Q, R} = Descriptor.
** exception error: no match of right hand side value {4,abcd}
12> P.  
   * 1: variable 'P' is unbound **
13> Descriptor.  
   [4,abcd]
14>{P, Q} = Descriptor.  
   [4,abcd]
15> P.  
   4
16> f().
   ok
17> put(aa, hello).  
   undefined
18> get(aa).
   hello
19> Y = test1:demo(1).
   hello
20> get().
   [[aa, worked]]
21> put(aa, hello).
   worked
22> Z = test1:demo(2).
   ** exception error: no match of right hand side value 1  
      in function test1:demo/1
23> Z.  
   * 1: variable 'Z' is unbound **
24> get(aa).
   hello
25> erase(), put(aa, hello).
   undefined
26> spawn(test1, demo, [1]).
<0.57.0>
27> get(aa).
   hello
28> io:format("hello hello\ 
      ").
   hello hello ok
29> e(28).
   hello hello ok
30> v(28).
   ok
31> c(ex).
   {ok,ex}
32> rr(ex).
   [rec]
33> rl(rec).
   -record(rec,[a,b = val()]).
   ok
34> #rec[].
   ** exception error: undefined shell command val/0
35> #rec[b = 3].
   #rec[a = undefined,b = 3]
36> rp(v(-1)).
   #rec[a = undefined,b = 3]
   ok
Comments

Command 1 sets the variable \texttt{Str} to the string "abcd".
Command 2 sets \( L \) to the length of the string evaluating the BIF `atom_to_list`.

Command 3 builds the tuple `Descriptor`.

Command 4 prints the value of the variable `L`.

Command 5 evaluates the internal shell command `b()`, which is an abbreviation of “bindings”. This prints the current shell variables and their bindings. The `ok` at the end is the return value of the `b()` function.

Command 6 `f(L)` evaluates the internal shell command `f(L)` (abbreviation of “forget”). The value of the variable `L` is removed.

Command 7 prints the new bindings.

Command 8 has no effect since `L` has no value.

Command 9 performs a pattern matching operation on `Descriptor`, binding a new value to `L`.

Command 10 prints the current value of `L`.

Command 11 tries to match \{P, Q, R\} against `Descriptor` which is \{4, abc\}. The match fails and none of the new variables become bound. The printout starting with 

```
** exception error: 
```

is not the value of the expression (the expression had no value because its evaluation failed), but rather a warning printed by the system to inform the user that an error has occurred. The values of the other variables (\( L \), \( Str \), etc.) are unchanged.

Commands 12 and 13 show that \( P \) is unbound because the previous command failed, and that `Descriptor` has not changed.

Commands 14 and 15 show a correct match where \( P \) and \( Q \) are bound.

Command 16 clears all bindings.

The next few commands assume that `test1:demo(X)` is defined in the following way:

```lisp
demo(X) -> 
    put(aa, worked),
    X = 1,
    X + 10.
```

Commands 17 and 18 set and inspect the value of the item `aa` in the process dictionary.

Command 19 evaluates `test1:demo(1)`. The evaluation succeeds and the changes made in the process dictionary become visible to the shell. The new value of the dictionary item `aa` can be seen in command 20.

Commands 21 and 22 change the value of the dictionary item `aa` to `hello` and call `test1:demo(2)`. Evaluation fails and the changes made to the dictionary in `test1:demo(2)`, before the error occurred, are discarded.

Commands 23 and 24 show that \( Z \) was not bound and that the dictionary item `aa` has retained its original value.

Commands 25, 26 and 27 show the effect of evaluating `test1:demo(1)` in the background. In this case, the expression is evaluated in a newly spawned process. Any changes made in the process dictionary are local to the newly spawned process and therefore not visible to the shell.

Commands 28, 29 and 30 use the history facilities of the shell.

Command 29 is `e(28)`. This re-evaluates command 28. Command 30 is `v(28)`. This uses the value (result) of command 28. In the cases of a pure function (a function with
no side effects), the result is the same. For a function with side effects, the result can be different.

The next few commands show some record manipulation. It is assumed that `ex.erl` defines a record like this:

```erlang
-record(rec, {a, b = val()}).
```

```erlang
val() ->
    3.
```

Commands 31 and 32 compiles the file `ex.erl` and reads the record definitions in `ex.beam`. If the compiler did not output any record definitions on the BEAM file, `rr(ex)` tries to read record definitions from the source file instead.

Command 33 prints the definition of the record named `rec`.

Command 34 tries to create a `rec` record, but fails since the function `val/0` is undefined. Command 35 shows the workaround: explicitly assign values to record fields that cannot otherwise be initialized.

Command 36 prints the newly created record using record definitions maintained by the shell.

Command 37 defines a record directly in the shell. The definition replaces the one read from the file `ex.beam`.

Command 38 creates a record using the new definition, and prints the result.

Command 39 and 40 show that record definitions are updated as side effects. The evaluation of the command fails but the definition of `rec` has been carried out.

For the next command, it is assumed that `test1:loop(N)` is defined in the following way:

```erlang
loop(N) ->
    io:format("Hello Number: ~w~n", [N]),
    loop(N+1).
```

Command 41 evaluates `test1:loop(0)`, which puts the system into an infinite loop. At this point the user types `Control G`, which suspends output from the current process, which is stuck in a loop, and activates JCL mode. In JCL mode the user can start and stop jobs.

In this particular case, the `i` command ("interrupt") is used to terminate the looping program, and the `c` command is used to connect to the shell again. Since the process was running in the background before we killed it, there will be more printouts before the "** exception exit: killed" message is shown.

Command 42 creates an ETS table.

Command 43 tries to insert a tuple into the ETS table but the first argument (the table) is missing. The exception kills the evaluator process.

Command 44 corrects the mistake, but the ETS table has been destroyed since it was owned by the killed evaluator process.

Command 46 sets the exception handling of the evaluator process to `true`. The exception handling can also be set when starting Erlang, like this: `erl -stdlib shell -catch_exception true`.

Command 48 makes the same mistake as in command 43, but this time the evaluator process lives on. The single star at the beginning of the printout signals that the exception has been caught.
Command 49 successfully inserts the tuple into the ETS table.
The halt() command exits the Erlang runtime system.

J CL Mode

When the shell starts, it starts a single evaluator process. This process, together with any
local processes which it spawns, is referred to as a job. Only the current job, which is
said to be connected, can perform operations with standard IO. All other jobs, which
are said to be detached, are blocked if they attempt to use standard IO.
All jobs which do not use standard IO run in the normal way.
The shell escape key `^G (Control G)` detaches the current job and activates JCL mode.
The JCL mode prompt is "-- > ". If "?" is entered at the prompt, the following help
message is displayed:

|-- > ?
c [nn] - connect to job
i [nn] - interrupt job
k [nn] - kill job
j - list all jobs
s [shell] - start local shell
r [node [shell]] - start remote shell
q - quit erlang
? | h - this message

The JCL commands have the following meaning:
c [nn] Connects to job number <nn> or the current job. The standard shell is
resumed. Operations which use standard IO by the current job will be interleaved
with user inputs to the shell.
i [nn] Stops the current evaluator process for job number nn or the current job, but
does not kill the shell process. Accordingly, any variable bindings and the process
dictionary will be preserved and the job can be connected again. This command
can be used to interrupt an endless loop.
k [nn] Kills job number nn or the current job. All spawned processes in the job are
killed, provided they have not evaluated the group_leader/1 BIF and are located
on the local machine. Processes spawned on remote nodes will not be killed.
j Lists all jobs. A list of all known jobs is printed. The current job name is prefixed
with '*'.
s Starts a new job. This will be assigned the new index [nn] which can be used in
references.
s [shell] Starts a new job. This will be assigned the new index [nn] which can be
used in references. If the optional argument shell is given, it is assumed to be a
module that implements an alternative shell.
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. If the optional argument shell is given, it is assumed to be a
module that implements an alternative shell.
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).
Displaces this message.

It is possible to alter the behavior of shell escape by means of the STDLIB application variable shell_esc. The value of the variable can be either jcl (erl -stdlib shell_esc jcl) or abort (erl -stdlib shell_esc abort). The first option sets ^G to activate JCL mode (which is also default behavior). The latter sets ^G to terminate the current shell and start a new one. JCL mode cannot be invoked when shell_esc is set to abort.

If you want an Erlang node to have a remote job active from the start (rather than the default local job), you start Erlang with the -remsh flag. Example: erl -sname this_node -remsh other_node@other_host

Restricted Shell

The shell may be started in a restricted mode. In this mode, the shell evaluates a function call only if allowed. This feature makes it possible to, for example, prevent a user from accidentally calling a function from the prompt that could harm a running system (useful in combination with the the system flag +Bi).

When the restricted shell evaluates an expression and encounters a function call or an operator application, it calls a callback function (with information about the function call in question). This callback function returns true to let the shell go ahead with the evaluation, or false to abort it. There are two possible callback functions for the user to implement:

\[
\text{local}\_\text{allowed}(Func, ArgList, State) \rightarrow \{true, NewState\} \mid \{false, NewState\}
\]

to determine if the call to the local function Func with arguments ArgList should be allowed.

\[
\text{non}\_\text{local}\_\text{allowed}(FuncSpec, ArgList, State) \rightarrow \{true, NewState\} \mid \{false, NewState\} \mid \{[\text{redirect}, \text{NewFuncSpec}, \text{NewArgList}], \text{NewState}\}
\]

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 139] manual page. (Arguments in ArgList are evaluated before the callback functions are called.)

The State argument is a tuple \{ShellState, ExprState\}. The return value NewState has the same form. This may be used to carry a state between calls to the callback functions. Data saved in ShellState lives through an entire shell session. Data saved in ExprState lives only through the evaluation of the current expression.

There are two ways to start a restricted shell session:

- Use the STDLIB application variable restricted_shell and specify, as its value, the name of the callback module. Example (with callback functions implemented in callback_mod.erl): $ erl -stdlib restricted_shell callback_mod

- From a normal shell session, call function shell:start_restricted/1. This exits the current evaluator and starts a new one in restricted mode.

Notes:
When restricted shell mode is activated or deactivated, new jobs started on the node will run in restricted or normal mode respectively.

If restricted mode has been enabled on a particular node, remote shells connecting to this node will also run in restricted mode.

The callback functions cannot be used to allow or disallow execution of functions called from compiled code (only functions called from expressions entered at the shell prompt).

Errors when loading the callback module is handled in different ways depending on how the restricted shell is activated:

- If the restricted shell is activated by setting the kernel variable during emulator startup and the callback module cannot be loaded, a default restricted shell allowing only the commands q() and init:stop() is used as fallback.
- If the restricted shell is activated using shell: start_restricted/1 and the callback module cannot be loaded, an error report is sent to the error logger and the call returns {error, Reason}.

Exports

history(N) -> integer()

Types:

- N = integer()

Sets the number of previous commands to keep in the history list to N. The previous number is returned. The default number is 20.

results(N) -> integer()

Types:

- N = integer()

Sets the number of results from previous commands to keep in the history list to N. The previous number is returned. The default number is 20.

catch_exception(Bool) -> Bool

Types:

- Bool = bool()

Sets the exception handling of the evaluator process. The previous exception handling is returned. The default (false) is to kill the evaluator process when an exception occurs, which causes the shell to create a new evaluator process. When the exception handling is set to true the evaluator process lives on which means that for instance ports and ETS tables as well as processes linked to the evaluator process survive the exception.

start_restricted(Module) -> ok | {error, Reason}

Types:

- Module = atom()
- Reason = atom()
Exits a normal shell and starts a restricted shell. Module specifies the callback module for the functions `local_allowed/3` and `non_local_allowed/3`. The function is meant to be called from the shell.

If the callback module cannot be loaded, an error tuple is returned. The Reason in the error tuple is the one returned by the code loader when trying to load the code of the callback module.

```
stop_restricted() -> ok
```

Exits a restricted shell and starts a normal shell. The function is meant to be called from the shell.
shell_default

Erlang Module

The functions in shell_default are called when no module name is given in a shell command. Consider the following shell dialogue:

1 > lists:reverse("abc").
   "cba"
2 > c(foo).
   {ok, foo}

In command one, the module lists is called. In command two, no module name is specified. The shell searches the modules user_default followed by shell_default for the function foo/1.

shell_default is intended for “system wide” customizations to the shell. user_default is intended for “local” or individual user customizations.

Hint

To add your own commands to the shell, create a module called user_default and add the commands you want. Then add the following line as the first line in your .erlang file in your home directory.

code:loadabs("$PATH/user_default").

$PATH is the directory where your user_default module can be found.
slave

Erlang Module

This module provides functions for starting Erlang slave nodes. All slave nodes which are started by a master will terminate automatically when the master terminates. All TTY output produced at the slave will be sent back to the master node. File I/O is done via the master.

Slave nodes on other hosts than the current one are started with the program `rsh`. The user must be allowed to `rsh` to the remote hosts without being prompted for a password. This can be arranged in a number of ways (refer to the `rsh` documentation for details). A slave node started on the same host as the master inherits certain environment values from the master, such as the current directory and the environment variables. For what can be assumed about the environment when a slave is started on another host, read the documentation for the `rsh` program.

An alternative to the `rsh` program can be specified on the command line to `erl` as follows: `-rsh Program`.

The slave node should use the same file system at the master. At least, Erlang/OTP should be installed in the same place on both computers and the same version of Erlang should be used.

Currently, a node running on Windows NT can only start slave nodes on the host on which it is running.

The master node must be alive.

Exports

\[
\begin{align*}
\text{start(Host)} & \Rightarrow \\
\text{start(Host, Name)} & \Rightarrow \\
\text{start(Host, Name, Args)} & \Rightarrow \text{ok, Node} \mid \text{error, Reason}
\end{align*}
\]

Types:
- `Host = Name = atom()`
- `Args = string()`
- `Node = node()`
- `Reason = timeout \mid \text{no}_rsh \mid \text{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, \texttt{Node}\}, where \texttt{Node} is the name of the new node. Otherwise it returns \{error, \texttt{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.

- \{already_running, \texttt{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} = \texttt{Name} = \texttt{atom()}
- \texttt{Args} = \texttt{string()}
- \texttt{Node} = \texttt{node()}
- \texttt{Reason} = \texttt{timeout} | \texttt{no_rsh} | \{already_running, \texttt{Node}\}

Starts a slave node in the same way as \texttt{start/1,2,3}, except that the slave node is linked to the currently executing process. If that process terminates, the slave node also terminates.

See \texttt{start/1,2,3} for a description of arguments and return values.
stop(Node) -> ok

Types:
- Node = node()

Stops (kills) a node.

pseudo([Master | ServerList]) -> ok

Types:
- Master = node()
- ServerList = [atom()]

Calls pseudo(Master, ServerList). If we want to start a node from the command line and set up a number of pseudo servers, an Erlang runtime system can be started as follows:

```erl
% 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:

```erl
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 \( B = \{ x : A : S(x) \} \). 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 \( \{ (x, y) : x \in X \text{ and } y \in Y \} \). A relation is a subset of \( X \times Y \). Let R be a relation. The fact that \( (x, y) \) belongs to R is written as \( xRy \). 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 \( \{ x : xRy \text{ for some } y \} \). The range of R is the set \( \{ y : xRy \text{ for some } x \} \). The converse of R is the set \( \{ (y, x) : (x, y) \in R \} \). If A is a subset of X, then the image of A under R is the set \( \{ y : xRy \text{ for some } x \in A \} \), and 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 $XY$, such that the domain of $F$ is equal to $X$ and such that for every $x$ in $X$ there is a unique element $y$ in $Y$ with $(x,y)$ in $F$. The latter condition can be formulated as follows: if $xFy$ and $xFz$ then $y=z$. In this module, it will not be required that the domain of $F$ be equal to $X$ for a relation to be considered a function. Instead of writing $(x,y)\in F$ or $xFy$, we write $F(x)=y$ when $F$ is a function, and say that $F$ maps $x$ onto $y$, or that the value of $F$ at $x$ is $y$. Since functions are relations, the definitions of the last paragraph (domain, range, and so on) apply to functions as well. If the converse of a function $F$ is a function $F'$, then $F'$ is called the inverse of $F$.

The relative product of two functions $F_1$ 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 $(n$-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 $n$-ary relation $R$ onto coordinate $i$ is the set $\{x[i]: (x[1],...,x[i],...,x[n]) \in R \}$ for some $x[j] \in X[j]$, $1 \leq j \leq n$ and not $i=j$. The projections of a binary relation $R$ onto the first and second coordinates are the domain and the range of $R$ respectively. The relative product of binary relations can be generalized to $n$-ary relations as follows. Let $TR$ be an ordered set $(R[1],...,R[n])$ of binary relations from $X[i]$ to $Y[i]$ and $S$ a binary relation from $(Y[1]...Y[n])$ to $Z$. The relative product of $TR$ and $S$ is the binary relation $T$ from $X$ to $Z$ defined so that $xTz$ if and only if there exists an element $y[i]$ in $Y[i]$ for each $1 \leq i \leq n$ such that $xR[i]y[i]$ and $([y[1],...,y[n]])S$. Now let $TR$ be an ordered set $(R[1],...,R[n])$ of binary relations from $X[i]$ to $Y[i]$ and $S$ a subset of $X[1]...X[n]$. The multiple relative product of $TR$ and $S$ is defined to be the set $\{z, z= ((x[1],...,x[n]), (y[1],...,y[n])) \}$ for some $((x[1],...,x[n]))S$ and for some $(x[i],y[i])$ in $R[i]$, $1 \leq i \leq n$. The natural join of an $n$-ary relation $R$ and an $m$-ary relation $S$ on coordinate $i$ and $j$ is defined to be the set $\{z:$...
The sets recognized by this module will be represented by elements of the relation `Sets`, defined as the smallest set such that:

- for every atom `T` except `.` and for every term `X`, `(T,X)` belongs to `Sets` (atomic sets);
- `(['.'],[])` belongs to `Sets` (the untyped empty set);
- for every tuple `T = {T[1],...,T[n]}` and for every tuple `X = {X[1],...,X[n]}`, if `(T[i],X[i])` belongs to `Sets` for every `1 <= i <= n` then `(T,X)` belongs to `Sets` (ordered sets);
- for every term `T`, if `X` is the empty list or a non-empty sorted list `[X[1],...,X[n]]` without duplicates such that `(T,X[i])` belongs to `Sets` for every `1 <= i <= n`, then `([T],X)` belongs to `Sets` (typed unordered sets).

An external set is an element of the range of `Sets`. A type is an element of the domain of `Sets`. If `S` is an element `(T,X)` of `Sets`, then `T` is a valid type of `X`, `T` is the type of `S`, and `X` is the external set of `S`. From `term/2` [page 426] 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 433], checks that the arguments represent binary relations by matching `[{A,B}]` against the type of the first argument (Arg1 say), and `[{C,D}]` against the type of the second argument (Arg2 say). The fact that `[{A,B}]` matches the type of Arg1 is to be interpreted as Arg1 representing a binary relation from `X` to `Y`, where `X` is defined as all sets `Set(x)` for some element `x` in `Sets` the type of which is `A`, and similarly for `Y`. In the same way Arg2 is interpreted as representing a binary relation from `W` to `Z`. Finally it is checked that `B` matches `C`, which is sufficient to ensure that `W` is equal to `Y`. The untyped empty set is handled separately: its type, `['.' ]`, matches the type of any unordered set.

A few functions of this module (drestriction/3, family_projection/2, partition/2, partition_family/2, projection/2, restriction/3, substitution/2) accept an Erlang function as a means to modify each element of a given unordered set. Such a function, called `SetFun` in the following, can be specified as a functional object (fun), a tuple `{external,Fun}`, or an integer. If `SetFun` is specified as a fun, the fun is applied to each element of the given set and the return value is
assumed to be a set. If SetFun is specified as a tuple \(\text{external, Fun}\), Fun is applied to
the external set of each element of the given set and the return value is assumed to be
an external set. Selecting the elements of an unordered set as external sets and
assembling a new unordered set from a list of external sets is in the present
implementation more efficient than modifying each element as a set. However, this
optimization can only be utilized when the elements of the unordered set are atomic or
ordered sets. It must also be the case that the type of the elements matches some clause
of Fun (the type of the created set is the result of applying Fun to the type of the given
set), and that Fun does nothing but selecting, duplicating or rearranging parts of the
elements. Specifying a SetFun as an integer \(I\) is equivalent to specifying \(\text{external, fun(X) -
}\) \(\text{element}(I,X)\), but is to be preferred since it makes it possible to handle
this case even more efficiently. Examples of SetFuns:

\[
\begin{align*}
\{\text{sofs, union}\} \\
\text{fun}(S) & \rightarrow \text{sofs:partition}(1, S) \text{ end} \\
\{\text{external, fun(A) -
}\} & \rightarrow \text{A end} \\
\{\text{external, fun(\{A, \_\},C) -
}\} & \rightarrow \{C, A\} \text{ end} \\
\{\text{external, fun(\_,\{\_,\}_C\}) -
}\} & \rightarrow \text{C end} \\
\{\text{external, fun(\_,\{\_,\{\_\},E\}_C\}) -
}\} & \rightarrow \{E, \{E, C\}\} \text{ end}\}
\end{align*}
\]

2

The order in which a SetFun is applied to the elements of an unordered set is not
specified, and may change in future versions of sofs.

The execution time of the functions of this module is dominated by the time it takes to
sort lists. When no sorting is needed, the execution time is in the worst case
proportional to the sum of the sizes of the input arguments and the returned value. A
few functions execute in constant time: from external, is empty set, is set,
is sofs set, to external, type.

The functions of this module exit the process with a badarg, bad function, or
type mismatch message when given badly formed arguments or sets the types of which
are not compatible.

Types

\[
\begin{align*}
\text{anyset}() & = \text{an unordered, ordered or atomic set}\- \\
\text{binary relation()} & = \text{a binary relation}\- \\
\text{bool}() & = \text{true | false}\- \\
\text{external set}() & = \text{an external set}\- \\
\text{family}() & = \text{a family (of subsets)}\- \\
\text{function}() & = \text{a function}\- \\
\text{ordset}() & = \text{an ordered set}\- \\
\text{relation}() & = \text{an n-ary relation}\- \\
\text{set}() & = \text{an unordered set}\- \\
\text{set of sets}() & = \text{an unordered set of set}\- \\
\text{set fun}() & = \text{integer}() \geq 1 \\
| \{\text{external, fun(external set) -
}\} & \rightarrow \text{external set}\} \\
| \text{fun(anyset()) -
}\} & \rightarrow \text{anyset}\} \\
\text{spec fun}() & = \{\text{external, fun(external set) -
}\} \rightarrow \text{bool}\} \\
| \text{fun(anyset()) -
}\} & \rightarrow \text{bool}\} \\
\text{type}() & = \text{a type}\-
\end{align*}
\]
Exports

\[ \text{a_function(Tuples [, Type]) -> Function} \]

Types:
- \( \text{Function} = \text{function}() \)
- \( \text{Tuples} = [\text{tuple()}] \)
- \( \text{Type} = \text{type}() \)

Creates a function [page 416]. \( \text{a_function(F,T)} \) is equivalent to \( \text{from_term(F,T)} \), if the result is a function. If no type [page 417] is explicitly given, \([\text{atom,atom}]\) is used as type of the function.

\[ \text{canonical_relation(SetOfSets) -> BinRel} \]

Types:
- \( \text{BinRel} = \text{binary_relation}() \)
- \( \text{SetOfSets} = \text{set_of_sets}() \)

Returns the binary relation containing the elements \((E,\text{Set})\) such that \( \text{Set} \) belongs to \( \text{SetOfSets} \) and \( E \) belongs to \( \text{Set} \). If \( \text{SetOfSets} \) is a partition [page 416] of a set \( X \) and \( R \) is the equivalence relation in \( X \) induced by \( \text{SetOfSets} \), then the returned relation is the canonical map [page 416] from \( X \) onto the equivalence classes with respect to \( R \).

\[
\text{Ss} = \text{sofs:from_term}([[[a,b],[b,c]]]),
\text{CR} = \text{sofs:canonical_relation(Ss)},
\text{sofs:to_external(CR)}.
\]

\[ \text{composite(Function1, Function2) -> Function3} \]

Types:
- \( \text{Function1} = \text{Function2} = \text{Function3} = \text{function}() \)

Returns the composite [page 416] of the functions \( \text{Function1} \) and \( \text{Function2} \).

\[
\text{F1} = \text{sofs:a_function}([[[a,1],[b,2],[c,2]]]),
\text{F2} = \text{sofs:a_function}([[[1,x],[2,y],[3,z]]]),
\text{F} = \text{sofs:composite(F1, F2)},
\text{sofs:to_external(F)}.
\]

\[ \text{constant_function(Set, AnySet) -> Function} \]

Types:
- \( \text{AnySet} = \text{anyset}() \)
- \( \text{Function} = \text{function}() \)
- \( \text{Set} = \text{set}() \)

Creates the function [page 416] that maps each element of the set \( \text{Set} \) onto \( \text{AnySet} \).
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()

Returns the converse [page 415] 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 415] 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 416] 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 417] 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 415] 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 415] of BinRel1 to Set.

```erl
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.

```erl
1> SetFun = {external, fun({A,B,C}) -> {B,C} end},
   R1 = sofs:relation([[a,aa,1],[b,bb,2],[c,cc,3]]),
   R2 = sofs:restriction([[b,2],[cc,3],[dd,4]]),
   R3 = sofs:drestriction(SetFun, R1, R2),
   sofs:to external(R3).
[[a,aa,1]]
```

drestriction(F,S1,S2) is equivalent to difference(S1, restriction(F,S1,S2)).

empty_set() -> Set

Types:

• Set = set()

Returns the untyped empty set [page 417]. empty_set() is equivalent to from_term([], []).

extension(BinRel1, Set, AnySet) -> BinRel2

Types:

• AnySet = anyset()
• BinRel1 = BinRel2 = binary_relation()
• Set = set()

Returns the extension [page 415] of BinRel1 such that for each element E in Set that does not belong to the domain [page 415] of BinRel1, BinRel2 contains the pair (E, AnySet).

```erl
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 416]. family(F,T) is equivalent to from_term(F,T), if the result is a family. If no type [page 417] 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 416], then Family3 is the family such that the index set is equal to the index set of Family1, and Family3[i] is the difference between Family1[i] and Family2[i] if Family2 maps i, Family1[i] otherwise.

1> F1 = sofs:family([[a, [1,2]], [b, [3,4]]]),
   F2 = sofs:family([[b, [4,5]], [c, [6,7]]]),
   F3 = sofs:family_difference(F1, F2),
   sofs:to_external(F3).
   [{a, [1,2]}, [b, [3]]]

family_domain(Family1) -> Family2

Types:
- Family1 = Family2 = family()

If Family1 is a family [page 416] 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 415] of Family1[i].

1> FR = sofs:from_term([[a, [[1,a], [2,b], [3,c]]], [b, []], [c, [[4,d], [5,e]]]]),
   F = sofs:family_domain(FR),
   sofs:to_external(F).
   [{a, [1,2,3]}, [b, []], [c, [4,5]]]

family_field(Family1) -> Family2

Types:
- Family1 = Family2 = family()

If Family1 is a family [page 416] 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 415] of Family1[i].

1> FR = sofs:from_term([[a, [[1,a], [2,b], [3,c]]], [b, []], [c, [[4,d], [5,e]]]]),
   F = sofs:family_field(FR),
   sofs:to_external(F).
   [{a, [1,2,3, a,b,c]}, [b, []], [c, [4,5,d,e]]]

family_field(Family1) is equivalent to family_union(family_domain(Family1),
family_range(Family1)).
family_intersection(Family1) -> Family2

Types:
- Family1 = Family2 = family()

If Family1 is a family [page 416] 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 415] of Family1[i].

If Family1[i] is an empty set for some i, then the process exits with a badarg message.

```prolog
1> F1 = sofs:from_term([[a,[[1,2,3],[2,3,4]]],[b,[[x,y,z],[x,y]]]]),
   F2 = sofs:family_intersection(F1),
   sofs:to_external(F2).
[[[a,[2,3]],b,[x,y]]]
```

family_intersection(Family1, Family2) -> Family3

Types:
- Family1 = Family2 = Family3 = family()

If Family1 and Family2 are families [page 416], then Family3 is the family such that the index set is the intersection of Family1's and Family2's index sets, and Family3[i] is the intersection of Family1[i] and Family2[i].

```prolog
1> F1 = sofs:family([[a,[1,2]],[b,[3,4]],[c,[5,6]]]),
   F2 = sofs:family([[b,[4,5]],[c,[7,8]],[d,[9,10]]]),
   F3 = sofs:family_intersection(F1, F2),
   sofs:to_external(F3).
[[[b,[4]],c,[[]]]]
```

c-family_projection(SetFun, Family1) -> Family2

Types:
- SetFun = set_fun()
- Family1 = Family2 = family()
- Set = set()

If Family1 is a family [page 416] then Family2 is the family with the same index set as Family1 such that Family2[i] is the result of calling SetFun with Family1[i] as argument.

```prolog
1> F1 = sofs:from_term([[a,[[1,2],[2,3]]],[b,[]]]),
   F2 = sofs:family_projection([sofs, union], F1),
   sofs:to_external(F2).
[[[a,[1,2,3]],b,[[]]]]
```

c-family_range(Family1) -> Family2

Types:
- Family1 = Family2 = family()

If Family1 is a family [page 416] 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 415] of Family1[i].

```prolog
```
family_specification(Fun, Family1) -> Family2

Types:
- Fun = spec_fun()
- Family1 = Family2 = family()

If Family1 is a family [page 416], then Family2 is the restriction [page 415] 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 417] 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 416] Family. For each pair (a, \{b[1],...,b[n]\}) of Family, the vertex a as well the edges (a,b[i]) for 1\leq i \leq n are added to a newly created directed graph.

If no graph type is given, digraph:new/1 is used for creating the directed graph, otherwise the GraphType argument is passed on as second argument to digraph:new/2.

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 416], then BinRel is the binary relation containing all pairs (i,x) such that i belongs to the index set of Family and x belongs to Family[i].

1> F = sofs:family([[a,[]],[b,[1]],[c,[2,3]]]),
   R = sofs:family_to_relation(F),
   sofs:to_external(R).
   [{b,1},{c,2},{c,3}]
family_union(Family1) -> Family2

Types:

- Family1 = Family2 = family()

If Family1 is a family [page 416] 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 415] of Family1[i].

```erlang
1> F1 = sofs:from_term([[a,[[1,2],[2,3]]],[b,[[[]]]]]),
F2 = sofs:family_union(F1),
sofs:to_external(F2).
[{[a,[[1,2],[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 416], 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.

```erlang
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).
```

field(BinRel) -> Set

Types:

- BinRel = binary_relation()
- Set = set()

Returns the field [page 415] of the binary relation BinRel.

```erlang
1> R = sofs:relation([[i,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 417] ExternalSet and the type [page 417] Type. It is assumed that Type is a valid type [page 417] of ExternalSet.

```erlang
from_sets(ListOfSets) -> Set
```
Types:
- `Set = set()`
- `ListOfSets = [anyset()]`

Returns the unordered set [page 417] containing the sets of the list `ListOfSets`.

```erlang
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 417] containing the sets of the non-empty tuple `TupleOfSets`.

```erlang
from_sets(TupleOfSets) -> Ordset.
```

### from_term(Term [, Type]) -> AnySet

Types:
- `AnySet = anyset()`
- `Term = term()`
- `Type = type()`

Creates an element of Sets [page 417] by traversing the term `Term`, sorting lists, removing duplicates and deriving or verifying a valid type [page 417] for the obtained external set. An explicitly given type [page 417] `Type` can be used to limit the depth of the traversal; an atomic type stops the traversal, as demonstrated by this example where “foo” and “foo” are left unmodified:

```erlang
1> S = sofs:from_term([["foo"],[1,1]],["foo","[2,2]"],[[atom, [atom]]]),
   sofs:to_external(S).
[ ["foo"],[1],["foo","[2,2]"
```

### from_term can be used for creating atomic or ordered sets. The only purpose of such a set is that of later building unordered sets since all functions in this module that do anything operate on unordered sets. Creating unordered sets from a collection of ordered sets may be the way to go if the ordered sets are big and one does not want to waste heap by rebuilding the elements of the unordered set. An example showing that a set can be built “layer by layer”:

```erlang
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`.

```erl
image(BinRel, Set1) -> Set2

Types:
  • BinRel = binary_relation()
  • Set1 = Set2 = set()

Returns the image [page 415] of the set `Set1` under the binary relation `BinRel`.

1> R = sofs:relation([[[1,a],[2,b],[2,c],[3,d]]]),
   S1 = sofs:set([1,2]),
   S2 = sofs:image(R, S1),
   sofs:to_external(S2).
[a,b,c]
```

```erl
intersection(SetOfSets) -> Set

Types:
  • Set = set()
  • SetOfSets = set_of_sets()

Returns the intersection [page 415] of the set of sets `SetOfSets`.
Intersecting an empty set of sets exits the process with a `badarg` message.

intersection(Set1, Set2) -> Set3

Types:
  • Set1 = Set2 = Set3 = set()

Returns the intersection [page 415] of `Set1` and `Set2`.

intersection_of_family(Family) -> Set

Types:
  • Family = family()
  • Set = set()

Returns the intersection of the family [page 416] `Family`.
Intersecting an empty family exits the process with a `badarg` message.

1> F = sofs:family([[[a,[0,2,4]], [b,[0,1,2]], [c,[2,3]]]]),
   S = sofs:intersection_of_family(F),
   sofs:to_external(S).
[2]
```

```erl
inverse(Function1) -> Function2

Types:
  • Function1 =Function2 = function()

Returns the inverse [page 416] of the function `Function1`.
```
1> R1 = sofs:relation([[1,a],[2,b],[3,c]]),
   R2 = sofs:inverse(R1),
   sofs:to_external(R2),
   [{a,1},{b,2},{c,3}]

inverse_image(BinRel, Set1) -> Set2

Types:
  • BinRel = binary_relation()
  • Set1 = Set2 = set()

Returns the inverse image [page 415] 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 416] 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 415], 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 415], false otherwise.

is_set(AnySet) -> Bool

Types:
- AnySet = anyset()
- Bool = bool()

Returns true if AnySet is an unordered set [page 417], 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 417], 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 415] of Set2, false otherwise.

is_type(Term) -> Bool

Types:
- Bool = bool()
- Term = term()

Returns true if the term Term is a type [page 417].

join(Relation1, I, Relation2, J) -> Relation3

Types:
- Relation1 = Relation2 = Relation3 = relation()
- I = J = integer() ≥ 0

Returns the natural join [page 416] of the relations Relation1 and Relation2 on coordinates I and J.

R1 = sofs:relation([a,x,1,b,y,2]),
R2 = sofs:relation([1,f,g],[1,h,i],[2,3,4]),
J = sofs:join(R1, 3, R2, 1),
sofs:to_external(J).

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 416] of the ordered set (R[i],...,R[n]) and BinRel1.
1> Ri = sofs:relation([[[a,1],[b,2],[c,3]]]),
R = sofs:relation([[[a,b],[b,c],[c,a]]]),
MP = sofs:multiple_relative_product([Ri, Ri], R),
sofs:to_external(sofs:range(MP)).
[{{1,2},{2,3},{3,1}}]

no_elements(ASet) -> NoElements
Types:
• 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 416] 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 416] 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 416] of
Set1. If the result of applying SetFun to an element of Set1 yields an element in Set2,
the element belongs to Set3, otherwise the element belongs to Set4.
1> R1 = sofs:relation([[1,a],[2,b],[3,c]]),
   S = sofs:set([2,4,6]),
   [R2,R3] = sofs:partition(1, R1, S),
   [sofs:to_external(R2),sofs:to_external(R3)].

\[
\begin{align*}
\text{partition(F,S1,S2)} & \text{ is equivalent to } \{\text{restriction(F,S1,S2)}, \\
& \text{drestriction(F,S1,S2)}\}.
\end{align*}
\]

\[
\text{partition_family(SetFun, Set) } \rightarrow \text{ Family}
\]

Types:
- Family = family()
- SetFun = set_fun()
- Set = set()

Returns the family [page 416] Family where the indexed set is a partition [page 416] 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 416].

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).

\[
\begin{align*}
\{\{a,a\},\{a,a,a,a\}\},\{\{a,b\},\{a,a,b,b\}\},
\end{align*}
\]

\[
\text{product(TupleOfSets)} \rightarrow \text{ Relation}
\]

Types:
- Relation = relation()
- TupleOfSets = tuple-of(set())

Returns the Cartesian product [page 416] of the non-empty tuple of sets TupleOfSets. If \(x[1],...,x[n]\) is an element of the n-ary relation Relation, then \(x[i]\) is drawn from element \(i\) of TupleOfSets.

1> S1 = sofs:set([a,b]),
   S2 = sofs:set([1,2]),
   S3 = sofs:set([x,y]),
   P3 = sofs:product([S1,S2,S3]),
   sofs:to_external(P3).

\[
\begin{align*}
\{\{a,1\},\{a,1\},\{a,2\},\{a,2,y\},\{b,1\},\{b,1\},\{b,2\},\{b,2\}\}
\end{align*}
\]

\[
\text{product(Set1, Set2)} \rightarrow \text{ BinRel}
\]

Types:
- BinRel = binary_relation()
- Set1 = Set2 = set()

Returns the Cartesian product [page 415] of Set1 and Set2.

1> S1 = sofs:set([1,2]),
   S2 = sofs:set([a,b]),
   R = sofs:product(S1, S2),
   sofs:to_external(R).

\[
\begin{align*}
\{\{1,a\},\{1,b\},\{2,a\},\{2,b\}\}
\end{align*}
\]
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 416] 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 415] 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 415]. relation(R, T) is equivalent to from_term(R, T), if T is a type [page 417] 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 416] Family such that the index set is equal to the domain [page 415] of the binary relation BinRel, and Family[i] is the image [page 415] 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 416] of the ordered set \( (R[i],...,R[n]) \) and BinRel1.

If BinRel1 is omitted, the relation of equality between the elements of the Cartesian product [page 416] of the ranges of \( R[i],\) rangeR[1]...rangeR[n], is used instead (intuitively, nothing is “lost”).

1> TR = sofs:relation([[1,a],[1,aa],[2,b]]),
   R1 = sofs:relation([[1,u],[2,v],[3,c]]),
   R2 = sofs:relative_product([TR, R1]),
   sofs:to_external(R2).
   [{1,[a,u]},{1,[aa,u]},{2,[b,v]}]

Note that relative_product([R1],R2) is different from relative_product(R1,R2); the tuple of one element is not identified with the element itself.

relative_product(BinRel1, BinRel2) -> BinRel3

Types:
- BinRel1 = BinRel2 = BinRel3 = binary_relation()

Returns the relative product [page 415] of the binary relations BinRel1 and BinRel2.

relative_product1(BinRel1, BinRel2) -> BinRel3

Types:
- BinRel1 = BinRel2 = BinRel3 = binary_relation()

Returns the relative product [page 415] of the converse [page 415] 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 415] of the binary relation BinRel1 to Set.
1> R1 = sofs:relation([\([1,a],[2,b],[3,c]\)],
S = sofs:set([\([1,2,4]\)]),
R2 = sofs:restriction(R1, S),
sofs:to_external(R2).
\([\{1,a\},\{2,b\}]\)

restriction(SetFun, Set1, Set2) -> Set3

Types:
- SetFun = \(\text{set fun()}\)
- Set1 = Set2 = Set3 = set()

Returns a subset of Set1 containing those elements that yield an element in Set2 as the result of applying SetFun.

1> S1 = sofs:relation([\([1,a],[2,b],[3,c]\)],
S2 = sofs:set([\([b,c,d]\)]),
S3 = sofs:restriction(2, S1, S2),
sofs:to_external(S3).
\([\{2,b\},\{3,c\}]\)

set(Terms [, Type]) -> Set

Types:
- Set = set()
- Terms = [term()]
- Type = \(\text{type}()\)

Creates an unordered set [page 417]. set(L,T) is equivalent to from_term(L,T), if the result is an unordered set. If no type [page 417] is explicitly given, [atom] is used as type of the set.

specification(Fun, Set1) -> Set2

Types:
- Fun = \(\text{spec fun()}\)
- 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 417] of each element, otherwise Fun is applied to each element.

1> R1 = sofs:relation([\([a,1],[b,2]\)]),
R2 = sofs:relation([\([x,1],[x,2],[y,3]\)]),
S1 = sofs:from_sets([\([R1,R2]\)]),
S2 = sofs:specification([sofs,is_a_function], S1),
sofs:to_external(S2).
\([\{\{a,1\},\{b,2\}\}]\)

strict_relation(BinRel1) -> BinRel2

Types:
- BinRel1 = BinRel2 = binary_relation()

Returns the strict relation [page 416] corresponding to the binary relation BinRel1.
1> R1 = sofs:relation([[1,1],[1,2],[2,1],[2,2]]),
R2 = sofs:strict_relation(R1),
sofs:to_external(R2).
[[1,2],[2,1]]

substitution(SetFun, Set1) -> Set2

Types:
  • SetFun = set_fun()
  • Set1 = Set2 = set()

Returns a function, the domain of which is Set1. The value of an element of the domain
is the result of applying SetFun to the element.

1> L = [[a,1],[b,2]].
[[a,1],[b,2]]
2> sofs:to_external(sofs:projection(1,sofs:relation(L))).
[a,b]
3> sofs:to_external(sofs:substitution(1,sofs:relation(L))).
[['[a,1],a'],['[b,2],b']]
4> SetFun = {external, fun([A,] =E) -> [E,A] end},
sofs:to_external(sofs:projection(SetFun,sofs:relation(L))).
[['[a,1],a'],['[b,2],b']]

The relation of equality between the elements of \{a,b,c\}:

1> I = sofs:substitution(fun(A) -> A end, sofs:set([a,b,c])),
sofs:to_external(I).
[[a,a],[b,b],[c,c]]

Let SetOfSets be a set of sets and BinRel a binary relation. The function that maps each
element Set of SetOfSets onto the image 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 (or the Boolean sum) of Set1 and Set2.
sofs

1> S1 = sofs:set([1,2,3]),
   S2 = sofs:set([2,3,4]),
   P = sofs:symdiff(S1, S2),
   sofs:to_external(P).
[1,4]

symmetric_partition(Set1, Set2) => [Set3, Set4, Set5]
Types:
  • Set1 = Set2 = Set3 = Set4 = Set5 = set()
Returns a triple of sets: Set3 contains the elements of Set1 that do not belong to Set2;
  Set4 contains the elements of Set1 that belong to Set2; Set5 contains the elements of
  Set2 that do not belong to Set1.

to_external(AnySet) => ExternalSet
Types:
  • ExternalSet = external_set()
  • AnySet = anyset()
Returns the external set [page 417] 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 417] of an atomic, ordered or unordered set.

union(SetOfSets) => Set
Types:
  • Set = set()
  • SetOfSets = set_of_sets()
Returns the union [page 415] of the set of sets SetOfSets.

union(Set1, Set2) => Set3
Types:
  • Set1 = Set2 = Set3 = set()
Returns the union [page 415] of Set1 and Set2.
union_of_family(Family) -> Set

Types:
- Family = family()
- Set = set()

Returns the union of the family [page 416] 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 416] W corresponding to the binary relation BinRel1. Let F be the field [page 415] 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 120], digraph(3) [page 125], orddict(3) [page 296], ordsets(3) [page 301], sets(3) [page 397]
string

Erlang Module

This module contains functions for string processing.

Exports

len(String) -> Length
Types:
• String = string()
• Length = integer()
Returns the number of characters in the string.

equal(String1, String2) -> bool()
Types:
• String1 = String2 = string()
Tests whether two strings are equal. Returns true if they are, otherwise false.

concat(String1, String2) -> String3
Types:
• String1 = String2 = String3 = string()
Concatenates two strings to form a new string. Returns the new string.

chr(String, Character) -> Index
rchr(String, Character) -> Index
Types:
• String = string()
• Character = char()
• Index = integer()
Returns the index of the first/last occurrence of Character in String. 0 is returned if Character does not occur.

str(String, SubString) -> Index
rstr(String, SubString) -> Index
Types:
• String = SubString = string()
- \texttt{Index = integer()}  
  Returns the position where the first/last occurrence of \texttt{SubString} begins in \texttt{String}. 0 is returned if \texttt{SubString} does not exist in \texttt{String}. For example:

\begin{verbatim}
> string:st(" Hello Hello World World ", "Hello World").
8
\end{verbatim}

\texttt{span(String, Chars) \rightarrow Length} 

\texttt{cspan(String, Chars) \rightarrow Length} 

\textbf{Types:} 
- \texttt{String = Chars = string()} 
- \texttt{Length = integer()}  

Returns the length of the maximum initial segment of \texttt{String}, which consists entirely of characters from (not from) \texttt{Chars}. For example:

\begin{verbatim}
> string:span(" \t abcdef", "\t").
5
> string:cspan(" \t abcdef", "\t").
0
\end{verbatim}

\texttt{substr(String, Start) \rightarrow SubString} 

\texttt{substr(String, Start, Length) \rightarrow Substring} 

\textbf{Types:} 
- \texttt{String = SubString = string()} 
- \texttt{Start = Length = integer()}  

Returns a substring of \texttt{String}, starting at the position \texttt{Start}, and ending at the end of the string or at length \texttt{Length}. For example:

\begin{verbatim}
> substr("Hello World", 4, 5).
"lo Wo"
\end{verbatim}

\texttt{tokens(String, SeparatorList) \rightarrow Tokens} 

\textbf{Types:} 
- \texttt{String = SeparatorList = string()} 
- \texttt{Tokens = [string()]}  

Returns a list of tokens in \texttt{String}, separated by the characters in \texttt{SeparatorList}. For example:

\begin{verbatim}
> tokens("abc defxghix jkl", ".x ").
["abc", "def", "ghi", "jkl"]
\end{verbatim}

\texttt{join(StringList, Separator) \rightarrow String} 

\textbf{Types:} 
- \texttt{StringList = [string()]} 
- \texttt{Separator = string()}
Returns a string with the elements of StringList separated by the string in Separator.

For example:

```prolog
> join(['one', 'two', 'three'], ', ').
"one, two, three"
```

**chars(Character, Number) -> String**

**chars(Character, Number, Tail) -> String**

- **Types:**
  - `Character = char()`
  - `Number = integer()`
  - `String = string()`

- Returns a string consisting of `Number` of characters `Character`. Optionally, the string can end with the string `Tail`.

**copies(String, Number) -> Copies**

- **Types:**
  - `String = Copies = string()`
  - `Number = integer()`

- Returns a string containing `String` repeated `Number` times.

**words(String) -> Count**

**words(String, Character) -> Count**

- **Types:**
  - `String = string()`
  - `Character = char()`
  - `Count = integer()`

- Returns the number of words in `String`, separated by blanks or `Character`. For example:

```prolog
> 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 `Character`s. For example:

```prolog
> string:sub_word(" Hello old boy!",3,$o).
"ld b"
```
strip(String) -> Stripped
strip(String, Direction) -> Stripped
strip(String, Direction, Character) -> Stripped

Types:
- String = Stripped = string()
- Direction = left | right | both
- Character = char()

Returns a string, where leading and/or trailing blanks or a number of Character have been removed. Direction can be left, right, or both and indicates from which direction blanks are to be removed. The function strip/1 is equivalent to strip(String, both).

For example:
> string:strip("...Hello.....", both, $).
"Hello"

left(String, Number) -> Left
left(String, Number, Character) -> Left

Types:
- String = Left = string()
- Character = char
- Number = integer()

Returns the String with the length adjusted in accordance with Number. The left margin is fixed. If the length(String) < Number, String is padded with blanks or Characters.

For example:
> string:left("Hello", 10, $).
"Hello....."

right(String, Number) -> Right
right(String, Number, Character) -> Right

Types:
- String = Right = string()
- Character = char
- Number = integer()

Returns the String with the length adjusted in accordance with Number. The right margin is fixed. If the length of (String) < Number, String is padded with blanks or Characters.

For example:
> string:right("Hello", 10, $).
".....Hello"

centre(String, Number) -> Centered
centre(String, Number, Character) -> Centered

Types:
* String = Centered = string()
* Character = char
* Number = integer()

Returns a string, where String is centred in the string and surrounded by blanks or characters. The resulting string will have the length Number.

sub_string(String, Start) -> SubString
sub_string(String, Start, Stop) -> SubString

Types:
* String = SubString = string()
* Start = Stop = integer()

Returns a substring of String, starting at the position Start to the end of the string, or to and including the Stop position.

For example:
sub_string("Hello World", 4, 8).
"lo Wo"

to_float(String) -> {Float,Rest} | {error,Reason}

Types:
* String = string()
* Float = float()
* Rest = string()
* Reason = no_float | not_a_list

Argument String is expected to start with a valid text represented float (the digits being ASCII values). Remaining characters in the string after the float are returned in Rest.

Example:

> {F1,Fs} = string:to_float("1.0-1.0e-1"),
> {F2,[]} = string:to_float(Fs),
> F1+F2.
0.9
> string:to_float("3/2=1.5").
{error,no_float}
> string:to_float("-1.5eX").
{-1.5,"eX"}

to_integer(String) -> {Int,Rest} | {error,Reason}

Types:
* String = string()
* Int = integer()
* Rest = string()
* Reason = no_integer | not_a_list
Argument String is expected to start with a valid text represented integer (the digits being ASCII values). Remaining characters in the string after the integer are returned in Rest.

Example:

```erlang
> {I1,Is} = string:to_integer("33+22"),
> {I2,[]} = string:to_integer(Is),
> I1-I2.
11
> string:to_integer("0.5").
{0,".5"}
> string:to_integer("x=2").
{error,no_integer}
```

Types:
- `String = Result = string()`
- `Char = CharResult = integer()`

The given string or character is case-converted. Note that the supported character set is ISO/IEC 8859-1 (a.k.a. Latin 1), all values outside this set is unchanged.

Notes

Some of the general string functions may seem to overlap each other. The reason for this is that this string package is the combination of two earlier packages and all the functions of both packages have been retained.

**Note:**
Any undocumented functions in string should not be used.
supervisor

Erlang Module

A behaviour module for implementing a supervisor, a process which supervises other processes called child processes. A child process can either be another supervisor or a worker process. Worker processes are normally implemented using one of the gen_event, gen_fsm, or gen_server behaviours. A supervisor implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. Supervisors are used to build an hierarchical process structure called a supervision tree, a nice way to structure a fault tolerant application.

Refer to OTP Design Principles for more information.

A supervisor assumes the definition of which child processes to supervise to be located in a callback module exporting a pre-defined set of functions.

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

Supervision Principles

The supervisor is responsible for starting, stopping and monitoring its child processes. The basic idea of a supervisor is that it should keep its child processes alive by restarting them when necessary.

The children of a supervisor is defined as a list of child specifications. When the supervisor is started, the child processes are started in order from left to right according to this list. When the supervisor terminates, it first terminates its child processes in reversed start order, from right to left.

A supervisor can have one of the following restart strategies:

- `one_for_one` - if one child process terminates and should be restarted, only that child process is affected.
- `one_for_all` - if one child process terminates and should be restarted, all other child processes are terminated and then all child processes are restarted.
- `rest_for_one` - if one child process terminates and should be restarted, the 'rest' of the child processes - i.e. the child processes after the terminated child process in the start order - are terminated. Then the terminated child process and all child processes after it are restarted.
- `simple_one_for_one` - a simplified one_for_one supervisor, where all child processes are dynamically added instances of the same process type, i.e. running the same code.

The functions `terminate_child/2`, `delete_child/2` and `restart_child/2` are invalid for `simple_one_for_one` supervisors and will return `{error, simple_one_for_one}` if the specified supervisor uses this restart strategy.
To prevent a supervisor from getting into an infinite loop of child process terminations and restarts, a maximum restart frequency is defined using two integer values $MaxR$ and $MaxT$. If more than $MaxR$ restarts occur within $MaxT$ seconds, the supervisor terminates all child processes and then itself.

This is the type definition of a child specification:

```prolog
child_spec() = {Id, StartFunc, Restart, Shutdown, Type, Modules}
```

- **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 Info an arbitrary term which is ignored by the supervisor.

The start function can also return \( \text{ignore} \) if the child process for some reason cannot be started, in which case the child specification will be kept by the supervisor but the non-existing child process will be ignored.

If something goes wrong, the function may also return an error tuple \( \{\text{error}, \text{Error}\} \).

Note that the 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 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/2`. If no name is provided, the supervisor is not registered.

Module is the name of the callback module.

Args is an arbitrary term which is passed as the argument to `Module:init/1`.

If the supervisor and its child processes are successfully created (i.e. if all child process start functions return `{ok,Child}, {ok,Child,Info}, or ignore`) the function returns `{ok,Pid}`, where Pid is the pid of the supervisor. If there already exists a process with the specified `SupName` the function returns `{error, {already_started,Pid}}`, where Pid is the pid of that process.

If `Module:init/1` returns `ignore`, this function returns `ignore` as well and the supervisor terminates with reason `normal`. If `Module:init/1` fails or returns an incorrect value, this function returns `{error,Term}` where Term is a term with information about the error, and the supervisor terminates with reason `Term`. 
If any child process start function fails or returns an error tuple or an erroneous value, the function returns \{error,shutdown\} and the supervisor terminates all started child processes and then itself with reason shutdown.

```
start_child(SupRef, ChildSpec) -> Result
Types:
  • SupRef = Name | \{Name,Node\} | \{global,Name\} | pid()
  • Name = Node = atom()
  • ChildSpec = child_spec() | [term()]
  • Result = \{ok,Child\} | \{ok,Child,Info\} | \{error,Error\}
  • Child = pid() | undefined
  • Info = term()
  • Error = already_present | \{already_started,Child\} | term()
```

Dynamically adds a child specification to the supervisor `SupRef` which starts the corresponding child process.

`SupRef` can be:

- the pid,
- `Name`, if the supervisor is locally registered,
- `\{Name,Node\}`, if the supervisor is locally registered at another node, or
- `\{global,Name\}`, if the supervisor is globally registered.

ChildSpec should be a valid child specification (unless the supervisor is a simple_one_for_one supervisor, see below). The child process will be started by using the start function as defined in the child specification.

If the case of a simple_one_for_one supervisor, the child specification defined in `Module:init/1` will be used and `ChildSpec` should instead be an arbitrary list of terms `List`. The child process will then be started by appending `List` to the existing start function arguments, i.e. by calling `apply(M, F, A++List)` where `\{M,F,A\}` is the start function defined in the child specification.

If there already exists a child specification with the specified `Id`, ChildSpec is discarded and the function returns \{error,already_present\} or \{error,\{already_started,Child\}\}, depending on if the corresponding child process is running or not.

If the child process start function returns \{ok,Child\} or \{ok,Child,Info\}, the child specification and pid is added to the supervisor and the function returns the same value.

If the child process start function returns `ignore`, the child specification is added to the supervisor, the pid is set to undefined and the function returns \{ok,undefined\}.

If the child process start function returns an error tuple or an erroneous value, or if it fails, the child specification is discarded and the function returns \{error,Error\} where Error is a term containing information about the error and child specification.

```
terminate_child(SupRef, Id) -> Result
Types:
  • SupRef = Name | \{Name,Node\} | \{global,Name\} | pid()
  • Name = Node = atom()
  • Id = term()
```
Tell the supervisor `SupRef` to terminate the child process corresponding to the child specification identified by `Id`. The process, if there is one, is terminated but the child specification is kept by the supervisor. This means that the child process may be later be restarted by the supervisor. The child process can also be restarted explicitly by calling `restart_child/2`. Use `delete_child/2` to remove the child specification.

See `start_child/2` for a description of `SupRef`.

If successful, the function returns `ok`. If there is no child specification with the specified `Id`, the function returns `{error,not_found}`.

### delete_child(SupRef, Id) -> Result

Types:
- `SupRef = Name | {Name,Node} | {global,Name} | pid()`
- `Name = Node = atom()`
- `Id = term()`
- `Result = ok | {error,Error}`
- `Error = running | not_found | simple_one_for_one`

Tell the supervisor `SupRef` to delete the child specification identified by `Id`. The corresponding child process must not be running, use `terminate_child/2` to terminate it.

See `start_child/2` for a description of `SupRef`.

If successful, the function returns `ok`. If the child specification identified by `Id` exists but the corresponding process is running, the function returns `{error,running}`. If the child specification identified by `Id` does not exist, the function returns `{error,not_found}`.

### restart_child(SupRef, Id) -> Result

Types:
- `SupRef = Name | {Name,Node} | {global,Name} | pid()`
- `Name = Node = atom()`
- `Id = term()`
- `Result = {ok,Child} | {ok,Child,Info} | {error,Error}`
- `Child = pid() | undefined`
- `Error = running | not_found | simple_one_for_one | term()`

Tell the supervisor `SupRef` to restart a child process corresponding to the child specification identified by `Id`. The child specification must exist and the corresponding child process must not be running.

See `start_child/2` for a description of `SupRef`.

If the child specification identified by `Id` does not exist, the function returns `{error,not_found}`. If the child specification exists but the corresponding process is already running, the function returns `{error,running}`.

If the child process start function returns `{ok,Child}` or `{ok,Child,Info}`, the pid is added to the supervisor and the function returns the same value.
If the child process start function returns `ignore`, the pid remains set to `undefined` and the function returns `{ok, undefined}`.

If the child process start function returns an error tuple or an erroneous value, or if it fails, the function returns `{error, Error}` where `Error` is a term containing information about the error.

\[
\text{which\_children}(\text{SupRef}) \rightarrow [{\text{Id}, \text{Child}, \text{Type}, \text{Modules}}]
\]

**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.

\[
\text{check\_childspts}([\text{ChildSpec}]) \rightarrow \text{Result}
\]

**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 215], gen_fsm(3) [page 225], gen_server(3) [page 236], sys(3) [page 454]
supervisor_bridge

Erlang Module

A behaviour module for implementing a supervisor_bridge, a process which connects a subsystem not designed according to the OTP design principles to a supervision tree. The supervisor_bridge sits between a supervisor and the subsystem. It behaves like a real supervisor to its own supervisor, but has a different interface than a real supervisor to the subsystem. Refer to OTP Design Principles for more information.

A supervisor_bridge assumes the functions for starting and stopping the subsystem to be located in a callback module exporting a pre-defined set of functions. The sys module can be used for debugging a supervisor_bridge.

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

Exports

start_link(Module, Args) -> Result
start_link(SupBridgeName, Module, Args) -> Result

Types:
- SupBridgeName = {local,Name} | {global,Name}
- Name = atom()
- Module = atom()
- Args = term()
- Result = {ok,Pid} | ignore | {error,Error}
- Pid = pid()
- Error = {already_started,Pid} | term()

Creates a supervisor_bridge process, linked to the calling process, which calls Module:init/1 to start the subsystem. To ensure a synchronized start-up procedure, this function does not return until Module:init/1 has returned.

If SupBridgeName=local,Name the supervisor_bridge is registered locally as Name using register/2. If SupBridgeName=global,Name the supervisor_bridge is registered globally as Name using global:register_name/2. If no name is provided, the supervisor_bridge is not registered. If there already exists a process with the specified SupBridgeName the function returns {error,{already_started,Pid}}, where Pid is the pid of that process.

Module is the name of the callback module.

Args is an arbitrary term which is passed as the argument to Module:init/1.
If the supervisor_bridge and the subsystem are successfully started the function returns {ok,Pid}, where Pid is 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 444], sys(3) [page 454]
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}, Modules is a list of the currently active modules in the process. This message is used by the release handler to find which processes execute a certain module. The process may at a later time be suspended and ordered to perform a code change for one of its modules.

System Events

When debugging a process with the functions of this module, the process generates system_events which are then treated in the debug function. For example, trace formats the system events to the tty. There are three predefined system events which are used when a process receives or sends a message. The process can also define its own system events. It is always up to the process itself to format these events.

Exports

`log(Name, Flag)`

`log(Name, Flag, Timeout) -> ok | {ok, [system_event()]}`  
Types:
- `Flag = true | {true, N} | false | get | print`
- `N = integer() > 0`

Turns the logging of system events ON or OFF. If ON, a maximum of `N` events are kept in the debug structure (the default is 10). If `Flag` is `get`, a list of all logged events is returned. If `Flag` is `print`, the logged events are printed to standard io. The events are formatted with a function that is defined by the process that generated the event (with a call to `sys:handle_debug/4`).

`log_to_file(Name, Flag)`

`log_to_file(Name, Flag, Timeout) -> ok | {error, open_file}`  
Types:
- `Flag = FileName | false`
- `FileName = string()`  

Enables or disables the logging of all system events in textual format to the file. The events are formatted with a function that is defined by the process that generated the event (with a call to `sys:handle_debug/4`).

`statistics(Name, Flag)`

`statistics(Name, Flag, Timeout) -> ok | {ok, Statistics}`  
Types:
- `Flag = true | false | get`
- Statistics = [{start_time, {Date1, Time1}}, {current_time, {Date, Time2}}, {reductions, integer()}, {messages_in, integer()}, {messages_out, integer()}]
- Date1 = Date2 = {Year, Month, Day}
- Time1 = Time2 = {Hour, Min, Sec}

Enables or disables the collection of statistics. If Flag is get, the statistical collection is returned.

trace(Name, Flag)
trace(Name, Flag, Timeout) -> void()

Types:
- Flag = boolean()

Prints all system events on standard io. The events are formatted with a function that is defined by the process that generated the event (with a call to sys:handle_debug/4).

no_debug(Name)
no_debug(Name, Timeout) -> void()

Turns off all debugging for the process. This includes functions that have been installed explicitly with the install function, for example triggers.

suspend(Name)
suspend(Name, Timeout) -> void()

Suspends the process. When the process is suspended, it will only respond to other system messages, but not other messages.

resume(Name)
resume(Name, Timeout) -> void()

Resumes a suspended process.

change_code(Name, Module, OldVsn, Extra)
change_code(Name, Module, OldVsn, Extra, Timeout) -> ok | {error, Reason}

Types:
- OldVsn = undefined | term()
- Module = atom()
- Extra = term()

Tells the process to change code. The process must be suspended to handle this message. The Extra argument is reserved for each process to use as its own. The function Mod:system_code_change/4 is called. OldVsn is the old version of the Module.

get_status(Name)
get_status(Name, Timeout) -> {status, Pid, {module, Mod}, [PDict, SysState, Parent, Dbg, Misc]}

Types:
- PDict = [{Key, Value}]
- SysState = running | suspended
- Parent = pid()}
- Dbg = [dbg_opt()]
- Misc = term()

Gets the status of the process.

install(Name, [Func, FuncState])
install(Name, [Func, FuncState], Timeout)

Types:
- Func = dbg_fun()
- dbg_fun() = fun(FuncState, Event, ProcState) -> done | NewFuncState
- FuncState = term()
- Event = system_event()
- ProcState = term()
- NewFuncState = term()

This function makes it possible to install other debug functions than the ones defined above. An example of such a function is a trigger, a function that waits for some special event and performs some action when the event is generated. This could, for example, be turning on low level tracing.

Func is called whenever a system event is generated. This function should return done, or a new func state. In the first case, the function is removed. It is removed if the function fails.

remove(Name, Func)
remove(Name, Func, Timeout) -> void()

Types:
- Func = dbg_fun()

Removes a previously installed debug function from the process. Func must be the same as previously installed.

Process Implementation Functions

The following functions are used when implementing a special process. This is an ordinary process which does not use a standard behaviour, but a process which understands the standard system messages.
Exports

debug_options(Options) -> [dbg_opt()]

Types:
- Options = [Opt]
- Opt = trace | log | statistics | {log_to_file, FileName} | {install, {Func, FuncState}}
- Func = dbg_fun()
- FuncState = term()

This function can be used by a process that initiates a debug structure from a list of options. The values of the Opt argument are the same as the corresponding functions.

get_debug(Item,Debug,Default) -> term()

Types:
- Item = log | statistics
- Debug = [dbg_opt()]
- Default = term()

This function gets the data associated with a debug option. Default is returned if the Item is not found. Can be used by the process to retrieve debug data for printing before it terminates.

handle_debug([dbg_opt()],FormFunc,Extra,Event) -> [dbg_opt()]

Types:
- FormFunc = dbg_fun()
- Extra = term()
- Event = system_event()

This function is called by a process when it generates a system event. FormFunc is a formatting function which is called as FormFunc(Device, Event, Extra) in order to print the events, which is necessary if tracing is activated. Extra is any extra information which the process needs in the format function, for example the name of the process.

handle_system_msg(Msg,From,Parent,Module,Debug,Misc)
This function is used by a process module that wishes to take care of system messages. The process receives a {system, From, Msg} message and passes the Msg and From to this function.

This function never returns. It calls the function Module:system_continue(Parent, NDebug, Misc) where the process continues the execution, or Module:system_terminate(Reason, Parent, Debug, Misc) if the process should terminate. The Module must export system_continue/3, system_terminate/4, and system_code_change/4 (see below).

The Misc argument can be used to save internal data in a process, for example its state. It is sent to Module:system_continue/3 or Module:system_terminate/4

\[
\text{print_log(Debug) } \rightarrow \text{ void()}
\]

Types:
- Debug = [dbg_opt()]

Prints the logged system events in the debug structure using FormFunc as defined when the event was generated by a call to handle_debug/4.

\[
\text{Mod:system_continue(Parent, Debug, Misc)}
\]

Types:
- Parent = pid()
- Debug = [dbg_opt()]
- Misc = term()

This function is called from sys:handle_system_msg/6 when the process should continue its execution (for example after it has been suspended). This function never returns.

\[
\text{Mod:system_terminate(Reason, Parent, Debug, Misc)}
\]

Types:
- Reason = term()
- Parent = pid()
- Debug = [dbg_opt()]
- Misc = term()

This function is called from sys:handle_system_msg/6 when the process should terminate. For example, this function is called when the process is suspended and its parent orders shut-down. It gives the process a chance to do a clean-up. This function never returns.

\[
\text{Mod:system_code_change(Misc, Module, OldVsn, Extra) } \rightarrow \{\text{ok, NMisc}\}
\]

Types:
- Misc = term()
- OldVsn = undefined | term()
- Module = atom()
- Extra = term()
- NMisc = term()
Called from sys:handle_system_msg/6 when the process should perform a code change. The code change is used when the internal data structure has changed. This function converts the Misc argument to the new data structure. OldVsn is the vsn attribute of the old version of the Module. If no such attribute was defined, the atom undefined is sent.
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

\texttt{start()} \rightarrow \texttt{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.

\texttt{apply\_after(Time, Module, Function, Arguments)} \rightarrow \{ok, TRef\} \| \{error, Reason\}

Types:
- \texttt{Time} = \texttt{integer()} in Milliseconds
- \texttt{Module} = \texttt{Function} = \texttt{atom()}
- \texttt{Arguments} = \{\texttt{term()}\}

Evaluates \texttt{apply(M, F, A)} after \texttt{Time} amount of time has elapsed. Returns \{ok, TRef\}, or \{error, Reason\}.

\texttt{send\_after(Time, Pid, Message)} \rightarrow \{ok, TRef\} \| \{error, Reason\}

\texttt{send\_after(Time, Message)} \rightarrow \{ok, TRef\} \| \{error, Reason\}

Types:
- \texttt{Time} = \texttt{integer()} in Milliseconds
- \texttt{Pid} = \texttt{pid()} \| \texttt{atom()}
- \texttt{Message} = \texttt{term()}
- \texttt{Result} = \{ok, TRef\} \| \{error, Reason\}

\texttt{send\_after/3} Evaluates \texttt{Pid ! Message} 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\}. 
send_after/2  Same as send_after(Time, self(), Message).

exit_after(Time, Pid, Reason1) -> {ok, TRef} | {error,Reason2}
exit_after(Time, Reason1) -> {ok, TRef} | {error,Reason2}
kill_after(Time, Pid) -> {ok, TRef} | {error,Reason2}
kill_after(Time) -> {ok, TRef} | {error,Reason2}

Types:
- Time = integer() in milliseconds
- Pid = pid() | atom()
- Reason1 = Reason2 = term()

exit_after/3  Send an exit signal with reason Reason1 to Pid Pid. Returns {ok, TRef}, or {error, Reason2}.
exit_after/2  Same as exit_after(Time, self(), Reason1).
kill_after/2  Same as exit_after(Time, Pid, kill).
kill_after/1  Same as exit_after(Time, self(), kill).

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

Types:
- Time = integer() in milliseconds
- Module = Function = atom()
- Arguments = [term()]

Evaluates apply(Module, Function, Arguments) repeatedly at intervals of Time. Returns {ok, TRef}, or {error, Reason}.

send_interval(Time, Pid, Message) -> {ok, TRef} | {error, Reason}
send_interval(Time, Message) -> {ok, TRef} | {error, Reason}

Types:
- Time = integer() in milliseconds
- Pid = pid() | atom()
- Message = term()
- Reason = term()

send_interval/3  Evaluates Pid ! Message repeatedly after Time amount of time has elapsed. (Pid can also be an atom of a registered name.) Returns {ok, TRef} or {error, Reason}.
send_interval/2  Same as send_interval(Time, self(), Message).

cancel(TRef) -> {ok, cancel} | {error, Reason}

Cancels a previously requested timeout. TRef is a unique timer reference returned by the timer function in question. Returns {ok, cancel}, or {error, Reason} when TRef is not a timer reference.
sleep(Time) -> ok
Types:
- Time = integer() in milliseconds or the atom infinity

Suspends the process calling this function for Time amount of milliseconds and then returns ok, or suspend the process forever if Time is the atom infinity. Naturally, this function does not return immediately.

tc(Module, Function, Arguments) -> {Time, Value}

Types:
- Module = Function = atom()
- Arguments = [term()]
- Time = integer() in microseconds
- Value = term()

Evaluates apply(Module, Function, Arguments) and measures the elapsed real time. Returns {Time, Value}, where Time is the elapsed real time in microseconds, and Value is what is returned from the apply.

now_diff(T2, T1) -> Tdiff

Types:
- T1 = T2 = {MegaSecs, Secs, MicroSecs}
- Tdiff = MegaSecs = Secs = MicroSecs = integer()

Calculates the time difference Tdiff = T2 - T1 in microseconds, where T1 and T2 probably are timestamp tuples returned from erlang:now/0.

seconds(Seconds) -> Milliseconds

Returns the number of milliseconds in Seconds.

minutes(Minutes) -> Milliseconds

Return the number of milliseconds in Minutes.

hours(Hours) -> Milliseconds

Returns the number of milliseconds in Hours.

hms(Hours, Minutes, Seconds) -> Milliseconds

Returns the number of milliseconds in Hours + Minutes + Seconds.
Examples

This example illustrates how to print out "Hello World!" in 5 seconds:

```
1> timer:apply_after(5000, io, format, ["~nHello World!~n", []]).
   {ok,TRef}
   Hello World!
```

The following coding example illustrates a process which performs a certain action and if this action is not completed within a certain limit, then the process is killed.

```
Pid = spawn(mod, fun, [foo, bar]),
%% If pid is not finished in 10 seconds, kill him
{ok, R} = timer:kill_after(timer:seconds(10), Pid),
...
%% We change our mind...
timer:cancel(R),
...  
```

WARNING

A timer can always be removed by calling `cancel/1`.

A n interval timer, i.e. a timer created by evaluating any of the functions `apply_after/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`.  

This module contains functions for converting between different character representations. Basically it converts between iso-latin-1 characters and Unicode ditto, but it can also convert between different Unicode encodings (like UTF-8, UTF-16 and UTF-32).

The default Unicode encoding in Erlang is in binaries UTF-8, which is also the format in which built in functions and libraries in OTP expect to find binary Unicode data. In lists, Unicode data is encoded as integers, each integer representing one character and encoded simply as the Unicode codepoint for the character.

Other Unicode encodings than integers representing codepoints or UTF-8 in binaries are referred to as "external encodings". The iso-latin-1 encoding is in binaries and lists referred to as latin1-encoding.

It is recommended to only use external encodings for communication with external entities where this is required. When working inside the Erlang/OTP environment, it is recommended to keep binaries in UTF-8 when representing Unicode characters. Latin1 encoding is supported both for backward compatibility and for communication with external entities not supporting Unicode character sets.

### DATA TYPES

- **unicode_binary()** = binary() with characters encoded in UTF-8 coding standard
- **unicode_char()** = integer() representing valid unicode codepoint

<table>
<thead>
<tr>
<th>chardata() = charlist()</th>
<th>unicode_binary()</th>
</tr>
</thead>
</table>

| charlist() = [unicode_char() | unicode_binary() | charlist()] |
|-------------|----------------|
| an unicode_binary is allowed as the tail of the list |

- **external_unicode_binary()** = binary() with characters coded in a user specified Unicode

<table>
<thead>
<tr>
<th>external_chardata() = external_charlist()</th>
<th>external_unicode_binary()</th>
</tr>
</thead>
</table>

| external_charlist() = [unicode_char() | external_unicode_binary() | external_charlist()] |
|---------------------------|--------------------------|
| an external_unicode_binary is allowed as the tail of the list |

- **latin1_binary()** = binary() with characters coded in iso-latin-1
- **latin1_char()** = integer() representing valid latin1 character (0-255)

<table>
<thead>
<tr>
<th>latin1_chardata() = latin1_charlist()</th>
<th>latin1_binary()</th>
</tr>
</thead>
</table>

| latin1_charlist() = [latin1_char() | latin1_binary() | latin1_charlist()] |
|------------------------|-----------------|
| a latin1_binary is allowed as the tail of the list |
Exports

bom_to_encoding(Bin) -> Encoding, Length

Types:
- Bin = binary() of byte size 4 or more
- Encoding = latin1 | utf8 | {utf16,little} | {utf16,big} | {utf32,little} | {utf32,big}
- Length = int()

Check for a UTF byte order mark (BOM) in the beginning of a binary. If the supplied binary Bin begins with a valid byte order mark for either UTF-8, UTF-16 or UTF-32, the function returns the encoding identified along with the length of the BOM in bytes.

If no BOM is found, the function returns {latin1, 0}

characters_to_list(Data) -> list() | error, list(), RestData | {incomplete, list(), binary()}

Types:
- Data = latin1_chardata() | chardata() | external_chardata()
- RestData = latin1_chardata() | chardata() | external_chardata()

Same as characters_to_list(Data, unicode).

characters_to_list(Data, InEncoding) -> list() | error, list(), RestData | {incomplete, list(), binary()}

Types:
- Data = latin1_chardata() | chardata() | external_chardata()
- RestData = latin1_chardata() | chardata() | external_chardata()
- InEncoding = latin1 | unicode | utf8 | utf16 | utf32 | {utf16,little} | {utf16,big} | {utf32,little} | {utf32,big}

This function converts a possibly deep list of integers and binaries into a list of integers representing unicode characters. The binaries in the input may have characters encoded as latin1 (0 - 255, one character per byte), in which case the InEncoding parameter should be given as latin1, or have characters encoded as one of the UTF-encodings, which is given as the InEncoding parameter. Only when the InEncoding is one of the UTF encodings, integers in the list are allowed to be greater than 255.

If InEncoding is latin1, the Data parameter corresponds to the iodata() type, but for unicode, the Data parameter can contain integers greater than 255 (unicode characters beyond the iso-latin-1 range), which would make it invalid as iodata().

The purpose of the function is mainly to be able to convert combinations of unicode characters into a pure unicode string in list representation for further processing. For writing the data to an external entity, the reverse function characters_to_binary/3 [page 468] comes in handy.

The option unicode is an alias for utf8, as this is the preferred encoding for Unicode characters in binaries. utf16 is an alias for {utf16,big} and utf32 is an alias for {utf32,big}. The big and little atoms denote big or little endian encoding.

If for some reason, the data cannot be converted, either because of illegal unicode/latin1 characters in the list, or because of invalid UTF encoding in any binaries, an error tuple is returned. The error tuple contains the tag error, a list representing the characters
that could be converted before the error occurred and a representation of the characters including and after the offending integer/bytes. The last part is mostly for debugging as it still constitutes a possibly deep and/or mixed list, not necessarily of the same depth as the original data. The error occurs when traversing the list and whatever’s left to decode is simply returned as is.

However, if the input Data is a pure binary, the third part of the error tuple is guaranteed to be a binary as well.

Errors occur for the following reasons:

- Integers out of range - If InEncoding is latin1, an error occurs whenever an integer greater than 255 is found in the lists. If InEncoding is of a Unicode type, error occurs whenever an integer greater than 16#10FFFF (the maximum Unicode character) or in the range 16#D800 to 16#DFFF (invalid Unicode range) is found.

- UTF encoding incorrect - If InEncoding is one of the UTF types, the bytes in any binaries have to be valid in that encoding. Errors can occur for various reasons, including “pure” decoding errors (like the upper bits of the bytes being wrong), the bytes are decoded to a too large number, the bytes are decoded to a code-point in the invalid Unicode range or encoding is “overlong”, meaning that a number should have been encoded in fewer bytes. The case of a truncated UTF is handled specially, see the paragraph about incomplete binaries below. If InEncoding is latin1, binaries are always valid as long as they contain whole bytes, as each byte falls into the valid iso-latin-1 range.

A special type of error is when no actual invalid integers or bytes are found, but a trailing binary() consists of too few bytes to decode the last character. This error might occur if bytes are read from a file in chunks or binaries in other ways are split on non UTF character boundaries. In this case an incomplete tuple is returned instead of the error tuple. It consists of the same parts as the error tuple, but the tag is incomplete instead of error and the last element is always guaranteed to be a binary consisting of the first part of a (so far) valid UTF character.

If one UTF characters is split over two consecutive binaries in the Data, the conversion succeeds. This means that a character can be decoded from a range of binaries as long as the whole range is given as input without errors occurring. Example:

```erlang
decode_data(Data) ->
    case unicode:characters_to_list(Data, unicode) of
        {incomplete, Encoded, Rest} ->
            More = get_some_more_data(),
            Encoded ++ decode_data([Rest, More]);
        {error, Encoded, Rest} ->
            handle_error(Encoded, Rest);
        List -> List
    end.
```

Bit-strings that are not whole bytes are however not allowed, so a UTF character has to be split along 8-bit boundaries to ever be decoded.

If any parameters are of the wrong type, the list structure is invalid (a number as tail) or the binaries does not contain whole bytes (bit-strings), a badarg exception is thrown.

```erlang
characters_to_binary(Data) ->
    binary() |
    {error, binary(), RestData} |
    {incomplete, binary(), binary()}
```

```
```
Types:
- `Data = latin1|chardata()|chardata()|external|chardata()`
- `RestData = latin1|chardata()|chardata()|external|chardata()`

Same as `characters_to_binary(Data, unicode, unicode)`.

\[ \text{characters_to_binary(Data, InEncoding) \to \{error, binary(), RestData\}} | \{incomplete, binary(), binary()\} \]

Types:
- `Data = latin1|chardata()|chardata()|external|chardata()`
- `RestData = latin1|chardata()|chardata()|external|chardata()`
- `InEncoding = latin1|unicode|utf8|utf16|utf32|{utf16,little}|{utf16,big}|{utf32,little}|{utf32,big}`

Same as `characters_to_binary(Data, InEncoding, unicode)`.

\[ \text{characters_to_binary(Data, InEncoding, OutEncoding) \to \{error, binary(), RestData\}} | \{incomplete, binary(), binary()\} \]

Types:
- `Data = latin1|chardata()|chardata()|external|chardata()`
- `RestData = latin1|chardata()|chardata()|external|chardata()`
- `InEncoding = latin1|unicode|utf8|utf16|utf32|{utf16,little}|{utf16,big}|{utf32,little}|{utf32,big}`
- `OutEncoding = latin1|unicode|utf8|utf16|utf32|{utf16,little}|{utf16,big}|{utf32,little}|{utf32,big}`

This function behaves as `characters_to_list/2` [page 466], but produces a binary instead of a unicode list. The `InEncoding` defines how input is to be interpreted if binaries are present in the `Data`, while `OutEncoding` defines in what format output is to be generated.

The option `unicode` is an alias for `utf8`, as this is the preferred encoding for Unicode characters in binaries. `utf16` is an alias for `{utf16,big}` and `utf32` is an alias for `{utf32,big}`. The `big` and `little` atoms denote big or little endian encoding.

Errors and exceptions occur as in `characters_to_list/2` [page 466], but the second element in the error or incomplete tuple will be a `binary()` and not a `list()`.

\[ \text{encoding_to_bom(InEncoding) \to Bin} \]

Types:
- `Bin = binary() of byte_size 4 or less`
- `InEncoding = latin1|unicode|utf8|utf16|utf32|{utf16,little}|{utf16,big}|{utf32,little}|{utf32,big}`
- `Length = int()`

Create an UTF byte order mark (BOM) as a binary from the supplied `InEncoding`. The BOM is, if supported at all, expected to be placed first in UTF encoded files or messages. The function returns `<<>>>` for the latin1 encoding, there is no BOM for ISO-latin-1.

It can be noted that the BOM for UTF-8 is seldom used, and it is really not a byte order mark. There are obviously no byte order issues with UTF-8, so the BOM is only there to differentiate UTF-8 encoding from other UTF formats.
**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>hkcuc</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>hkcuc</td>
<td>HKEY_CURRENT_CONFIG</td>
</tr>
<tr>
<td>dyn_data</td>
<td>HKEY_DYN_DATA</td>
</tr>
<tr>
<td>hkd</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

\texttt{change}\_key(\texttt{RegHandle}, \texttt{Key}) \rightarrow \texttt{ReturnValue}

Types:
- \texttt{RegHandle} = \texttt{term()}
- \texttt{Key} = \texttt{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 \\.

\texttt{change}\_key\_create(\texttt{RegHandle}, \texttt{Key}) \rightarrow \texttt{ReturnValue}

Types:
- \texttt{RegHandle} = \texttt{term()}
- \texttt{Key} = \texttt{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.

\texttt{close(RegHandle)} \rightarrow \texttt{ReturnValue}

Types:
- \texttt{RegHandle} = \texttt{term()}

Closes the registry. After that, the \texttt{RegHandle} cannot be used.

\texttt{current}\_key(\texttt{RegHandle}) \rightarrow \texttt{ReturnValue}

Types:
- \texttt{RegHandle} = \texttt{term()}
- \texttt{ReturnValue} = \{ok, \texttt{string()}\}

Returns the path to the current key. This is the equivalent of \texttt{pwd}.

Note that the current key is stored in the driver, and might be invalid (e.g. if the key has been removed).

\texttt{delete}\_key(\texttt{RegHandle}) \rightarrow \texttt{ReturnValue}

Types:
- \texttt{RegHandle} = \texttt{term()}
- \texttt{ReturnValue} = \{ok\} | \{error, \texttt{ErrorId}\}
Deletes the current key, if it is valid. Calls the Win32 API function RegDeleteKey(). Note that this call does not change the current key, (unlike change_key_create/2.) This means that after the call, the current key is invalid.

```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.

subkeys(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} | {error, ErrorId}
- ValuePairs = [ValuePair]
- ValuePair = {Name, Value}
- Name = string | default
- Value = string() | integer() | binary()

Retrieves a list of all values on the current key. The values have types corresponding to the registry types, see value. Calls the Win32 API function EnumRegValuesEx().

SEE ALSO

Win32 Programmer’s Reference (from Microsoft)
erl_posix.msg
The Windows 95 Registry (book from O'Reilly)
zip

Erlang Module

The zip module archives and extract files to and from a zip archive. The zip format is specified by the “ZIP Appnote.txt” file available on PKWare’s website www.pkware.com. The zip module supports zip archive versions up to 6.1. However, password-protection and Zip64 is not supported.

By convention, the name of a zip file should end in “.zip”. To abide to the convention, you’ll need to add “.zip” yourself to the name.

Zip archives are created with the zip/2 [page 474] or the zip/3 [page 474] 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 476] or the unzip/2 [page 476] function. (They are also available as extract.)

To return a list of the files in a zip archive, use the list_dir/1 [page 477] or the list_dir/2 [page 477] function. (They are also available as table.)

To print a list of files to the Erlang shell, use either the t/1 [page 477] or tt/1 [page 477] 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 477], zip_get [page 478], zip_list_dir [page 478] and zip_close [page 478] 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

```
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} | {compress, What} | {uncompress, What} | {silent, What} | {what, What} | {add, [Extension]} | {del, [Extension]}
- **Extension** = string()
- **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.

It is possible to override the default behavior and explicitly control what types of files that should be compressed by using the \{compress, What\} and \{uncompress, What\} options. It is possible to have several compress and uncompress options. In order to trigger compression of a file, its extension must match with the compress condition and must not match the uncompress condition. For example if compress is set to \["gif", "jpg"\] and uncompress is set to \["jpg"\], only files with "gif" as extension will be compressed. No other files will be compressed.

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.)

- \{compress, What\} Controls what types of files that will be compressed. It is by default set to all. The following values of What are allowed:
  - all means that all files will be compressed (as long as they pass the uncompress condition).
  - \[Extension\] means that only files with exactly these extensions will be compressed.
  - \{add, [Extension]\} adds these extensions to the list of compress extensions.
  - \{del, [Extension]\} deletes these extensions from the list of compress extensions.

- \{uncompress, What\} Controls what types of files that will be uncompressed. It is by default set to \[".Z",".zip",".zoo",".arc",".lzh",".arj"\]. The following values of What are allowed:
  - all means that no files will be compressed.
  - \[Extension\] means that files with these extensions will be uncompressed.
  - \{add, [Extension]\} adds these extensions to the list of uncompress extensions.
  - \{del, [Extension]\} deletes these extensions from the list of uncompress extensions.

```erlang
unzip(Archive) -> RetValue
unzip(Archive, Options) -> RetValue
extract(Archive) -> RetValue
extract(Archive, Options) -> RetValue
```

**Types:**
- Archive = filename() | binary()
- Options = [Option]
The unzip/1 function extracts all files from a zip archive. The unzip/2 function provides options to extract some files, and more. If the Archive argument is given as a binary, the contents of the binary is assumed to be a zip archive, otherwise it should be a filename.

The following options are available:

- **file list, FileList** By default, all files will be extracted from the zip archive. With the **file list, FileList** option, the unzip/2 function will only extract the files whose names are included in FileList. The full paths, including the names of all sub directories within the zip archive, must be specified.
- **cooked** By default, the open/2 function will open the zip file in raw mode, which is faster but does not allow a remote (erlang) file server to be used. Adding cooked to the mode list will override the default and open zip file without the raw option. The same goes for the files extracted.
- **keep old files** By default, all existing files with the same name as file in the zip archive will be overwritten. With the keep old files option, the unzip/2 function will not overwrite any existing files. Note that even with the memory option given, which means that no files will be overwritten, files existing will be excluded from the result.
- **verbose** Print an informational message as each file is being extracted.
- **memory** Instead of extracting to the current directory, the memory option will give the result as a list of tuples {Filename, Binary}, where Binary is a binary containing the extracted data of the file named Filename in the zip archive.
- **cwd, CWD** Use the given directory as current directory, it will be prepended to file names when extracting them from the zip-archive. (Acting like a file:favorite/2, but without changing the global cwd property.)

```
list_dir(Archive) -> RetValue
list_dir(Archive, Options)
table(Archive) -> RetValue
table(Archive, Options)
```

Types:

- **Archive** = filename() | binary()
- **RetValue** = {ok, [Comment, Files]} | {error, Reason}
- **Comment** = zip_comment()
- **Files** = [zip_file()]
- **Options** = [Option]
The `list_dir/1` function retrieves the names of all files in the zip archive `Archive`. The `list_dir/2` function provides options.

As synonyms, the functions `table/2` and `table/3` are provided, to make it resemble the `erl_tar` module.

The result value is the tuple `{ok, List}`, where `List` contains the zip archive comment as the first element.

The following options are available:

- `cooked`. By default, the `open/2` function will open the zip file in `raw` mode, which is faster but does not allow a remote (erlang) file server to be used. Adding `cooked` to the mode list will override the default and open zip file without the `raw` option.

### t(Archive)

Types:
- `Archive = filename() | binary() | ZipHandle`
- `ZipHandle = pid()`

The `t/1` function prints the names of all files in the zip archive `Archive` to the Erlang shell. (Similar to "tart".)

### tt(Archive)

Types:
- `Archive = filename() | binary()`

The `tt/1` function prints names and information about all files in the zip archive `Archive` to the Erlang shell. (Similar to "tar tv").

### zip_open(Archive) -> {ok, ZipHandle} | {error, Reason}

### zip_open(Archive, Options) -> {ok, ZipHandle} | {error, Reason}

Types:
- `Archive = filename() | binary()`
- `Options = [Option]`
- `Options = cooked | memory | {cwd, CWD}`
- `CWD = string()`
- `ZipHandle = pid()`

The `zip_open` function opens a zip archive, and reads and saves its directory. This means that subsequently reading files from the archive will be faster than unzipping files one at a time with `unzip`.

The archive must be closed with `zip_close/1`.

### zip_list_dir(ZipHandle) -> Result | {error, Reason}

Types:
- `Result = [ZipComment, ZipFile...]`
- `ZipComment = #zip_comment{}`
• ZipFile = #zip_file{}
• ZipHandle = pid()

The `zip_list_dir/1` function returns the file list of an open zip archive.

```
zip_get(ZipHandle) -> {ok, [Result]} | {error, Reason}
zip_get(FileName, ZipHandle) -> {ok, Result} | {error, Reason}
```

**Types:**
- FileName = filename()
- ZipHandle = pid()
- Result = filename() | {filename(), binary()}

The `zip_get` function extracts one or all files from an open archive. The files will be unzipped to memory or to file, depending on the options given to the `zip_open` function when the archive was opened.

```
zip_close(ZipHandle) -> ok | {error, einval}
```

**Types:**
- ZipHandle = pid()

The `zip_close/1` function closes a zip archive, previously opened with `zip_open`. All resources are closed, and the handle should not be used after closing.
List of Figures

1.1 Cyrillic characters in an Erlang shell ................................................. 18
1.2 Unicode characters in allowed and disallowed context ............................ 18
1.3 io_lib:format with Unicode translation ............................................... 20
## Index of Modules and Functions

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Line Number</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a_function/2</code></td>
<td><code>sofs</code></td>
<td>419</td>
</tr>
<tr>
<td><code>abcast/2</code></td>
<td><code>gen_server</code></td>
<td>240</td>
</tr>
<tr>
<td><code>abcast/3</code></td>
<td><code>gen_server</code></td>
<td>240</td>
</tr>
<tr>
<td><code>absname/1</code></td>
<td><code>filename</code></td>
<td>198</td>
</tr>
<tr>
<td><code>absname/2</code></td>
<td><code>filename</code></td>
<td>199</td>
</tr>
<tr>
<td><code>absname_join/2</code></td>
<td><code>filename</code></td>
<td>199</td>
</tr>
<tr>
<td><code>abstract/1</code></td>
<td><code>erl_parse</code></td>
<td>149</td>
</tr>
<tr>
<td><code>acos/1</code></td>
<td><code>math</code></td>
<td>283</td>
</tr>
<tr>
<td><code>acosh/1</code></td>
<td><code>math</code></td>
<td>283</td>
</tr>
<tr>
<td><code>add/2</code></td>
<td><code>gb_sets</code></td>
<td>205</td>
</tr>
<tr>
<td><code>add/3</code></td>
<td><code>erl_tar</code></td>
<td>161</td>
</tr>
<tr>
<td><code>add/4</code></td>
<td><code>erl_tar</code></td>
<td>161</td>
</tr>
<tr>
<td><code>add_binding/3</code></td>
<td><code>erl_eval</code></td>
<td>140</td>
</tr>
<tr>
<td><code>add_edge/3</code></td>
<td><code>digraph</code></td>
<td>125</td>
</tr>
<tr>
<td><code>add_edge/4</code></td>
<td><code>digraph</code></td>
<td>125</td>
</tr>
<tr>
<td><code>add_edge/5</code></td>
<td><code>digraph</code></td>
<td>125</td>
</tr>
<tr>
<td><code>add_element/2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>gb_sets</code></td>
<td>205</td>
</tr>
<tr>
<td></td>
<td><code>ordsets</code></td>
<td>302</td>
</tr>
<tr>
<td></td>
<td><code>sets</code></td>
<td>398</td>
</tr>
<tr>
<td></td>
<td><code>add_handler/3</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>gen_event</code></td>
<td>217</td>
</tr>
<tr>
<td></td>
<td><code>add_sup_handler/3</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>gen_event</code></td>
<td>217</td>
</tr>
<tr>
<td></td>
<td><code>add_vertex/1</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>digraph</code></td>
<td>126</td>
</tr>
<tr>
<td></td>
<td><code>add_vertex/2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>digraph</code></td>
<td>126</td>
</tr>
<tr>
<td></td>
<td><code>add_vertex/3</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>digraph</code></td>
<td>126</td>
</tr>
<tr>
<td></td>
<td><code>all/0</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>dets</code></td>
<td>105</td>
</tr>
<tr>
<td></td>
<td><code>ets</code></td>
<td>167</td>
</tr>
<tr>
<td></td>
<td><code>all/2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>lists</code></td>
<td>266</td>
</tr>
<tr>
<td></td>
<td><code>any/2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>lists</code></td>
<td>266</td>
</tr>
<tr>
<td></td>
<td><code>append/1</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>lists</code></td>
<td>266</td>
</tr>
<tr>
<td></td>
<td><code>qlc</code></td>
<td>326</td>
</tr>
<tr>
<td></td>
<td><code>append/2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>lists</code></td>
<td>267</td>
</tr>
<tr>
<td></td>
<td><code>qlc</code></td>
<td>326</td>
</tr>
<tr>
<td></td>
<td><code>append/3</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>dict</code></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td><code>orddict</code></td>
<td>296</td>
</tr>
<tr>
<td></td>
<td><code>append_list/3</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>dict</code></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td><code>orddict</code></td>
<td>296</td>
</tr>
<tr>
<td></td>
<td><code>append_values/2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>proplists</code></td>
<td>313</td>
</tr>
</tbody>
</table>
Index of Modules and Functions

apply_after/4
  timer, 461
apply_interval/4
  timer, 462
arborescence_root/1
digraph_utils, 133
arith_op/2
erl_internal, 144
array
  default/1, 81
  fix/1, 81
  foldl/2, 81
  foldr/2, 81
  from_list/1, 81
  from_orddict/1, 82
  get/1, 82
  is_array/1, 82
  is_fix/1, 82
  map/2, 82
  new/0, 82
  new/1, 82, 83
  relax/1, 83
  reset/1, 83
  resize/1, 84
  set/1, 84
  size/1, 84
  sparse_foldl/2, 84
  sparse_foldr/2, 84
  sparse_map/2, 85
  sparse_size/1, 85
  sparse_to_list/1, 85
  sparse_to_orddict/1, 85
  to_list/1, 85
  to_orddict/1, 85
asin/1
  math, 283
asinh/1
  math, 283
atan/1
  math, 283
atan2/2
  math, 283
atanh/1
  math, 283
attach/1
  pool, 306
attribute/1
erl_pp, 151
attribute/2
erl_pp, 151
attributes_info/1
erl_scan, 157
attributes_info/2
erl_scan, 157
balance/1
gb_sets, 205
gb_trees, 210
base64
  decode/1, 86
  decode_to_string/1, 86
  encode/1, 86
  encode_to_string/1, 86
  mime_decode/1, 86
  mime_decode_to_string/1, 86
basename/1
  filename, 199
basename/2
  filename, 199
bchunk/2
dets, 105
beam_lib
  chunks/2, 90
  chunks/3, 90
  clear_crypto_key_fun/0, 94
  cmp/2, 92
  cmp_dirs/2, 92
  crypto_key_fun/1, 93
  diff_dirs/2, 92
  format_error/1, 93
  info/1, 91
  md5/1, 91
  strip/1, 92
  strip_files/1, 92
  strip_release/1, 93
  version/1, 90
bif/2
erl_internal, 144
binding/2
erl_eval, 140
bindings/1
erl_eval, 140
bom_to_encoding/1
  unicode, 466
bool_op/2
<table>
<thead>
<tr>
<th>Module/Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>erl_internal, 144</td>
<td></td>
</tr>
<tr>
<td>bt/1</td>
<td>95</td>
</tr>
<tr>
<td>c, 95</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>bt/1, 95</td>
<td></td>
</tr>
<tr>
<td>c/1, 95</td>
<td></td>
</tr>
<tr>
<td>c/2, 95</td>
<td></td>
</tr>
<tr>
<td>cd/1, 95</td>
<td></td>
</tr>
<tr>
<td>flush/0, 96</td>
<td></td>
</tr>
<tr>
<td>help/0, 96</td>
<td></td>
</tr>
<tr>
<td>i/0, 96</td>
<td></td>
</tr>
<tr>
<td>i/3, 96</td>
<td></td>
</tr>
<tr>
<td>l/1, 96</td>
<td></td>
</tr>
<tr>
<td>lc/1, 96</td>
<td></td>
</tr>
<tr>
<td>ls/0, 96</td>
<td></td>
</tr>
<tr>
<td>ls/1, 96</td>
<td></td>
</tr>
<tr>
<td>m/0, 96</td>
<td></td>
</tr>
<tr>
<td>m/1, 97</td>
<td></td>
</tr>
<tr>
<td>memory/0, 97</td>
<td></td>
</tr>
<tr>
<td>memory/1, 97</td>
<td></td>
</tr>
<tr>
<td>nc/1, 97</td>
<td></td>
</tr>
<tr>
<td>nc/2, 97</td>
<td></td>
</tr>
<tr>
<td>ni/0, 96</td>
<td></td>
</tr>
<tr>
<td>nl/1, 97</td>
<td></td>
</tr>
<tr>
<td>nregs/0, 98</td>
<td></td>
</tr>
<tr>
<td>pid/3, 97</td>
<td></td>
</tr>
<tr>
<td>pwd/0, 97</td>
<td></td>
</tr>
<tr>
<td>q/0, 98</td>
<td></td>
</tr>
<tr>
<td>regs/0, 98</td>
<td></td>
</tr>
<tr>
<td>xm/1, 98</td>
<td></td>
</tr>
<tr>
<td>y/1, 98</td>
<td></td>
</tr>
<tr>
<td>y/2, 98</td>
<td></td>
</tr>
<tr>
<td>c/1, 95</td>
<td></td>
</tr>
<tr>
<td>calendar</td>
<td></td>
</tr>
<tr>
<td>date_to_gregorian_days/1, 100</td>
<td></td>
</tr>
<tr>
<td>date_to_gregorian_days/3, 100</td>
<td></td>
</tr>
<tr>
<td>datetime_to_gregorian_seconds/2, 100</td>
<td></td>
</tr>
<tr>
<td>day_of_the_week/1, 100</td>
<td></td>
</tr>
<tr>
<td>day_of_the_week/3, 100</td>
<td></td>
</tr>
<tr>
<td>gregorian_days_to_date/1, 100</td>
<td></td>
</tr>
<tr>
<td>gregorian_seconds_to_datetime/1, 100</td>
<td></td>
</tr>
<tr>
<td>is_leap_year/1, 100</td>
<td></td>
</tr>
<tr>
<td>last_day_of_the_month/2, 100</td>
<td></td>
</tr>
<tr>
<td>local_time/0, 101</td>
<td></td>
</tr>
<tr>
<td>local_time_to_universal_time/2, 101</td>
<td></td>
</tr>
<tr>
<td>local_time_to_universal_time_dst/2, 101</td>
<td></td>
</tr>
<tr>
<td>now_to_datetime/1, 102</td>
<td></td>
</tr>
<tr>
<td>now_to_local_time/1, 101</td>
<td></td>
</tr>
<tr>
<td>now_to_universal_time/1, 102</td>
<td></td>
</tr>
<tr>
<td>seconds_to_daytime/1, 102</td>
<td></td>
</tr>
<tr>
<td>seconds_to_time/1, 102</td>
<td></td>
</tr>
<tr>
<td>time_difference/2, 102</td>
<td></td>
</tr>
<tr>
<td>time_to_seconds/1, 102</td>
<td></td>
</tr>
<tr>
<td>universal_time/0, 102</td>
<td></td>
</tr>
<tr>
<td>universal_time_to_local_time/2, 103</td>
<td></td>
</tr>
<tr>
<td>valid_date/1, 103</td>
<td></td>
</tr>
<tr>
<td>valid_date/3, 103</td>
<td></td>
</tr>
<tr>
<td>call/2</td>
<td></td>
</tr>
<tr>
<td>gen_server, 238</td>
<td></td>
</tr>
<tr>
<td>call/3</td>
<td></td>
</tr>
<tr>
<td>gen_event, 218</td>
<td></td>
</tr>
<tr>
<td>gen_server, 238</td>
<td></td>
</tr>
<tr>
<td>call/4</td>
<td></td>
</tr>
<tr>
<td>gen_event, 218</td>
<td></td>
</tr>
<tr>
<td>cancel/1</td>
<td></td>
</tr>
<tr>
<td>timer, 462</td>
<td></td>
</tr>
<tr>
<td>cancel_timer/1</td>
<td></td>
</tr>
<tr>
<td>gen_fsm, 230</td>
<td></td>
</tr>
<tr>
<td>canonical_relation/1</td>
<td></td>
</tr>
<tr>
<td>sofs, 419</td>
<td></td>
</tr>
<tr>
<td>cast/2</td>
<td></td>
</tr>
<tr>
<td>gen_server, 240</td>
<td></td>
</tr>
<tr>
<td>catch_exception/1</td>
<td></td>
</tr>
<tr>
<td>shell, 409</td>
<td></td>
</tr>
<tr>
<td>cd/1</td>
<td></td>
</tr>
<tr>
<td>c, 95</td>
<td></td>
</tr>
<tr>
<td>centre/2</td>
<td></td>
</tr>
<tr>
<td>string, 441</td>
<td></td>
</tr>
<tr>
<td>centre/3</td>
<td></td>
</tr>
<tr>
<td>string, 441</td>
<td></td>
</tr>
<tr>
<td>change_code/4</td>
<td></td>
</tr>
<tr>
<td>sys, 456</td>
<td></td>
</tr>
<tr>
<td>change_code/5</td>
<td></td>
</tr>
<tr>
<td>sys, 456</td>
<td></td>
</tr>
<tr>
<td>change_key/2</td>
<td></td>
</tr>
<tr>
<td>win32reg, 470</td>
<td></td>
</tr>
<tr>
<td>change_key_create/2</td>
<td></td>
</tr>
<tr>
<td>win32reg, 470</td>
<td></td>
</tr>
<tr>
<td>char_list/1</td>
<td></td>
</tr>
<tr>
<td>io_lib, 263</td>
<td></td>
</tr>
<tr>
<td>characters_to_binary/1</td>
<td></td>
</tr>
</tbody>
</table>
Index of Modules and Functions

unicode, 467
characters_to_binary/2
  unicode, 468
characters_to_binary/3
  unicode, 468
characters_to_list/1
  unicode, 466
characters_to_list/2
  unicode, 466
chars/2
  string, 440
chars/3
  string, 440
check/1
  file_sorter, 194
check/2
  file_sorter, 194
check_childspecs/1
  supervisor, 449
chr/2
  string, 438
chunks/2
  beam_lib, 90
chunks/3
  beam_lib, 90
clear_crypto_key_fun/0
  beam_lib, 94
close/1
  dets, 106
epp, 137
erl_tar, 161
win32reg, 470
cmp/2
  beam_lib, 92
cmp_dirs/2
  beam_lib, 92
columns/1
  io, 247
comp_op/2
  erl_internal, 145
compact/1
  proplists, 313
compile/1
  re, 346
  compile/2
  re, 346
  components/1
digraph_utils, 133
  composite/2
  sofs, 419
  concat/1
  lists, 267
  concat/2
  string, 438
  condensation/1
digraph_utils, 133
  cons/2
  queue, 341
  constant_function/2
  sofs, 419
  converse/1
  sofs, 420
  copies/2
  string, 440
cos/1
  math, 283
cosh/1
  math, 283
create/1
  pg, 304
create/2
  erl_tar, 161
  pg, 304
  zip, 474
create/3
  erl_tar, 162
  zip, 474
crypto_key_fun/1
  beam_lib, 93
cspan/2
  string, 439
current_key/1
  win32reg, 470
cursor/2
  qlc, 326
cyclic_strong_components/1
digraph_utils, 133
dae/1
  queue, 342
date_to_gregorian_days/1
calendar, 100
date_to_gregorian_days/3
calendar, 100
datetime_to_gregorian_seconds/2
calendar, 100
day_of_the_week/1
calendar, 100
day_of_the_week/3
calendar, 100
debug_options/1
  sys, 458
decode/1
    base64, 86
decode_to_string/1
    base64, 86
deep_char_list/1
  io_lib, 263
default/1
  array, 81
del_binding/2
  erLevel, 140
del_edge/2
  digraph, 126
del_edges/2
  digraph, 126
del_element/2
  gb_sets, 205
orsets, 302
  sets, 398
del_path/3
  digraph, 126
del_vertex/2
  digraph, 127
del_vertices/2
  digraph, 127
delete/1
  digraph, 127
  ets, 167
delete/2
  ets, 167
dets, 106
  ets, 167
  gb_sets, 205
gb_trees, 210
lists, 267
proplists, 313
delete_all_objects/1
  dets, 106
  ets, 167
delete_any/2
  gb_sets, 205
gb_trees, 211
delete_child/2
  supervisor, 448
delete_cursor/1
  qlc, 327
delete_handler/3
  gen_event, 219
delete_key/1
  win32reg, 470
delete_object/2
  dets, 106
  ets, 167
delete_value/2
  win32reg, 471
dets
  all/0, 105
  bchunk/2, 105
  close/1, 106
  delete/2, 106
  delete_all_objects/1, 106
  delete_object/2, 106
  first/1, 106
  fold1/3, 107
  foldr/3, 107
  from_ets/2, 107
  info/1, 107
  info/2, 108
  init_table/3, 108
  insert/2, 109
  insert_new/2, 109
  is_compatible_bchunk_format/2, 110
  is_dets_file/1, 110
  lookup/2, 110
  match/1, 110
  match/2, 111
  match/3, 111
  match_delete/2, 111
  match_object/1, 112
match_object/2, 112
match_object/3, 112
member/2, 113
next/2, 113
open_file/1, 113
open_file/2, 113
pid2name/1, 115
repair_continuation/2, 115
safe_fixtable/2, 115
select/1, 116
select/2, 116
select/3, 116
select_delete/2, 117
slot/2, 117
sync/1, 117
table/2, 118
to_ets/2, 118
traverse/2, 119
update_counter/3, 119
dict
append/3, 120
append_list/3, 120
erase/2, 120
fetch/2, 121
fetch_keys/1, 121
filter/2, 121
find/2, 121
fold/3, 121
from_list/1, 121
is_key/2, 122
map/2, 122
merge/3, 122
new/0, 122
size/1, 122
store/3, 122
to_list/1, 123
update/3, 123
update/4, 123
update_counter/3, 123
diff_dirs/2
beam_lib, 92
difference/2
gb_sets, 205
sofs, 420
digraph
add_edge/3, 125
add_edge/4, 125
add_edge/5, 125
add_vertex/1, 126
add_vertex/2, 126
add_vertex/3, 126
del_edge/2, 126
del_edges/2, 126
del_path/3, 126
del_vertex/2, 127
del_vertices/2, 127
delete/1, 127
del_edges/2, 127
del_edges/2, 127
get_cycle/2, 128
get_path/3, 128
get_short_cycle/2, 128
get_short_path/3, 128
in_degree/2, 129
in_edges/2, 129
in_neighbours/2, 129
info/1, 129
new/0, 130
new/1, 130
no_edges/1, 130
no_vertices/1, 130
out_degree/2, 130
out_edges/2, 130
out_neighbours/2, 130
vertex/2, 131
vertices/1, 131
digraph_to_family/2
sofs, 420
digraph_utils
arborescence_root/1, 133
components/1, 133
condensation/1, 133
cyclic_strong_components/1, 133
is_acyclic/1, 133
is_arborescence/1, 134
is_tree/1, 134
loop_vertices/1, 134
postorder/1, 134
preorder/1, 134
reachable/2, 134
reachable_neighbours/2, 135
reaching/2, 135
reaching_neighbours/2, 135
strong_components/1, 135
subgraph/3, 135
topsort/1, 136
dirname/1
filename, 200
domain/1
sofs, 420
Index of Modules and Functions

drestriction/2
  sofs, 420

drestriction/3
  sofs, 421

drop/1
  queue, 340

drop_r/1
  queue, 340

dropwhile/2
  lists, 267

duplicate/2
  lists, 267

e/2
  qlc, 327

edge/2
  digraph, 127

edges/1
  digraph, 127

edges/2
  digraph, 127

empty/0
  gb_sets, 206
  gb_trees, 211

empty_set/0
  sofs, 421

encode/1
  base64, 86

encode_to_string/1
  base64, 86

encoding_to_bom/1
  unicode, 468

ensure_dir/1
  filelib, 195

enter/3
  gb_trees, 211

enter_loop/3
  gen_server, 241

enter_loop/4
  gen_fsm, 230
  gen_server, 241

enter_loop/5
  gen_fsm, 230
  gen_server, 241
  gen_fsm, 230
  gen_server, 241

enter_loop/6
  gen_fsm, 230

epp
  close/1, 137
  open/2, 137
  open/3, 137

parse_erl_form/1, 137

parse_file/3

equal/2
  string, 438

erase/2
  dict, 120
  orddict, 296

erf/1
  math, 284

erfc/1
  math, 284

erl_eval
  add_binding/3, 140
  binding/2, 140
  bindings/1, 140
  del_binding/2, 140
  expr/2, 139
  expr/3, 139
  expr/4, 139
  expr_list/2, 140
  expr_list/3, 140
  expr_list/4, 140

exprs/2
  exprs/3
  exprs/4

exprs/2
  exprs/3
  exprs/4

exprs/2
  exprs/3
  exprs/4

new-bindings/0, 140

erl_expand_records
  module/2, 142

erl_id_trans
  parse_transform/2, 143

erl_internal
  arith_op/2, 144
  bif/2, 144
  bool_op/2, 144
  comp_op/2, 145
  guard_bif/2, 144
  list_op/2, 145
  op_type/2, 145
  send_op/2, 145
  type_test/2, 144

erl_lint
  format_error/1, 147
Index of Modules and Functions

is_guard_test/1, 147
module/1, 146
t/1, 164
table/1, 164
table/2, 164
module/2, 146
tt/1, 165
module/3, 146

erl_parse
abstract/1, 149
format_error/1, 149
normalise/1, 149
closed/1, 148
parse_form/1, 148
tokens/1, 149
tokens/2, 149

erl_pp
attribute/1, 151
attribute/2, 151
expr/1, 152
expr/2, 152
expr/3, 152
expr/4, 152
exprs/1, 152
exprs/2, 152
exprs/3, 152
form/1, 151
form/2, 151
function/1, 151
function/2, 151
guard/1, 151
guard/2, 151

erl_scan
attributes_info/1, 157
attributes_info/2, 157
format_error/1, 158
reserved_word/1, 158
set_attribute/3, 158
string/1, 154
string/2, 154
string/3, 154
token_info/1, 156
token_info/2, 156
tokens/3, 155
tokens/4, 155

erl_tar
add/3, 161
add/4, 161
close/1, 161
create/2, 161
create/3, 162
extract/1, 162
extract/2, 162
format_error/1, 163
error_message/2
lib, 264

ets
all/0, 167
delete/1, 167
delete/2, 167
delete_all_objects/1, 167
delete_object/2, 167
file2tab/1, 168
file2tab/2, 168
first/1, 168
foldl/3, 169
foldr/3, 169
from_dets/2, 169
fun2ms/1, 169
i/0, 171
i/1, 171
info/1, 171
info/2, 171
init_table/2, 172
insert/2, 172
insert_new/2, 173
is_compiled_ms/1, 173
last/1, 173
lookup/2, 174
lookup_element/3, 174
match/1, 175
match/2, 174
match/3, 175
match_delete/2, 175
match_object/1, 176
match_object/2, 176
match_object/3, 176
match_spec_compile/1, 176
match_spec_run/2, 177
member/2, 177
new/2, 178
next/2, 178
prev/2, 179
rename/2, 179
repair_continuation/2, 179
safe_fixtable/2, 180
select/1, 183
select/2, 181

ets
add/3, 161
add/4, 161
close/1, 161
create/2, 161
create/3, 162
extract/1, 162
extract/2, 162
format_error/1, 163
open/2, 163

ETS
all/0, 167
delete/1, 167
delete/2, 167
delete_all_objects/1, 167
delete_object/2, 167
file2tab/1, 168
file2tab/2, 168
first/1, 168
foldl/3, 169
foldr/3, 169
from_dets/2, 169
fun2ms/1, 169
i/0, 171
i/1, 171
info/1, 171
info/2, 171
init_table/2, 172
insert/2, 172
insert_new/2, 173
is_compiled_ms/1, 173
last/1, 173
lookup/2, 174
lookup_element/3, 174
match/1, 175
match/2, 174
match/3, 175
match_delete/2, 175
match_object/1, 176
match_object/2, 176
match_object/3, 176
match_spec_compile/1, 176
match_spec_run/2, 177
member/2, 177
new/2, 178
next/2, 178
prev/2, 179
rename/2, 179
repair_continuation/2, 179
safe_fixtable/2, 180
select/1, 183
select/2, 181
<table>
<thead>
<tr>
<th>Function</th>
<th>Module/Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>select/3</td>
<td>182</td>
</tr>
<tr>
<td>select_count/2</td>
<td>183</td>
</tr>
<tr>
<td>select_delete/2</td>
<td>183</td>
</tr>
<tr>
<td>slot/2</td>
<td>184</td>
</tr>
<tr>
<td>tab2file/2</td>
<td>184</td>
</tr>
<tr>
<td>tab2file/3</td>
<td>184</td>
</tr>
<tr>
<td>tab2list/1,185</td>
<td></td>
</tr>
<tr>
<td>tabfile_info/1,185</td>
<td></td>
</tr>
<tr>
<td>table/2</td>
<td>186</td>
</tr>
<tr>
<td>test_ms/2</td>
<td>187</td>
</tr>
<tr>
<td>to_dets/2</td>
<td>187</td>
</tr>
<tr>
<td>update_counter/3</td>
<td>188</td>
</tr>
<tr>
<td>update_element/4</td>
<td>188</td>
</tr>
<tr>
<td>eval/2</td>
<td>qlc , 327</td>
</tr>
<tr>
<td>exit_after/2</td>
<td>timer , 462</td>
</tr>
<tr>
<td>exit_after/3</td>
<td>timer , 462</td>
</tr>
<tr>
<td>exp/1</td>
<td>math , 283</td>
</tr>
<tr>
<td>expand/1</td>
<td>win32reg , 471</td>
</tr>
<tr>
<td>expand/2</td>
<td>proplists , 313</td>
</tr>
<tr>
<td>expr/1</td>
<td>erl_pp , 152</td>
</tr>
<tr>
<td>expr/2</td>
<td>erl_eval , 139</td>
</tr>
<tr>
<td>expr/3</td>
<td>erl_pp , 152</td>
</tr>
<tr>
<td>expr/4</td>
<td>erl_eval , 139</td>
</tr>
<tr>
<td>expr_list/2</td>
<td>erl_eval , 140</td>
</tr>
<tr>
<td>expr_list/3</td>
<td>erl_eval , 140</td>
</tr>
<tr>
<td>expr_list/4</td>
<td>erl_eval , 140</td>
</tr>
<tr>
<td>exprs/1</td>
<td>erl_pp , 152</td>
</tr>
<tr>
<td>exprs/2</td>
<td>erl_eval , 139</td>
</tr>
<tr>
<td>exprs/3</td>
<td>erl_eval , 139</td>
</tr>
<tr>
<td>exprs/4</td>
<td>erl_eval , 139</td>
</tr>
<tr>
<td>extension/1</td>
<td>filename , 200</td>
</tr>
<tr>
<td>extension/3</td>
<td>sofs , 421</td>
</tr>
<tr>
<td>extract/1</td>
<td>erl_tar , 162</td>
</tr>
<tr>
<td>extract/2</td>
<td>zip , 475</td>
</tr>
<tr>
<td>family/2</td>
<td>sofs , 422</td>
</tr>
<tr>
<td>family_difference/2</td>
<td>sofs , 422</td>
</tr>
<tr>
<td>family_domain/1</td>
<td>sofs , 422</td>
</tr>
<tr>
<td>family_field/1</td>
<td>sofs , 422</td>
</tr>
<tr>
<td>family_intersection/1</td>
<td>sofs , 423</td>
</tr>
<tr>
<td>family_intersection/2</td>
<td>sofs , 423</td>
</tr>
<tr>
<td>family_projection/2</td>
<td>sofs , 423</td>
</tr>
<tr>
<td>family_range/1</td>
<td>sofs , 423</td>
</tr>
<tr>
<td>family_specification/2</td>
<td>sofs , 424</td>
</tr>
<tr>
<td>family_to_digraph/2</td>
<td>sofs , 424</td>
</tr>
<tr>
<td>family_to_relation/1</td>
<td>sofs , 424</td>
</tr>
<tr>
<td>family_union/1</td>
<td>sofs , 425</td>
</tr>
<tr>
<td>family_union/2</td>
<td>sofs , 425</td>
</tr>
<tr>
<td>fetch/2</td>
<td></td>
</tr>
</tbody>
</table>
Index of Modules and Functions

- `dict`, 121
- `orddict`, 297
- `fetch_keys/1`
  - `dict`, 121
  - `orddict`, 297
- `field/1`
  - `sofs`, 425
- `file2tab/1`
  - `ets`, 168
- `file2tab/2`
  - `ets`, 168
- `file_size/1`
  - `filelib`, 195
- `file_sorter`
  - `check/1`, 194
  - `check/2`, 194
  - `keycheck/2`, 194
  - `keycheck/3`, 194
  - `keymerge/3`, 194
  - `keymerge/4`, 194
  - `keysort/2`, 193
  - `keysort/3`, 193
  - `keysort/4`, 193
  - `merge/2`, 193
  - `merge/3`, 193
  - `sort/1`, 193
  - `sort/2`, 193
  - `sort/3`, 193
- `filelib`
  - `ensure_dir/1`, 195
  - `file_size/1`, 195
  - `fold_files/5`, 195
  - `is_dir/1`, 196
  - `is_file/1`, 196
  - `is_regular/1`, 196
  - `last_modified/1`, 196
  - `wildcard/1`, 196
  - `wildcard/2`, 197
- `filename`
  - `absname/1`, 198
  - `absname/2`, 199
  - `absname_join/2`, 199
  - `basename/1`, 199
  - `basename/2`, 199
  - `dirname/1`, 200
  - `extension/1`, 200
  - `find_src/1`, 202
  - `find_src/2`, 202
  - `flatten/1`, 200
- `join/1`, 200
- `join/2`, 201
- `nativename/1`, 201
- `pathtype/1`, 201
- `rootname/1`, 201
- `rootname/2`, 201
- `split/1`, 202
- `filter/2`
  - `dict`, 121
  - `gb_sets`, 206
  - `lists`, 268
  - `orddict`, 297
  - `ordsets`, 303
  - `queue`, 339
  - `sets`, 399
- `find/2`
  - `dict`, 121
  - `orddict`, 297
- `find_src/1`
  - `filename`, 202
- `find_src/2`
  - `filename`, 202
- `first/1`
  - `dets`, 106
  - `ets`, 168
- `first_match/2`
  - `regexp`, 392
- `fix/1`
  - `array`, 81
- `flatlength/1`
  - `lists`, 268
- `flatmap/2`
  - `lists`, 268
- `flatten/1`
  - `filename`, 200
  - `lists`, 268
- `flatten/2`
  - `lists`, 268
- `flush/0`
  - `c`, 96
- `flush_receive/0`
  - `lib`, 264
- `fold/3`
  - `dict`, 121
  - `gb_sets`, 206
  - `orddict`, 297
Index of Modules and Functions

- **ordsets**, 303
- **sets**, 399
- **fold/4**
  - **qlc**, 327
- **fold_files/5**
  - **filelib**, 195
- **foldl/2**
  - **array**, 81
- **foldl/3**
  - **dets**, 107
  - **ets**, 169
  - **lists**, 269
- **foldr/2**
  - **array**, 81
- **foldr/3**
  - **dets**, 107
  - **ets**, 169
  - **lists**, 269
- **foreach/2**
  - **lists**, 269
- **form/1**
  - **erlpp**, 151
- **form/2**
  - **erlpp**, 151
- **format/1**
  - **io**, 251
  - **proc_lib**, 311
- **format/2**
  - **io_lib**, 261
- **format/3**
  - **io**, 251
- **format_error/1**
  - **beam_lib**, 93
  - **erlJint**, 147
  - **erl_parse**, 149
  - **erl_scan**, 158
  - **erl_tar**, 163
  - **ms_transform**, 295
  - **qlc**, 328
  - **regexp**, 394
  - **win32reg**, 471
- **fread/2**
  - **io_lib**, 261
- **fread/3**
  - **io**, 255
  - **io_lib**, 262
  - **from_dets/2**
    - **ets**, 169
  - **from_ets/2**
    - **dets**, 107
  - **from_external/2**
    - **sofs**, 425
  - **from_list/1**
    - **array**, 81
    - **dict**, 121
    - **gb_sets**, 206
    - **orddict**, 297
    - **ordsets**, 302
    - **queue**, 339
    - **sets**, 398
  - **from_orrdict/1**
    - **array**, 82
    - **gb_trees**, 211
  - **from_orrset/1**
    - **gb_sets**, 206
  - **from_sets/1**
    - **sofs**, 425, 426
  - **from_term/2**
    - **sofs**, 426
  - **fun2ms/1**
    - **ets**, 169
  - **function/1**
    - **erlpp**, 151
  - **function/2**
    - **erlpp**, 151
  - **fwrite/1**
    - **io**, 251
  - **fwrite/2**
    - **io_lib**, 261
  - **fwrite/3**
    - **io**, 251

- **gb_sets**
  - **add/**, 205
  - **add_element/**, 205
  - **balance/**, 205
  - **del_element/**, 205
  - **delete/**, 205
  - **delete_any/**, 205
  - **difference/**, 205
  - **empty/**, 206
  - **filter/**, 206
### Index of Modules and Functions

<table>
<thead>
<tr>
<th>gb_trees</th>
<th>gen_server</th>
<th>gen_event</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance/1, 210</td>
<td>cancel_timer/1</td>
<td>add_handler/3</td>
</tr>
<tr>
<td>delete/2, 210</td>
<td>enter_loop/4</td>
<td>call/3, 218</td>
</tr>
<tr>
<td>delete_any/2, 211</td>
<td>enter_loop/5</td>
<td>call/4, 218</td>
</tr>
<tr>
<td>empty/0, 211</td>
<td>enter_loop/6</td>
<td>delete_handler/3, 219</td>
</tr>
<tr>
<td>enter/3, 211</td>
<td></td>
<td>Module:code_change/3, 223</td>
</tr>
<tr>
<td>from_orddict/1, 211</td>
<td></td>
<td>Module:handle_call/2, 222</td>
</tr>
<tr>
<td>get/2, 211</td>
<td></td>
<td>Module:handle_event/2, 221</td>
</tr>
<tr>
<td>insert/3, 212</td>
<td></td>
<td>Module:handle_info/2, 222</td>
</tr>
<tr>
<td>is_definded/2, 212</td>
<td></td>
<td>Module:init/1, 221</td>
</tr>
<tr>
<td>is_empty/1, 212</td>
<td></td>
<td>Module:terminate/2, 223</td>
</tr>
<tr>
<td>iterator/1, 212</td>
<td></td>
<td>notify/2, 218</td>
</tr>
<tr>
<td>keys/1, 212</td>
<td></td>
<td>start/0, 216</td>
</tr>
<tr>
<td>largest/1, 212</td>
<td></td>
<td>start/1, 216</td>
</tr>
<tr>
<td>lookup/2, 211</td>
<td></td>
<td>start_link/0, 216</td>
</tr>
<tr>
<td>map/2, 212</td>
<td></td>
<td>start_link/1, 216</td>
</tr>
<tr>
<td>next/1, 213</td>
<td></td>
<td>stop/1, 221</td>
</tr>
<tr>
<td>size/1, 213</td>
<td></td>
<td>swap_handler/5, 219</td>
</tr>
<tr>
<td>smallest/1, 213</td>
<td></td>
<td>swap_sup_handler/5, 220</td>
</tr>
<tr>
<td>take_largest/1, 213</td>
<td></td>
<td>sync_notify/2, 218</td>
</tr>
<tr>
<td>take_smallest/1, 213</td>
<td></td>
<td>which_handlers/1, 220</td>
</tr>
<tr>
<td>to_list/1, 213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>update/3, 214</td>
<td></td>
<td></td>
</tr>
<tr>
<td>values/1, 214</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gb_trees</th>
<th>gen_server</th>
<th>gen_event</th>
</tr>
</thead>
<tbody>
<tr>
<td>new/0, 206</td>
<td>abcast/2, 240</td>
<td>add_handler/3</td>
</tr>
<tr>
<td>next/1, 208</td>
<td>abcast/3, 240</td>
<td>call/3, 238</td>
</tr>
<tr>
<td>singleton/1, 208</td>
<td></td>
<td>call/2, 238</td>
</tr>
<tr>
<td>size/1, 208</td>
<td></td>
<td>cast/2, 240</td>
</tr>
<tr>
<td>subtract/2, 206</td>
<td></td>
<td>enter_loop/3, 241</td>
</tr>
<tr>
<td>take_largest/1, 208</td>
<td></td>
<td></td>
</tr>
<tr>
<td>take_smallest/1, 208</td>
<td></td>
<td></td>
</tr>
<tr>
<td>to_list/1, 208</td>
<td></td>
<td></td>
</tr>
<tr>
<td>union/1, 209</td>
<td></td>
<td></td>
</tr>
<tr>
<td>union/2, 209</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gb_trees</th>
<th>gen_server</th>
<th>gen_event</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance/1, 210</td>
<td></td>
<td>add_sup_handler/3, 217</td>
</tr>
<tr>
<td>delete/2, 210</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete_any/2, 211</td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty/0, 211</td>
<td></td>
<td></td>
</tr>
<tr>
<td>enter/3, 211</td>
<td></td>
<td></td>
</tr>
<tr>
<td>from_orddict/1, 211</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get/2, 211</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert/3, 212</td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_definded/2, 212</td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_empty/1, 212</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iterator/1, 212</td>
<td></td>
<td></td>
</tr>
<tr>
<td>keys/1, 212</td>
<td></td>
<td></td>
</tr>
<tr>
<td>largest/1, 212</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lookup/2, 211</td>
<td></td>
<td></td>
</tr>
<tr>
<td>map/2, 212</td>
<td></td>
<td></td>
</tr>
<tr>
<td>next/1, 213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size/1, 213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>smallest/1, 213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>take_largest/1, 213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>take_smallest/1, 213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>to_list/1, 213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>update/3, 214</td>
<td></td>
<td></td>
</tr>
<tr>
<td>values/1, 214</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

STDLIB
Index of Modules and Functions

- enter_loop/4, 241
- enter_loop/5, 241
- Module: code_change/3, 244
- Module: handle_call/3, 242
- Module: handle_cast/2, 243
- Module: handle_info/2, 243
- Module: init/1, 242
- Module: terminate/2, 244
- multi_call/2, 239
- multi_call/3, 239
- multi_call/4, 239
- reply/2, 240
- start/3, 238
- start/4, 238
- start_link/3, 237
- start_link/4, 237

get/1
  array , 82
  queue , 340

get/2
  gb_trees , 211

get_all_values/2
  proplists , 314

get_bool/2
  proplists , 314

get_chars/3
  io , 247

get_cycle/2
  digraph , 128

get_debug/3
  sys , 458

get_keys/1
  proplists , 314

get_line/2
  io , 247

get_node/0
  pool , 307

get_nodes/0
  pool , 307

get_path/3
  digraph , 128

get_r/1
  queue , 340

get_short_cycle/2
  digraph , 128

get_short_path/3
  digraph , 128

get_status/1
  sys , 456

get_status/2
  sys , 456

get_value/2
  proplists , 315

get_value/3
  proplists , 315

getopts/1
  io , 248

gregorian_days_to_date/1
  calendar , 100

gregorian_seconds_to_datetime/1
  calendar , 100

gsub/3
  regexp , 393

guard/1
  erl_pp , 151

guard/2
  erl_pp , 151

guard_bif/2
  erl_internal , 144

handle_debug/1
  sys , 458

handle_system_msg/6
  sys , 458

head/1
  queue , 341

help/0
  c , 96

hibernate/3
  proc_lib , 312

history/1
  shell , 409

hms/3
  timer , 463

hours/1
  timer , 463

i/o
  c , 96
  ets , 171
Index of Modules and Functions

i/1
  ets, 171

i/3
  c, 96

image/2
  sofs, 427

in/2
  queue, 338

in_degree/2
  digraph, 129

in_edges/2
  digraph, 129

in_neighbours/2
  digraph, 129

in_r/2
  queue, 338

indentation/2
  io_lib, 263

info/1
  beam_lib, 91
  dets, 107
  digraph, 129
  ets, 171

info/2
  dets, 108
  ets, 171
  qlc, 328

init/1
  queue, 342

init/3
  log.mf.h, 282

init/4
  log.mf.h, 282

init_ack/1
  proc_lib, 310

init_ack/2
  proc_lib, 310

init_table/2
  ets, 172

init_table/3
  dets, 108

initial_call/1
  proc_lib, 311

insert/2
  dets, 109
  ets, 172
  gb_sets, 206

insert/3
  gb_trees, 212

insert_new/2
  dets, 109
  ets, 173

install/3
  sys, 457

install/4
  sys, 457

intersection/1
  gb_sets, 207
  ordsets, 303
  sets, 399
  sofs, 427

intersection/2
  gb_sets, 207
  ordsets, 302
  sets, 398
  sofs, 427

intersection_of_family/1
  sofs, 427

inverse/1
  sofs, 427

inverse_image/2
  sofs, 428

io
  columns/1, 247
  format/1, 251
  format/3, 251
  fread/3, 255
  fwrite/1, 251
  fwrite/3, 251
  get_chars/3, 247
  get_line/2, 247
  getopt/1, 248
  nl/1, 247
  parse.erl_exprs/1, 258
  parse.erl_exprs/3, 258
  parse.erl_form/1, 258
  parse.erl_form/3, 258
  put_chars/2, 247
  read/2, 250
  read/3, 250
  rows/1, 257
  scan.erl_exprs/1, 257
<table>
<thead>
<tr>
<th>Function/Module</th>
<th>Module/Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan_erl_exprs/3</td>
<td>gb_sets, 207</td>
<td>257</td>
</tr>
<tr>
<td>scan_erl_form/1, 257</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>scan_erl_form/3, 257</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>setopts/2, 248</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>write/2, 250</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>io_lib</td>
<td>gb_sets, 207</td>
<td>257</td>
</tr>
<tr>
<td>char_list/1, 263</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>deep_char_list/1, 263</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>format/2, 261</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>fread/2, 261</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>fread/3, 262</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>fwrite/2, 261</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>indentation/2, 263</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>nl/0, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>print/1, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>print/4, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>printable_list/1, 263</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>write/1, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>write/2, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>write_atom/1, 262</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>write_char/1, 262</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>write_string/1, 262</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_a_function/1, 260</td>
<td>gb_sets, 207</td>
<td>257</td>
</tr>
<tr>
<td>is_acyclic/1, 260</td>
<td>gb_sets, 207</td>
<td>257</td>
</tr>
<tr>
<td>is_arborescence/1, 263</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_array/1, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_compatible_bchunk_format/2, 263</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_compiled_ms/1, 263</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_defined/2, 263</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_dir/1, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_empty/1, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_equal/2, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_element/2, 263</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_empty_set/1, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_file/1, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_fix/1, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_guard_test/1, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_key/2, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_leap_year/1, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_member/2, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_subset/2, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_tree/1, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
<tr>
<td>is_type/1, 260</td>
<td>gb_trees, 212</td>
<td></td>
</tr>
</tbody>
</table>

STDLIB
<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>gb_sets</td>
<td>gb_trees</td>
<td>208</td>
</tr>
<tr>
<td>gb_trees</td>
<td></td>
<td>212</td>
</tr>
<tr>
<td>join/1</td>
<td>filename</td>
<td>200</td>
</tr>
<tr>
<td>join/2</td>
<td>filename</td>
<td>201</td>
</tr>
<tr>
<td>join/2</td>
<td>pg</td>
<td>304</td>
</tr>
<tr>
<td>join/2</td>
<td>queue</td>
<td>339</td>
</tr>
<tr>
<td>join/2</td>
<td>string</td>
<td>439</td>
</tr>
<tr>
<td>join/4</td>
<td>sofs</td>
<td>429</td>
</tr>
<tr>
<td>keycheck/2</td>
<td>file_sorter</td>
<td>194</td>
</tr>
<tr>
<td>keycheck/3</td>
<td>file_sorter</td>
<td>194</td>
</tr>
<tr>
<td>keydelete/3</td>
<td>lists</td>
<td>269</td>
</tr>
<tr>
<td>keyfind/3</td>
<td>lists</td>
<td>270</td>
</tr>
<tr>
<td>keymap/3</td>
<td>lists</td>
<td>270</td>
</tr>
<tr>
<td>keymember/3</td>
<td>lists</td>
<td>270</td>
</tr>
<tr>
<td>keymerge/3</td>
<td>file_sorter</td>
<td>194</td>
</tr>
<tr>
<td>keymerge/4</td>
<td>file_sorter</td>
<td>194</td>
</tr>
<tr>
<td>keyreplace/4</td>
<td>lists</td>
<td>271</td>
</tr>
<tr>
<td>keys/1</td>
<td>gb_trees</td>
<td>212</td>
</tr>
<tr>
<td>keysort/2</td>
<td>file_sorter</td>
<td>193</td>
</tr>
<tr>
<td>keysort/3</td>
<td>file_sorter</td>
<td>193</td>
</tr>
<tr>
<td>keysort/3</td>
<td>qlc</td>
<td>329</td>
</tr>
<tr>
<td>keysort/4</td>
<td>file_sorter</td>
<td>193</td>
</tr>
<tr>
<td>keystore/4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lists</td>
<td></td>
<td>271</td>
</tr>
<tr>
<td>keytake/3</td>
<td>lists</td>
<td>272</td>
</tr>
<tr>
<td>kill_after/1</td>
<td>timer</td>
<td>462</td>
</tr>
<tr>
<td>kill_after/2</td>
<td>timer</td>
<td>462</td>
</tr>
<tr>
<td>l/1</td>
<td>c</td>
<td>96</td>
</tr>
<tr>
<td>lait/1</td>
<td>queue</td>
<td>342</td>
</tr>
<tr>
<td>largest/1</td>
<td>gb_sets</td>
<td>208</td>
</tr>
<tr>
<td>largest/1</td>
<td>gb_trees</td>
<td>212</td>
</tr>
<tr>
<td>last/1</td>
<td>ets</td>
<td>173</td>
</tr>
<tr>
<td>last/1</td>
<td>lists</td>
<td>272</td>
</tr>
<tr>
<td>last/1</td>
<td>queue</td>
<td>342</td>
</tr>
<tr>
<td>last_day_of_the_month/2</td>
<td>calendar</td>
<td>100</td>
</tr>
<tr>
<td>last_modified/1</td>
<td>filelib</td>
<td>196</td>
</tr>
<tr>
<td>lc/1</td>
<td>c</td>
<td>96</td>
</tr>
<tr>
<td>left/2</td>
<td>string</td>
<td>441</td>
</tr>
<tr>
<td>left/3</td>
<td>string</td>
<td>441</td>
</tr>
<tr>
<td>len/1</td>
<td>queue</td>
<td>338</td>
</tr>
<tr>
<td>len/1</td>
<td>string</td>
<td>438</td>
</tr>
<tr>
<td>liat/1</td>
<td>queue</td>
<td>342</td>
</tr>
<tr>
<td>liat/1</td>
<td>lib</td>
<td></td>
</tr>
<tr>
<td>liat/1</td>
<td>lib</td>
<td></td>
</tr>
<tr>
<td>lib</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lib</td>
<td></td>
<td></td>
</tr>
<tr>
<td>error_message/2</td>
<td></td>
<td>264</td>
</tr>
<tr>
<td>flush_receive/0</td>
<td></td>
<td>264</td>
</tr>
<tr>
<td>nonl/1</td>
<td></td>
<td>264</td>
</tr>
<tr>
<td>progname/0</td>
<td></td>
<td>264</td>
</tr>
<tr>
<td>send/2</td>
<td></td>
<td>264</td>
</tr>
<tr>
<td>sendw/2</td>
<td></td>
<td>265</td>
</tr>
<tr>
<td>list_dir/1</td>
<td>zip</td>
<td>476</td>
</tr>
<tr>
<td>list_dir/2</td>
<td>zip</td>
<td>476</td>
</tr>
<tr>
<td>Function</td>
<td>Module</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------</td>
<td>------</td>
</tr>
<tr>
<td>sublist/2</td>
<td>erlInternal</td>
<td>145</td>
</tr>
<tr>
<td>lists</td>
<td></td>
<td></td>
</tr>
<tr>
<td>all/2</td>
<td></td>
<td>266</td>
</tr>
<tr>
<td>any/2</td>
<td></td>
<td>266</td>
</tr>
<tr>
<td>append/1</td>
<td></td>
<td>266</td>
</tr>
<tr>
<td>append/2</td>
<td></td>
<td>267</td>
</tr>
<tr>
<td>concat/1</td>
<td></td>
<td>267</td>
</tr>
<tr>
<td>delete/2</td>
<td></td>
<td>267</td>
</tr>
<tr>
<td>dropwhile/2</td>
<td></td>
<td>267</td>
</tr>
<tr>
<td>duplicate/2</td>
<td></td>
<td>267</td>
</tr>
<tr>
<td>filter/2</td>
<td></td>
<td>267</td>
</tr>
<tr>
<td>flatlength/1</td>
<td></td>
<td>268</td>
</tr>
<tr>
<td>flatmap/2</td>
<td></td>
<td>268</td>
</tr>
<tr>
<td>flatten/1</td>
<td></td>
<td>268</td>
</tr>
<tr>
<td>flatten/2</td>
<td></td>
<td>268</td>
</tr>
<tr>
<td>foldl/3</td>
<td></td>
<td>269</td>
</tr>
<tr>
<td>foldr/3</td>
<td></td>
<td>269</td>
</tr>
<tr>
<td>foreach/2</td>
<td></td>
<td>269</td>
</tr>
<tr>
<td>keydelete/3</td>
<td></td>
<td>269</td>
</tr>
<tr>
<td>keyfind/3</td>
<td></td>
<td>270</td>
</tr>
<tr>
<td>keymap/3</td>
<td></td>
<td>270</td>
</tr>
<tr>
<td>keymember/3</td>
<td></td>
<td>270</td>
</tr>
<tr>
<td>keymerge/3</td>
<td></td>
<td>270</td>
</tr>
<tr>
<td>keyreplace/4</td>
<td></td>
<td>271</td>
</tr>
<tr>
<td>keysearch/3</td>
<td></td>
<td>271</td>
</tr>
<tr>
<td>keysort/2</td>
<td></td>
<td>271</td>
</tr>
<tr>
<td>keystore/4</td>
<td></td>
<td>271</td>
</tr>
<tr>
<td>keytake/3</td>
<td></td>
<td>272</td>
</tr>
<tr>
<td>last/1</td>
<td></td>
<td>272</td>
</tr>
<tr>
<td>map/2</td>
<td></td>
<td>272</td>
</tr>
<tr>
<td>mapfoldl/3</td>
<td></td>
<td>272</td>
</tr>
<tr>
<td>mapfoldr/3</td>
<td></td>
<td>273</td>
</tr>
<tr>
<td>max/1</td>
<td></td>
<td>273</td>
</tr>
<tr>
<td>member/2</td>
<td></td>
<td>273</td>
</tr>
<tr>
<td>merge/1</td>
<td></td>
<td>273</td>
</tr>
<tr>
<td>merge/2</td>
<td></td>
<td>273</td>
</tr>
<tr>
<td>merge/3</td>
<td></td>
<td>273</td>
</tr>
<tr>
<td>merge3/3</td>
<td></td>
<td>274</td>
</tr>
<tr>
<td>min/1</td>
<td></td>
<td>274</td>
</tr>
<tr>
<td>nth/2</td>
<td></td>
<td>274</td>
</tr>
<tr>
<td>nthtail/2</td>
<td></td>
<td>274</td>
</tr>
<tr>
<td>partition/2</td>
<td></td>
<td>275</td>
</tr>
<tr>
<td>prefix/2</td>
<td></td>
<td>275</td>
</tr>
<tr>
<td>reverse/1</td>
<td></td>
<td>275</td>
</tr>
<tr>
<td>reverse/2</td>
<td></td>
<td>275</td>
</tr>
<tr>
<td>seq/2</td>
<td></td>
<td>275</td>
</tr>
<tr>
<td>seq/3</td>
<td></td>
<td>275</td>
</tr>
<tr>
<td>sort/1</td>
<td></td>
<td>276</td>
</tr>
<tr>
<td>sort/2</td>
<td></td>
<td>276</td>
</tr>
<tr>
<td>split/2</td>
<td></td>
<td>276</td>
</tr>
<tr>
<td>splitwith/2</td>
<td></td>
<td>276</td>
</tr>
<tr>
<td>sublist/3</td>
<td></td>
<td>277</td>
</tr>
<tr>
<td>sublist/2</td>
<td></td>
<td>277</td>
</tr>
<tr>
<td>subtract/2</td>
<td></td>
<td>277</td>
</tr>
<tr>
<td>suffix/2</td>
<td></td>
<td>278</td>
</tr>
<tr>
<td>sum/1</td>
<td></td>
<td>278</td>
</tr>
<tr>
<td>takewhile/2</td>
<td></td>
<td>278</td>
</tr>
<tr>
<td>ukeymerge/3</td>
<td></td>
<td>278</td>
</tr>
<tr>
<td>ukeysort/2</td>
<td></td>
<td>278</td>
</tr>
<tr>
<td>umerge/1</td>
<td></td>
<td>279</td>
</tr>
<tr>
<td>umerge/2</td>
<td></td>
<td>279</td>
</tr>
<tr>
<td>umerge/3</td>
<td></td>
<td>279</td>
</tr>
<tr>
<td>umerge3/3</td>
<td></td>
<td>279</td>
</tr>
<tr>
<td>unzip/1</td>
<td></td>
<td>279</td>
</tr>
<tr>
<td>unzip3/1</td>
<td></td>
<td>280</td>
</tr>
<tr>
<td>usort/1</td>
<td></td>
<td>280</td>
</tr>
<tr>
<td>usort/2</td>
<td></td>
<td>280</td>
</tr>
<tr>
<td>zip/2</td>
<td></td>
<td>280</td>
</tr>
<tr>
<td>zip3/3</td>
<td></td>
<td>280</td>
</tr>
<tr>
<td>zipwith/3</td>
<td></td>
<td>281</td>
</tr>
<tr>
<td>zipwith3/4</td>
<td></td>
<td>281</td>
</tr>
<tr>
<td>local_time/0</td>
<td>calendar</td>
<td>101</td>
</tr>
<tr>
<td>local_time_to_universal_time/2</td>
<td>calendar</td>
<td>101</td>
</tr>
<tr>
<td>local_time_to_universal_time_dst/2</td>
<td>calendar</td>
<td>101</td>
</tr>
<tr>
<td>log/1</td>
<td>math</td>
<td>283</td>
</tr>
<tr>
<td>log/2</td>
<td>sys</td>
<td>455</td>
</tr>
<tr>
<td>log/3</td>
<td>sys</td>
<td>455</td>
</tr>
<tr>
<td>log10/1</td>
<td>math</td>
<td>283</td>
</tr>
<tr>
<td>log_mf_h</td>
<td>init/3</td>
<td>282</td>
</tr>
<tr>
<td>log_to_file/2</td>
<td>sys</td>
<td>455</td>
</tr>
<tr>
<td>log_to_file/3</td>
<td>sys</td>
<td>455</td>
</tr>
<tr>
<td>lookup/2</td>
<td>dets</td>
<td>110</td>
</tr>
<tr>
<td>lookup_all/2</td>
<td>gb_trees</td>
<td>211</td>
</tr>
<tr>
<td>lookup_all/2</td>
<td>proplists</td>
<td>315</td>
</tr>
</tbody>
</table>
Index of Modules and Functions

proplists, 315
lookup_element/3
ets, 174
loop_vertices/1
digraph_utils, 134
ls/0
c, 96
ls/1
c, 96
m/0
c, 96
m/1
c, 97
map/2
array, 82
dict, 122
gb_trees, 212
lists, 272
orddict, 298
mapfoldl/3
lists, 272
mapfoldr/3
lists, 273
match/1
dets, 110
ets, 175
match/2
dets, 111
ets, 174
regexp, 392
match/3
dets, 111
ets, 175
match_delete/2
dets, 111
ets, 175
match_object/1
dets, 112
ets, 176
match_object/2
dets, 112
ets, 176
match_object/3
dets, 112
ets, 176
match_spec_compile/1
ets, 176
match_spec_run/2
ets, 177
matches/2
regexp, 393
math
acos/1, 283
acosh/1, 283
asin/1, 283
asinh/1, 283
atan/1, 283
atan2/2, 283
atanh/1, 283
cos/1, 283
cosh/1, 283
erf/1, 284
erfc/1, 284
exp/1, 283
log/1, 283
log10/1, 283
pi/0, 283
pow/2, 283
sin/1, 283
sinh/1, 283
sqrt/1, 283
tan/1, 283
tanh/1, 283
max/1
lists, 273
md5/1
beam_lib, 91
member/2
dets, 113
ets, 177
lists, 273
queue, 340
members/1
pg, 305
memory/0
c, 97
memory/1
c, 97
merge/1
lists, 273
merge/2
file_sorter, 193
lists, 273
merge/3
  dict , 122
  file_sorter , 193
  lists , 273
  orddict , 298
merge3/3
  lists , 274
mime_decode/1
  base64 , 86
mime_decode_to_string/1
  base64 , 86
min/1
  lists , 274
minutes/1
  timer , 463
Mod: system_code_change/4
  sys , 459
Mod: system_continue/3
  sys , 459
Mod: system_terminate/4
  sys , 459
module/1
  erl_lint , 146
module/2
  erl_expand_records , 142
  erl_lint , 146
module/3
  erl_lint , 146
Module: code_change/3
  gen_event , 223
  gen_server , 244
Module: code_change/4
  gen_fsm , 235
Module: handle_call/2
  gen_event , 222
Module: handle_call/3
  gen_server , 242
Module: handle_cast/2
  gen_server , 243
Module: handle_event/2
  gen_event , 221
Module: handle_event/3
  gen_fsm , 232
Module: handle_info/2
  gen_event , 222
  gen_server , 243
Module: handle_info/3
  gen_fsm , 234
Module: handle_sync_event/4
  gen_fsm , 233
Module: init/1
  gen_event , 221
  gen_fsm , 231
  gen_server , 242
  supervisor , 450
  supervisor_bridge , 452
Module: StateName/2
  gen_fsm , 231
Module: StateName/3
  gen_fsm , 232
Module: terminate/2
  gen_event , 223
  gen_server , 244
  supervisor_bridge , 452
Module: terminate/3
  gen_fsm , 234
ms_transform
  format_error/1, 295
  parse_transform/2, 294
  transform_from_shell/3, 294
multi_call/2
  gen_server , 239
multi_call/3
  gen_server , 239
multi_call/4
  gen_server , 239
multiple_relative_product/2
  sofs , 429
nativename/1
  filename , 201
nc/1
  c , 97
nc/2
  c , 97
new/0
  array , 82
  dict , 122
  digraph , 130
  gb_sets , 206
Index of Modules and Functions

orddict, 298
ordsets, 301
queue, 338
sets, 397
new/1
    array, 82, 83
digraph, 130
new/2
    ets, 178
new_bindings/0
    erl_eval, 140
next/1
    gb_sets, 208
gb_trees, 213
next/2
    dets, 113
    ets, 178
next_answers/2
    qlc, 330
ni/0
    c, 96
nl/0
    io_lib, 260
nl/1
    c, 97
    io, 247
no_debug/1
    sys, 456
no_debug/2
    sys, 456
no_edges/1
    digraph, 130
no_elements/1
    sofs, 430
no_vertices/1
    digraph, 130
nonl/1
    lib, 264
normalise/1
    erl_parse, 149
normalize/2
    proplists, 315
notify/2
    gen_event, 218
    now_diff/2
        timer, 463
        now_to_datetime/1
            calendar, 102
        now_to_local_time/1
            calendar, 101
        now_to_universal_time/1
            calendar, 102
nregs/0
    c, 98
nth/2
    lists, 274
nthtail/2
    lists, 274
op_type/2
    erl_internal, 145
open/1
    win32reg, 471
open/2
    epp, 137
    erl_tar, 163
open/3
    epp, 137
open_file/1
    dets, 113
open_file/2
    dets, 113
orddict
    append/3, 296
    append_list/3, 296
    erase/2, 296
    fetch/2, 297
    fetch_keys/1, 297
    filter/2, 297
    find/2, 297
    fold/3, 297
    from_list/1, 297
    is_key/2, 298
    map/2, 298
    merge/3, 298
    new/0, 298
    size/1, 298
    store/3, 298
    to_list/1, 299
    update/3, 299
    update/4, 299
Index of Modules and Functions

update_counter/3, 299

ordsets
add_element/2, 302
del_element/2, 302
filter/2, 303
fold/3, 303
from_list/1, 302
intersection/1, 303
intersection/2, 302
is_disjoint/2, 303
is_element/2, 302
is_set/1, 301
is_subset/2, 303
new/0, 301
size/1, 301
subtract/2, 303
to_list/1, 301
union/1, 302
union/2, 302

out/1
queue, 338
out_degree/2
digraph, 130
out_edges/2
digraph, 130
out_neighbours/2
digraph, 130
out_r/1
queue, 339

parse/1
regexp, 394
parse_erl_exprs/1
io, 258
parse_erl_exprs/3
io, 258
parse_erl_form/1
epp, 137
io, 258
parse_erl_form/3
io, 258
parse_exprs/1
erl_parse, 148
parse_file/3
epp, 137
parse_form/1

erl_parse, 148
parse_term/1
erl_parse, 148
parse_transform/2
erl_id_trans, 143
ms_transform, 294
partition/1
sofs, 430
partition/2
lists, 275
sofs, 430
partition/3
sofs, 430
partition_family/2
sofs, 431
pathtype/1
filename, 201
peek/1
queue, 341
peek_r/1
queue, 341

pg
create/1, 304
create/2, 304
esend/2, 305
join/2, 304
members/1, 305
send/2, 305

pi/0
math, 283

pid/3
c, 97

pid2name/1
dets, 115

pool
attach/1, 306
get_node/0, 307
get_nodes/0, 307
spawn/3, 307
spawn_link/3, 307
start/1, 306
start/2, 306
stop/0, 307

postorder/1
digraph_utils, 134
Index of Modules and Functions

pow/2
  math, 283
prefix/2
  lists, 275
preorder/1
  digraph_utils, 134
prev/2
  ets, 179
print/1
  io_lib, 260
print/4
  io_lib, 260
print_log/1
  sys, 459
printable_list/1
  io_lib, 263
proc_lib
  format/1, 311
  hibernate/3, 312
  init_ack/1, 310
  init_ack/2, 310
  initial_call/1, 311
  spawn/1, 308
  spawn/2, 308
  spawn/3, 308
  spawn/4, 308
  spawn_link/1, 308
  spawn_link/2, 308
  spawn_link/3, 309
  spawn_link/4, 309
  spawn_opt/2, 309
  spawn_opt/3, 309
  spawn_opt/4, 309
  spawn_opt/5, 309
  start/3, 309
  start/4, 309
  start/5, 309
  start_link/3, 309
  start_link/4, 309
  start_link/5, 309
  translate_initial_call/1, 311
product/1
  sofs, 431
product/2
  sofs, 431
prognme/0
  lib, 264
projection/2
  sofs, 432
property/1
  proplists, 316
property/2
  proplists, 316
proplists
  append_values/2, 313
  compact/1, 313
  delete/2, 313
  expand/2, 313
  get_all_values/2, 314
  get_bool/2, 314
  get_keys/1, 314
  get_value/2, 315
  get_value/3, 315
  is_defined/2, 315
  lookup/2, 315
  lookup_all/2, 315
  normalize/2, 315
  property/1, 316
  property/2, 316
  split/2, 316
  substitute_aliases/2, 317
  substitute_negations/2, 317
  unfold/1, 317
pseudo/1
  slave, 414
pseudo/2
  slave, 414
pspawn/3
  pool, 307
pspawn_link/3
  pool, 307
put_chars/2
  io, 247
pwd/0
  c, 97
q/0
  c, 98
q/2
  qlc, 330
qlc
  append/1, 326
  append/2, 326
  cursor/2, 326
<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete_cursor/1</td>
<td></td>
<td>327</td>
</tr>
<tr>
<td>e/2</td>
<td></td>
<td>327</td>
</tr>
<tr>
<td>eval/2</td>
<td></td>
<td>327</td>
</tr>
<tr>
<td>fold/4</td>
<td></td>
<td>327</td>
</tr>
<tr>
<td>format_error/1</td>
<td></td>
<td>328</td>
</tr>
<tr>
<td>info/2</td>
<td></td>
<td>328</td>
</tr>
<tr>
<td>keysort/3</td>
<td></td>
<td>329</td>
</tr>
<tr>
<td>next_answers/2</td>
<td></td>
<td>330</td>
</tr>
<tr>
<td>q/2</td>
<td></td>
<td>330</td>
</tr>
<tr>
<td>sort/2</td>
<td></td>
<td>333</td>
</tr>
<tr>
<td>string_to_handle/3</td>
<td></td>
<td>333</td>
</tr>
<tr>
<td>table/2</td>
<td></td>
<td>333</td>
</tr>
<tr>
<td>cons/2</td>
<td></td>
<td>341</td>
</tr>
<tr>
<td>daeh/1</td>
<td></td>
<td>342</td>
</tr>
<tr>
<td>drop/1</td>
<td></td>
<td>340</td>
</tr>
<tr>
<td>drop_r/1</td>
<td></td>
<td>340</td>
</tr>
<tr>
<td>filter/2</td>
<td></td>
<td>339</td>
</tr>
<tr>
<td>from_list/1</td>
<td></td>
<td>339</td>
</tr>
<tr>
<td>get/1</td>
<td></td>
<td>340</td>
</tr>
<tr>
<td>get_r/1</td>
<td></td>
<td>340</td>
</tr>
<tr>
<td>head/1</td>
<td></td>
<td>341</td>
</tr>
<tr>
<td>in/2</td>
<td></td>
<td>338</td>
</tr>
<tr>
<td>in_r/2</td>
<td></td>
<td>338</td>
</tr>
<tr>
<td>init/1</td>
<td></td>
<td>342</td>
</tr>
<tr>
<td>is_empty/1</td>
<td></td>
<td>338</td>
</tr>
<tr>
<td>is_queue/1</td>
<td></td>
<td>338</td>
</tr>
<tr>
<td>join/2</td>
<td></td>
<td>339</td>
</tr>
<tr>
<td>lait/1</td>
<td></td>
<td>342</td>
</tr>
<tr>
<td>last/1</td>
<td></td>
<td>342</td>
</tr>
<tr>
<td>len/1</td>
<td></td>
<td>338</td>
</tr>
<tr>
<td>liat/1</td>
<td></td>
<td>342</td>
</tr>
<tr>
<td>member/2</td>
<td></td>
<td>340</td>
</tr>
<tr>
<td>new/0</td>
<td></td>
<td>338</td>
</tr>
<tr>
<td>out/1</td>
<td></td>
<td>338</td>
</tr>
<tr>
<td>out_r/1</td>
<td></td>
<td>339</td>
</tr>
<tr>
<td>peek/1</td>
<td></td>
<td>341</td>
</tr>
<tr>
<td>peek_r/1</td>
<td></td>
<td>341</td>
</tr>
<tr>
<td>reverse/1</td>
<td></td>
<td>339</td>
</tr>
<tr>
<td>snoc/2</td>
<td></td>
<td>342</td>
</tr>
<tr>
<td>split/2</td>
<td></td>
<td>339</td>
</tr>
<tr>
<td>tail/1</td>
<td></td>
<td>341</td>
</tr>
<tr>
<td>to_list/1</td>
<td></td>
<td>339</td>
</tr>
<tr>
<td>range/1</td>
<td>sofs, 432</td>
<td></td>
</tr>
<tr>
<td>re</td>
<td></td>
<td></td>
</tr>
<tr>
<td>compile/1</td>
<td></td>
<td>346</td>
</tr>
<tr>
<td>compile/2</td>
<td></td>
<td>346</td>
</tr>
<tr>
<td>replace/3</td>
<td></td>
<td>353</td>
</tr>
<tr>
<td>replace/4</td>
<td></td>
<td>353</td>
</tr>
<tr>
<td>run/2</td>
<td></td>
<td>348</td>
</tr>
<tr>
<td>run/3</td>
<td></td>
<td>348</td>
</tr>
<tr>
<td>split/2</td>
<td></td>
<td>354</td>
</tr>
<tr>
<td>split/3</td>
<td></td>
<td>354</td>
</tr>
<tr>
<td>reachable/2</td>
<td>digraph_utils, 134</td>
<td></td>
</tr>
<tr>
<td>reachable_neighbours/2</td>
<td>digraph_utils, 135</td>
<td></td>
</tr>
<tr>
<td>reaching/2</td>
<td>digraph_utils, 135</td>
<td></td>
</tr>
<tr>
<td>reaching_neighbours/2</td>
<td>digraph_utils, 135</td>
<td></td>
</tr>
<tr>
<td>read/2</td>
<td>io, 250</td>
<td></td>
</tr>
<tr>
<td>read/3</td>
<td>io, 250</td>
<td></td>
</tr>
<tr>
<td>regexp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>first_match/2</td>
<td></td>
<td>392</td>
</tr>
<tr>
<td>format_error/1</td>
<td></td>
<td>394</td>
</tr>
<tr>
<td>gsub/3</td>
<td></td>
<td>393</td>
</tr>
<tr>
<td>match/2</td>
<td></td>
<td>392</td>
</tr>
<tr>
<td>matches/2</td>
<td></td>
<td>393</td>
</tr>
<tr>
<td>parse/1</td>
<td></td>
<td>394</td>
</tr>
<tr>
<td>sh_to_awk/1</td>
<td></td>
<td>394</td>
</tr>
<tr>
<td>split/2</td>
<td></td>
<td>393</td>
</tr>
<tr>
<td>sub/3</td>
<td></td>
<td>393</td>
</tr>
<tr>
<td>regs/0</td>
<td>c, 98</td>
<td></td>
</tr>
<tr>
<td>relation/2</td>
<td>sofs, 432</td>
<td></td>
</tr>
<tr>
<td>relation_to_family/1</td>
<td>sofs, 432</td>
<td></td>
</tr>
<tr>
<td>relative_product/2</td>
<td>sofs, 433</td>
<td></td>
</tr>
<tr>
<td>relative_product1/2</td>
<td>sofs, 433</td>
<td></td>
</tr>
<tr>
<td>relax/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Module/Function</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>array, 83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>relay/1 slave, 414</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove/2 sys, 457</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove/3 sys, 457</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rename/2 ets, 179</td>
<td></td>
<td></td>
</tr>
<tr>
<td>repair_continuation/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dets, 115 ets, 179</td>
<td></td>
<td></td>
</tr>
<tr>
<td>replace/3 re, 353</td>
<td></td>
<td></td>
</tr>
<tr>
<td>replace/4 re, 353</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reply/2 gen_fsm, 229</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gen_server, 240</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reserved_word/1 erl_scan, 156</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reset/1 array, 83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>resize/1 array, 84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>restart_child/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>supervisor, 448</td>
<td></td>
<td></td>
</tr>
<tr>
<td>restriction/2 sofs, 433</td>
<td></td>
<td></td>
</tr>
<tr>
<td>restriction/3 sofs, 434</td>
<td></td>
<td></td>
</tr>
<tr>
<td>results/1 shell, 409</td>
<td></td>
<td></td>
</tr>
<tr>
<td>resume/1 sys, 456</td>
<td></td>
<td></td>
</tr>
<tr>
<td>resume/2 sys, 456</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reverse/1 lists, 275</td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue, 339</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reverse/2 lists, 275</td>
<td></td>
<td></td>
</tr>
<tr>
<td>right/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string, 441</td>
<td></td>
<td></td>
</tr>
<tr>
<td>right/3 string, 441</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rootname/1 filename, 201</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rootname/2 filename, 201</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rows/1 io, 257</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rstr/2 string, 438</td>
<td></td>
<td></td>
</tr>
<tr>
<td>run/2 re, 348</td>
<td></td>
<td></td>
</tr>
<tr>
<td>run/3 re, 348</td>
<td></td>
<td></td>
</tr>
<tr>
<td>safe-fixtable/2 dets, 115 ets, 180</td>
<td></td>
<td></td>
</tr>
<tr>
<td>scan_erl_exprs/1 io, 257</td>
<td></td>
<td></td>
</tr>
<tr>
<td>scan_erl_exprs/3 io, 257</td>
<td></td>
<td></td>
</tr>
<tr>
<td>scan_erl_form/1 io, 257</td>
<td></td>
<td></td>
</tr>
<tr>
<td>scan_erl_form/3 io, 257</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seconds/1 timer, 463</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seconds_to_daytime/1 calendar, 102</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seconds_to_time/1 calendar, 102</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seed/0 random, 343</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seed/3 random, 343</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seed0/0 random, 343</td>
<td></td>
<td></td>
</tr>
<tr>
<td>select/1 dets, 116 ets, 183</td>
<td></td>
<td></td>
</tr>
<tr>
<td>select/2 dets, 116</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ets, 181
select/3
  dets, 116
  ets, 182
select_count/2
  ets, 183
select_delete/2
  dets, 117
  ets, 183
send/2
  lib, 264
  pg, 305
send_after/2
  timer, 461
send_after/3
  timer, 461
send_all_state_event/2
  gen_fsm, 228
send_event/2
  gen_fsm, 227
send_event_after/2
  gen_fsm, 229
send_interval/2
  timer, 462
send_interval/3
  timer, 462
send_op/2
  erl INTERNAL, 145
sendw/2
  lib, 265
seq/2
  lists, 275
seq/3
  lists, 275
set/1
  array, 84
set/2
  sofs, 434
set_attribute/3
  erl SCAN, 158
set_value/3
  win32reg, 471
setopts/2
  io, 248
sets
  add_element/2, 398
  del_element/2, 398
  filter/2, 399
  fold/3, 399
  from_list/1, 398
  intersection/1, 399
  intersection/2, 398
  is_disjoint/2, 399
  is_element/2, 398
  is_set/1, 397
  is_subset/2, 399
  new/0, 397
  size/1, 397
  subtract/2, 399
  to_list/1, 397
  union/1, 398
  union/2, 398
sh_to_awk/1
  regexp, 394
shell
  catch_exception/1, 409
  history/1, 409
  results/1, 409
  start_restricted/1, 409
  stop_restricted/0, 410
sin/1
  math, 283
singleton/1
  gb_sets, 208
sinh/1
  math, 283
size/1
  array, 84
  dict, 122
  gb_sets, 208
  gb_trees, 213
  orddict, 298
  ordsets, 301
  sets, 397
slave
  pseudo/1, 414
  pseudo/2, 414
  relay/1, 414
  start/1, 412
  start/2, 412
  start/3, 412
  start_link/1, 413
<table>
<thead>
<tr>
<th>Function</th>
<th>Module/Package</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_link/2</td>
<td></td>
<td>413</td>
</tr>
<tr>
<td>start_link/3</td>
<td></td>
<td>413</td>
</tr>
<tr>
<td>stop/1</td>
<td></td>
<td>414</td>
</tr>
<tr>
<td>sleep/1</td>
<td>timer</td>
<td>462</td>
</tr>
<tr>
<td>slot/2</td>
<td></td>
<td>417</td>
</tr>
<tr>
<td></td>
<td>dets</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>ets</td>
<td>184</td>
</tr>
<tr>
<td>smallest/1</td>
<td></td>
<td>208</td>
</tr>
<tr>
<td></td>
<td>gb_sets</td>
<td>208</td>
</tr>
<tr>
<td></td>
<td>gb_trees</td>
<td>213</td>
</tr>
<tr>
<td>snoc/2</td>
<td></td>
<td>342</td>
</tr>
<tr>
<td>sofs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a_function/2</td>
<td>419</td>
</tr>
<tr>
<td></td>
<td>canonical_relation/1</td>
<td>419</td>
</tr>
<tr>
<td></td>
<td>composite/2</td>
<td>419</td>
</tr>
<tr>
<td></td>
<td>constant_function/2</td>
<td>419</td>
</tr>
<tr>
<td></td>
<td>converse/1</td>
<td>420</td>
</tr>
<tr>
<td></td>
<td>difference/2</td>
<td>420</td>
</tr>
<tr>
<td></td>
<td>digraph_to_family/2</td>
<td>420</td>
</tr>
<tr>
<td></td>
<td>domain/1</td>
<td>420</td>
</tr>
<tr>
<td></td>
<td>drestriction/2</td>
<td>420</td>
</tr>
<tr>
<td></td>
<td>drestriction/3</td>
<td>421</td>
</tr>
<tr>
<td></td>
<td>empty_set/0</td>
<td>421</td>
</tr>
<tr>
<td></td>
<td>extension/3</td>
<td>421</td>
</tr>
<tr>
<td></td>
<td>family/2</td>
<td>422</td>
</tr>
<tr>
<td></td>
<td>family_difference/2</td>
<td>422</td>
</tr>
<tr>
<td></td>
<td>family_domain/1</td>
<td>422</td>
</tr>
<tr>
<td></td>
<td>family_field/1</td>
<td>422</td>
</tr>
<tr>
<td></td>
<td>family_intersection/1</td>
<td>423</td>
</tr>
<tr>
<td></td>
<td>family_intersection/2</td>
<td>423</td>
</tr>
<tr>
<td></td>
<td>family_projection/2</td>
<td>423</td>
</tr>
<tr>
<td></td>
<td>family_range/1</td>
<td>423</td>
</tr>
<tr>
<td></td>
<td>family_specification/2</td>
<td>424</td>
</tr>
<tr>
<td></td>
<td>family_to_digraph/2</td>
<td>424</td>
</tr>
<tr>
<td></td>
<td>family_to_relation/1</td>
<td>424</td>
</tr>
<tr>
<td></td>
<td>family_union/1</td>
<td>425</td>
</tr>
<tr>
<td></td>
<td>family_union/2</td>
<td>425</td>
</tr>
<tr>
<td></td>
<td>field/1</td>
<td>425</td>
</tr>
<tr>
<td></td>
<td>from_external/2</td>
<td>425</td>
</tr>
<tr>
<td></td>
<td>from_sets/1</td>
<td>425, 426</td>
</tr>
<tr>
<td></td>
<td>from_term/2</td>
<td>426</td>
</tr>
<tr>
<td></td>
<td>image/2</td>
<td>427</td>
</tr>
<tr>
<td></td>
<td>intersection/1</td>
<td>427</td>
</tr>
<tr>
<td></td>
<td>intersection/2</td>
<td>427</td>
</tr>
<tr>
<td></td>
<td>intersection_of_family/1</td>
<td>427</td>
</tr>
<tr>
<td></td>
<td>inverse/1</td>
<td>427</td>
</tr>
<tr>
<td></td>
<td>inverse_image/2</td>
<td>427</td>
</tr>
<tr>
<td></td>
<td>is_a_function/1</td>
<td>428</td>
</tr>
<tr>
<td></td>
<td>is_disjoint/2</td>
<td>428</td>
</tr>
<tr>
<td></td>
<td>is_empty_set/1</td>
<td>428</td>
</tr>
<tr>
<td></td>
<td>is_equal/2</td>
<td>428</td>
</tr>
<tr>
<td></td>
<td>is_set/1</td>
<td>428</td>
</tr>
<tr>
<td></td>
<td>is_soifs_set/1</td>
<td>429</td>
</tr>
<tr>
<td></td>
<td>is_subset/2</td>
<td>429</td>
</tr>
<tr>
<td></td>
<td>is_type/1</td>
<td>429</td>
</tr>
<tr>
<td></td>
<td>join/4</td>
<td>429</td>
</tr>
<tr>
<td></td>
<td>multiple_relative_product/2</td>
<td>429</td>
</tr>
<tr>
<td></td>
<td>no_elements/1</td>
<td>430</td>
</tr>
<tr>
<td></td>
<td>partition/1</td>
<td>430</td>
</tr>
<tr>
<td></td>
<td>partition/2</td>
<td>430</td>
</tr>
<tr>
<td></td>
<td>partition/3</td>
<td>430</td>
</tr>
<tr>
<td></td>
<td>partition_family/2</td>
<td>431</td>
</tr>
<tr>
<td></td>
<td>product/1</td>
<td>431</td>
</tr>
<tr>
<td></td>
<td>product/2</td>
<td>431</td>
</tr>
<tr>
<td></td>
<td>projection/2</td>
<td>432</td>
</tr>
<tr>
<td></td>
<td>range/1</td>
<td>432</td>
</tr>
<tr>
<td></td>
<td>relation/2</td>
<td>432</td>
</tr>
<tr>
<td></td>
<td>relation_to_family/1</td>
<td>432</td>
</tr>
<tr>
<td></td>
<td>relative_product/2</td>
<td>433</td>
</tr>
<tr>
<td></td>
<td>relative_product1/2</td>
<td>433</td>
</tr>
<tr>
<td></td>
<td>restriction/2</td>
<td>433</td>
</tr>
<tr>
<td></td>
<td>restriction/3</td>
<td>434</td>
</tr>
<tr>
<td></td>
<td>set/2</td>
<td>434</td>
</tr>
<tr>
<td></td>
<td>specification/2</td>
<td>434</td>
</tr>
<tr>
<td></td>
<td>strict_relation/1</td>
<td>434</td>
</tr>
<tr>
<td></td>
<td>substitution/2</td>
<td>435</td>
</tr>
<tr>
<td></td>
<td>symmetric_partition/2</td>
<td>436</td>
</tr>
<tr>
<td></td>
<td>to_external/1</td>
<td>436</td>
</tr>
<tr>
<td></td>
<td>to_sets/1</td>
<td>436</td>
</tr>
<tr>
<td></td>
<td>type/1</td>
<td>436</td>
</tr>
<tr>
<td></td>
<td>union/1</td>
<td>436</td>
</tr>
<tr>
<td></td>
<td>union/2</td>
<td>436</td>
</tr>
<tr>
<td></td>
<td>union_of_family/1</td>
<td>437</td>
</tr>
<tr>
<td></td>
<td>weak_relation/1</td>
<td>437</td>
</tr>
<tr>
<td>sort/1</td>
<td>file_sorter</td>
<td>193</td>
</tr>
<tr>
<td></td>
<td>lists</td>
<td>276</td>
</tr>
<tr>
<td>sort/2</td>
<td>file_sorter</td>
<td>193</td>
</tr>
<tr>
<td></td>
<td>lists</td>
<td>276</td>
</tr>
<tr>
<td></td>
<td>qlc</td>
<td>333</td>
</tr>
<tr>
<td>sort/3</td>
<td>file_sorter</td>
<td>193</td>
</tr>
<tr>
<td>span/2</td>
<td>string</td>
<td>439</td>
</tr>
<tr>
<td>sparse_foldl/2</td>
<td>array</td>
<td>84</td>
</tr>
</tbody>
</table>
Index of Modules and Functions

sparse_foldr/2
   array, 84
sparse_map/2
   array, 85
sparse_size/1
   array, 85
sparse_to_list/1
   array, 85
sparse_to_orddict/1
   array, 85
spawn/1
   proc_lib, 308
spawn/2
   proc_lib, 308
spawn/3
   proc_lib, 308
spawn/4
   proc_lib, 308
spawn_link/1
   proc_lib, 308
spawn_link/2
   proc_lib, 308
spawn_link/3
   proc_lib, 309
spawn_link/4
   proc_lib, 309
spawn_opt/2
   proc_lib, 309
spawn_opt/3
   proc_lib, 309
spawn_opt/4
   proc_lib, 309
spawn_opt/5
   proc_lib, 309
specification/2
   sofs, 434
split/1
   filename, 202
split/2
   lists, 276
   proplists, 316
   queue, 339
   re, 354
   regexp, 393
split/3
   re, 354
splitwith/2
   lists, 276
sqrt/1
   math, 283
start/0
   gen_event, 216
   timer, 461
start/1
   gen_event, 216
   pool, 306
   slave, 412
start/2
   pool, 306
   slave, 412
start/3
   gen_fsm, 227
   gen_server, 238
   proc_lib, 309
   slave, 412
start/4
   gen_fsm, 227
   gen_server, 238
   proc_lib, 309
start/5
   proc_lib, 309
start_child/2
   supervisor, 447
start_link/0
   gen_event, 216
start_link/1
   gen_event, 216
   slave, 413
start_link/2
   slave, 413
   supervisor, 446
   supervisor_bridge, 451
start_link/3
   gen_fsm, 226
   gen_server, 237
   proc_lib, 309
   slave, 413
   supervisor, 446
   supervisor_bridge, 451
start_link/4
Index of Modules and Functions

gen_fsm, 226
  gen_server, 237
  proc_lib, 309
start_link/5
  proc_lib, 309
start_restricted/1
  shell, 409
start_timer/2
  gen_fsm, 229
statistics/2
  sys, 455
statistics/3
  sys, 455
stop/0
  pool, 307
stop/1
  gen_event, 221
  slave, 414
stop_restricted/0
  shell, 410
store/3
  dict, 122
  orddict, 298
str/2
  string, 438
strict_relation/1
  sofs, 434
string
  centre/2, 441
  centre/3, 441
  chars/2, 440
  chars/3, 440
  chr/2, 438
  concat/2, 438
  copies/2, 440
  cspan/2, 439
  equal/2, 438
  join/2, 439
  left/2, 441
  left/3, 441
  len/1, 438
  rchr/2, 438
  right/2, 441
  right/3, 441
  rstr/2, 438
  span/2, 439
  str/2, 438
strip/1, 441
strip/2, 441
strip/3, 441
  sub_string/2, 442
  sub_string/3, 442
  sub_word/2, 440
  sub_word/3, 440
  substr/2, 439
  substr/3, 439
  to_float/1, 442
  to_integer/1, 442
  to_lower/1, 443
  to_upper/1, 443
  tokens/2, 439
  words/1, 440
  words/2, 440
string/1
  erl_scan, 154
string/2
  erl_scan, 154
string/3
  erl_scan, 154
string_to_handle/3
  qlc, 333
strip/1
  beam_lib, 92
  string, 441
strip/2
  string, 441
strip/3
  string, 441
strip_files/1
  beam_lib, 92
strip_release/1
  beam_lib, 93
strong_components/1
  digraph Utils, 135
sub/3
  regexp, 393
sub_keys/1
  win32reg, 472
sub_string/2
  string, 442
sub_string/3
  string, 442
sub_word/2

508  STD LIB
<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>string, 440</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub_word/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string, 440</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subgraph/3</td>
<td>digraph_utils, 135</td>
<td></td>
</tr>
<tr>
<td>sublist/2</td>
<td>lists, 277</td>
<td></td>
</tr>
<tr>
<td>sublist/3</td>
<td>lists, 277</td>
<td></td>
</tr>
<tr>
<td>substitute_aliases/2</td>
<td>proplists, 317</td>
<td></td>
</tr>
<tr>
<td>substitute_negations/2</td>
<td>proplists, 317</td>
<td></td>
</tr>
<tr>
<td>substitution/2</td>
<td>sofs, 435</td>
<td></td>
</tr>
<tr>
<td>substr/2</td>
<td>string, 439</td>
<td></td>
</tr>
<tr>
<td>substr/3</td>
<td>string, 439</td>
<td></td>
</tr>
<tr>
<td>subtract/2</td>
<td>gb_sets, 206</td>
<td>lists, 277</td>
</tr>
<tr>
<td></td>
<td>ordsets, 303</td>
<td>sets, 399</td>
</tr>
<tr>
<td>suffix/2</td>
<td>lists, 278</td>
<td></td>
</tr>
<tr>
<td>sum/1</td>
<td>lists, 278</td>
<td></td>
</tr>
<tr>
<td>supervisor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>check_childspecs/1, 449</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete_child/2, 448</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Module: init/1, 450</td>
<td></td>
<td></td>
</tr>
<tr>
<td>restart_child/2, 448</td>
<td></td>
<td></td>
</tr>
<tr>
<td>start_child/2, 447</td>
<td></td>
<td></td>
</tr>
<tr>
<td>start_link/2, 446</td>
<td></td>
<td></td>
</tr>
<tr>
<td>start_link/3, 446</td>
<td></td>
<td></td>
</tr>
<tr>
<td>terminate_child/2, 447</td>
<td></td>
<td></td>
</tr>
<tr>
<td>which_children/1, 449</td>
<td></td>
<td></td>
</tr>
<tr>
<td>supervisor_bridge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Module: init/1, 452</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Module: terminate/2, 452</td>
<td></td>
<td></td>
</tr>
<tr>
<td>start_link/2, 451</td>
<td></td>
<td></td>
</tr>
<tr>
<td>start_link/3, 451</td>
<td></td>
<td></td>
</tr>
<tr>
<td>suspend/1</td>
<td>sys, 456</td>
<td></td>
</tr>
<tr>
<td>suspend/2</td>
<td>sys, 456</td>
<td></td>
</tr>
<tr>
<td>swap_handler/5</td>
<td>gen_event, 219</td>
<td></td>
</tr>
<tr>
<td>swap_sup_handler/5</td>
<td>gen_event, 220</td>
<td></td>
</tr>
<tr>
<td>symdiff/2</td>
<td>sofs, 435</td>
<td></td>
</tr>
<tr>
<td>symmetric_partition/2</td>
<td>sofs, 436</td>
<td></td>
</tr>
<tr>
<td>sync/1</td>
<td>dets, 117</td>
<td></td>
</tr>
<tr>
<td>sync_notify/2</td>
<td>gen_event, 218</td>
<td></td>
</tr>
<tr>
<td>sync_send_all_state_event/2</td>
<td>gen_fsm, 228</td>
<td></td>
</tr>
<tr>
<td>sync_send_all_state_event/3</td>
<td>gen_fsm, 228</td>
<td></td>
</tr>
<tr>
<td>sync_send_event/2</td>
<td>gen_fsm, 228</td>
<td></td>
</tr>
<tr>
<td>sync_send_event/3</td>
<td>gen_fsm, 228</td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td></td>
<td></td>
</tr>
<tr>
<td>change_code/4, 456</td>
<td></td>
<td></td>
</tr>
<tr>
<td>change_code/5, 456</td>
<td></td>
<td></td>
</tr>
<tr>
<td>debug_options/1, 458</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get_debug/3, 458</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get_status/1, 456</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get_status/2, 456</td>
<td></td>
<td></td>
</tr>
<tr>
<td>handle_debug/1, 458</td>
<td></td>
<td></td>
</tr>
<tr>
<td>handle_system_msg/6, 458</td>
<td></td>
<td></td>
</tr>
<tr>
<td>install/3, 457</td>
<td></td>
<td></td>
</tr>
<tr>
<td>install/4, 457</td>
<td></td>
<td></td>
</tr>
<tr>
<td>log/2, 455</td>
<td></td>
<td></td>
</tr>
<tr>
<td>log/3, 455</td>
<td></td>
<td></td>
</tr>
<tr>
<td>log_to_file/2, 455</td>
<td></td>
<td></td>
</tr>
<tr>
<td>log_to_file/3, 455</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mod: system_code_change/4, 459</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mod: system_contiue/3, 459</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mod: system_terminate/4, 459</td>
<td></td>
<td></td>
</tr>
<tr>
<td>no_debug/1, 456</td>
<td></td>
<td></td>
</tr>
<tr>
<td>no_debug/2, 456</td>
<td></td>
<td></td>
</tr>
<tr>
<td>print_log/1, 459</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove/2, 457</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove/3, 457</td>
<td></td>
<td></td>
</tr>
<tr>
<td>resume/1, 456</td>
<td></td>
<td></td>
</tr>
<tr>
<td>resume/2, 456</td>
<td></td>
<td></td>
</tr>
<tr>
<td>statistics/2, 455</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Module</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------</td>
<td>------</td>
</tr>
<tr>
<td>time_difference/2</td>
<td>calendar</td>
<td>102</td>
</tr>
<tr>
<td>time_to_seconds/1</td>
<td>calendar</td>
<td>102</td>
</tr>
<tr>
<td>timer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>apply_after/4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>apply_interval/4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cancel/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exit_after/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exit_after/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hours/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kill_after/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kill_after/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>minutes/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>now_diff/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seconds/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>send_after/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>send_after/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>send_interval/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>send_interval/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sleep/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>start/0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tc/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>to_dets/2</td>
<td>erl_tar, zip</td>
<td></td>
</tr>
<tr>
<td></td>
<td>164, 477</td>
<td></td>
</tr>
<tr>
<td>to_ets/2</td>
<td>erl_tar, ets, zip</td>
<td></td>
</tr>
<tr>
<td></td>
<td>164, 184, 476</td>
<td></td>
</tr>
<tr>
<td>to_external/1</td>
<td>sofs</td>
<td>436</td>
</tr>
<tr>
<td>to_float/1</td>
<td>string</td>
<td>442</td>
</tr>
<tr>
<td>to_integer/1</td>
<td>string</td>
<td>442</td>
</tr>
<tr>
<td>to_list/1</td>
<td>array, dict</td>
<td>85, 123</td>
</tr>
<tr>
<td></td>
<td>gb_sets, gb_trees</td>
<td>209, 213</td>
</tr>
<tr>
<td></td>
<td>orddict</td>
<td>299</td>
</tr>
<tr>
<td></td>
<td>ordsets</td>
<td>301</td>
</tr>
<tr>
<td></td>
<td>queue</td>
<td>339</td>
</tr>
<tr>
<td></td>
<td>sets</td>
<td>397</td>
</tr>
<tr>
<td>to_lower/1</td>
<td>string</td>
<td>443</td>
</tr>
<tr>
<td>to_orddict/1</td>
<td>array</td>
<td>85</td>
</tr>
<tr>
<td>to_sets/1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
sofs, 436

to_upper/1
  string, 443

token_info/1
  erl_scan, 156

token_info/2
  erl_scan, 156

tokens/1
  erl_parse, 149

tokens/2
  erl_parse, 149
  string, 439

tokens/3
  erl_scan, 155

tokens/4
  erl_scan, 155

topsort/1
  digraph_utils, 136

trace/2
  sys, 456

trace/3
  sys, 456

transform_from_shell/3
  ms_transform, 294

translate_initial_call/1
  proc_lib, 311

traverse/2
  dets, 119

tt/1
  erl_tar, 165
  zip, 477

type/1
  sofs, 436

type_test/2
  erl_internal, 144

ukeymerge/3
  lists, 278

ukeysort/2
  lists, 278

umerge/1
  lists, 279

umerge/2
  lists, 279

umerge3/3
  lists, 279

unfold/1
  proplists, 317

unicode
  bom_to_encoding/1, 1466
  characters_to_binary/1, 1467
  characters_to_binary/2, 1468
  characters_to_binary/3, 1468
  characters_to_list/1, 1466
  characters_to_list/2, 1466
  encoding_to_bom/1, 1468

uniform/0
  random, 343

uniform/1
  random, 343

uniform_a/1
  random, 344

uniform_a/2
  random, 344

union/1
  gb_sets, 209
  ordsets, 302
  sets, 398
  sofs, 436

union/2
  gb_sets, 209
  ordsets, 302
  sets, 398
  sofs, 436

union_of_family/1
  sofs, 437

universal_time/0
  calendar, 102

universal_time_to_local_time/2
  calendar, 103

unzip/1
  lists, 279
  zip, 475

unzip/2
  zip, 475

unzip3/1
  lists, 280
Index of Modules and Functions

update/3
  dict, 123
gb_trees, 214
orddict, 299

update/4
  dict, 123
orddict, 299

update_counter/3
  dets, 119
dict, 123
ets, 188
orddict, 299

update_element/4
  ets, 188

usort/1
  lists, 280

usort/2
  lists, 280

valid_date/1
  calendar, 103
valid_date/3
  calendar, 103

value/2
  win32reg, 472

values/1
  gb_trees, 214
win32reg, 472

version/1
  beam_lib, 90

vertex/2
  digraph, 131

vertices/1
  digraph, 131

weak_relation/1
  sofs, 437

which_children/1
  supervisor, 449

which_handlers/1
  gen_event, 220

wildcard/1
  filelib, 196

wildcard/2
  filelib, 197

win32reg
  change_key/2, 470
  change_key_create/2, 470
close/1, 470
current_key/1, 470
delete_key/1, 470
delete_value/2, 471
expand/1, 471
format_error/1, 471
open/1, 471
set_value/3, 471
sub_keys/1, 472
value/2, 472
values/1, 472

words/1
  string, 440

words/2
  string, 440

write/1
  io_lib, 260

write/2
  io, 250
  io_lib, 260

write_atom/1
  io_lib, 262

write_char/1
  io_lib, 262

write_string/1
  io_lib, 262

xm/1
  c, 98

y/1
  c, 98

y/2
  c, 98

zip
  create/2, 474
  create/3, 474
  extract/1, 475
  extract/2, 475
  list_dir/1, 476
  list_dir/2, 476
t/1, 477
table/1, 476
table/2, 476
Index of Modules and Functions

tt/1, 477
unzip/1, 475
unzip/2, 475
zip/2, 474
zip/3, 474
zip_close/1, 478
zip_get/1, 478
zip_get/2, 478
zip_list_dir/1, 477
zip_open/1, 477
zip_open/2, 477
zip/2
   lists, 280
   zip, 474
zip/3
   zip, 474
zip3/3
   lists, 280
zip_close/1
   zip, 478
zip_get/1
   zip, 478
zip_get/2
   zip, 478
zip_list_dir/1
   zip, 477
zip_open/1
   zip, 477
zip_open/2
   zip, 477
zip_with/3
   lists, 281
zip_with3/4
   lists, 281