1.1 The Erlang I/O-protocol

The I/O-protocol in Erlang specifies a way for a client to communicate with an I/O server and vice versa. The I/O server is a process that handles the requests and performs the requested task on e.g. an IO device. The client is any Erlang process wishing to read or write data from/to the IO 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 Virding’s 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 hold. The I/O server and client communicate with one single, rather simplistic protocol and no server state is ever present in the client. Any I/O server can be used together with any client code and client code need not be aware of the actual IO device the I/O server communicates with.

1.1.1 Protocol Basics

As described in Robert’s paper, I/O servers and clients communicate using io_request/io_reply tuples as follows:

\[
\text{io_request, From, ReplyAs, Request} \]
\[
\text{io_reply, ReplyAs, Reply} \]

The client sends an io_request tuple to the I/O server and the server eventually sends a corresponding io_reply tuple.

- **From** is the \( \text{pid()} \) of the client, the process which the I/O server sends the IO reply to.
- **ReplyAs** can be any datum and is returned in the corresponding io_reply. The *io* module monitors the I/O server, and uses the monitor reference as the ReplyAs datum. A more complicated client could have several outstanding I/O requests to the same I/O server and would then use different references (or something else) to differentiate among the incoming IO replies. The ReplyAs element should be considered opaque by the I/O server. Note that the \( \text{pid()} \) of the I/O server is not explicitly present in the io_reply tuple. The reply can be sent from any process, not necessarily the actual I/O server. The ReplyAs element is the only thing that connects one I/O request with an I/O-reply.
- **Request** and **Reply** are described below.

When an I/O server receives an io_request tuple, it acts upon the actual Request part and eventually sends an io_reply tuple with the corresponding Reply part.

1.1.2 Output Requests

To output characters on an IO device, the following Requests exist:

\[
\text{put_chars, Encoding, Characters} \]
The Erlang I/O-protocol

{put_chars, Encoding, Module, Function, Args}

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

- **Characters** are the data to be put on the IO 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**, and **Args** denote a function which will be called to produce the data (like io_lib:format/2). Args is a list of arguments to the function. The function should produce data in the given Encoding. The I/O server should call the function as apply(Mod, Func, Args) and will put the returned data on the IO 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 I/O server replies to the client with an io_reply tuple 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 I/O server (these requests 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 an IO device, the following Requests exist:

{get_until, Encoding, Prompt, Module, Function, 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/ExtraArgs. 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 is 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 IO device. Prompt is often ignored by the I/O server and if set to '' it should always be ignored (and result in nothing being written to the IO device).

- **Module**, **Function**, and **ExtraArgs** denote 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:
1.1 The Erlang I/O-protocol

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

The function will be called with the data the I/O server finds on its IO device, returning \{done, Result, RestChars\} when enough data is read (in which case Result is sent to the client and RestChars is kept in the I/O 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 IO device is the atom \(\text{eof}\). An emulation of the \texttt{get_line} request could be (inefficiently) implemented using the following functions:

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

until_newline(ThisFar, eof, _MyStopCharacter) ->
    \{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, [\$\n]\}}
    receive
        \{io_reply, IoServer, Data} ->
            Data
    end.
```

Note especially that the last element in the Request tuple ([\$\n]) is appended to the argument list when the function is called. The function should be called like \texttt{apply(Module, Function, [ State, Data | ExtraArgs ])} by the I/O server.

A fixed 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 IO device.

A single line (like in the example above) is requested with this Request:

\{get_line, Encoding, Prompt\}

- Encoding and Prompt as above.

Obviously, the \texttt{get_chars} and \texttt{get_line} could be implemented with the \texttt{get_until} request (and indeed they were originally), but demands for efficiency has made these additions necessary.
The I/O server replies to the client with an io_reply tuple 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 I/O 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 I/O server (these requests 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 I/O server has not only lead to the addition of the get_line and get_chars requests, but has also added the concept of I/O server options. No options are mandatory to implement, but all I/O servers in the Erlang standard libraries honor the binary option, which allows the Data element of the io_reply tuple to be a binary instead of a list 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 of the get_until request still gets list data regardless of the I/O 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 I/O server). The client has to be prepared for data received as answers to those requests to be in a variety of forms, but the I/O 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.

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 I/O 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 I/O server itself).

As an example, the I/O server for the interactive shell (in group.erl) understands the following options:

- {binary, boolean()} (or binary/list)
- {echo, boolean()}
- {expand_fun, fun()}
- {encoding, unicode/latin1} (or unicondeflatin1)

- of which the binary and encoding options are common for all I/O servers in OTP, while echo and expand are valid only for this I/O server. It is worth noting that the unicode option notifies how characters are actually put on the physical IO 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 I/O server should send one of the following as Reply:

- ok
- {error, Error}
1.1 The Erlang I/O-protocol

An error (preferably \texttt{enotsup}) is to be expected if the option is not supported by the I/O server (like if an \texttt{echo} option is sent in a \texttt{setopts} request to a plain file).

To retrieve options, this request is used:

\begin{verbatim}
getopts
\end{verbatim}

The \texttt{getopts} request asks for a complete list of all options supported by the I/O server as well as their current values.

The I/O server replies:

\begin{verbatim}
OptList
\end{verbatim}

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

1.1.5 Multiple I/O Requests

The \texttt{Request} element can in itself contain several \texttt{Requests} by using the following format:

\begin{verbatim}
{requests, Requests}
\end{verbatim}

- \texttt{Requests} is a list of valid \texttt{io_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 I/O server can for a list of requests send any of the valid results in the reply:

\begin{verbatim}
ok
{ok, Data}
{ok, Options}
{error, Error}
\end{verbatim}

- 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:

\begin{verbatim}
{get_geometry, Geometry}
\end{verbatim}

- \texttt{Geometry} is either the atom \texttt{rows} or the atom \texttt{columns}.

The I/O server should send the \texttt{Reply} as:

\begin{verbatim}
{ok, N}
{error, Error}
\end{verbatim}

- \texttt{N} is the number of character rows or columns the IO device has, if applicable to the IO device the I/O server handles, otherwise \{error, \texttt{enotsup}\} is a good answer.

1.1.7 Unimplemented Request Types

If an I/O server encounters a request it does not recognize (i.e. the \texttt{io_request} tuple is in the expected format, but the actual \texttt{Request} is unknown), the I/O server should send a valid reply with the error tuple:

\begin{verbatim}
{error, request}
\end{verbatim}

This makes it possible to extend the protocol with optional requests and for the clients to be somewhat backwards compatible.
1.1.8 An Annotated and Working Example I/O Server

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

Our example I/O 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 I/O server and a main loop handling the requests:

```erlang
-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, mode}).
start_link() ->
    spawn_link(?MODULE,init,[]).
init() ->
    Table = ets:new(noname,[ordered_set]),
    ?MODULE:loop(#state{table = Table, position = 0, mode=list}).
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.
reply(From, ReplyAs, Reply) ->
```

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 IO 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 us look at the reply function first...
1.1 The Erlang I/O-protocol

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:

```erlang
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
        _:_ ->
            {error, {error, Function}, State}
    end;
```

The `Encoding` tells us how the characters in the request 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 `Encoding` 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 us 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`, let us look into the few remaining requests:

```erlang
request({get_geometry, _}, 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 I/O 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 is 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 I/O server will not work with a simple file:write/2 if these are not added:

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

```
request(_Other, State) ->
    {error, {error, request}, State}.
```

Let us move further and actually handle the different requests, first the fairly generic multi-request type:

```
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 is first returned to the main loop and then sent back by the function io_reply).

The getopts and setopts requests are also simple to handle, we just change or read our state record:

```
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#state{mode=binary}};
                _ ->
                    {ok,ok,State}
            end;
        false ->
            {error,{error,enotsup},State}
    end.
check_valid_opts([]) ->
    true;
```
1.1 The Erlang I/O-protocol

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 I/O servers handle both \{setopts, [binary]\}, \{setopts, [list]\} and \{setopts,[[binary, boolean()]]\}, 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.

The getopts 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 I/O server. We have only one option, and hence return that.

So far our I/O server has been fairly generic (except for the \texttt{rewind} request handled in the main loop and the creation of an ETS table). Most I/O servers contain code similar to the one 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 \texttt{put_chars/3} 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 \texttt{split_data/3} and \texttt{apply_update/2} are implemented below.

Now we want to read data from the table. The \texttt{get_until/5} function reads data and applies the function until it says it is 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;
    end;

10 | Ericsson AB. All Rights Reserved.: STDLIB
1.1 The Erlang I/O-protocol

```erl
{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} ->
      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 I/O servers send data back to the client as lists, but switching mode to binary might increase efficiency if the I/O 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 tuple may, as its 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:

```erl
check(unicode, List) -> List;
check(latin1, List) ->
  try
    [ throw(not_unicode) || 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/5 function to implement get_chars and get_line (inefficiently):

```erl
until_newline([],eof,_MyStopCharacter) ->
  {done,eof,[]};
until_newline(ThisFar,eof,_MyStopCharacter) ->
  {done,ThisFar,[]};
until_newline(ThisFar,CharList,MyStopCharacter) ->
  case lists:splithwith(fun(X) -> X =/= MyStopCharacter end, CharList)
1.1 The Erlang I/O-protocol

```
{L,[]} -> {more,ThisFar++L};
{L2,[MyStopCharacter|Rest]} -> {done,ThisFar++L2++[MyStopCharacter],Rest}
end.

until_enough([],eof,_N) ->
  {done,eof,[]};
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 I/O 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
  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(Char, Row, Col) ->
  {This,Left} = my_split(?CHARS_PER_REC - Col, Char, []),
  {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:reverse(Char,0) ++ List});
    [{Row, OldData}] ->
      {Part1,} = my_split(Char,OldData,[]),
      {Part2} = my_split(Char+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 I/O server by using i.e. the \texttt{io} module or even the \texttt{file} module. It is as simple as that to implement a fully fledged I/O server in Erlang.

## 1.2 Using Unicode in Erlang

### 1.2.1 Unicode Implementation

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

The functionality described in EEP10 was implemented in Erlang/OTP R13A, but that was by no means the end of it. In Erlang/OTP R14B01 support for Unicode file names was added, although it was in no way complete and was by default disabled on platforms where no guarantee was given for the file name encoding. With Erlang/OTP R16A came support for UTF-8 encoded source code, among with enhancements to many of the applications to support both Unicode encoded file names as well as support for UTF-8 encoded files in several circumstances. Most notable is the support for UTF-8 in files read by \texttt{file:consult/1}, release handler support for UTF-8 and more support for Unicode character sets in the I/O-system. In Erlang/OTP 17.0, the encoding default for Erlang source files was switched to UTF-8.

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

### 1.2.2 Understanding Unicode

Experience with the Unicode support in Erlang has made it painfully clear that understanding Unicode characters and encodings is not as easy as one would expect. The complexity of the field as well as the implications of the standard requires thorough understanding of concepts rarely before thought of.

Furthermore the Erlang implementation requires understanding of concepts that never were an issue for many (Erlang) programmers. To understand and use Unicode characters requires that you study the subject thoroughly, even if you're an experienced programmer.

As an example, one could contemplate the issue of converting between upper and lower case letters. Reading the standard will make you realize that, to begin with, there's not a simple one to one mapping in all scripts. Take German as an example, where there's a letter "ß" (Sharp s) in lower case, but the uppercase equivalent is "SS". Or Greek, where "ß" has two different lowercase forms: "ß" in word-final position and "ß" elsewhere. Or Turkish where dotted and dot-less "ı" both exist in lower case and upper case forms, or Cyrillic "I" which usually has no lowercase form. Or of course languages that have no concept of upper case (or lower case). So, a conversion function will need to know not only one character at a time, but possibly the whole sentence, maybe the natural language the translation should be in and also take into account differences in input and output string length and so on. There is at the time of writing no Unicode \texttt{to_upper/to_lower} functionality in Erlang/OTP, but there are publicly available libraries that address these issues.

Another example is the accented characters where the same glyph has two different representations. Let's look at the Swedish "ö". There's a code point for that in the Unicode standard, but you can also write it as "ö" followed by \texttt{U+0308} (Combining Diaeresis, with the simplified meaning that the last letter should have a "¨" above). They have exactly the same glyph. They are for most purposes the same, but they have completely different representations. For example MacOS X converts all file names to use Combining Diaeresis, while most other programs (including Erlang) try to hide that by doing the opposite when for example listing directories. However it's done, it's usually important to normalize such characters to avoid utter confusion.

The list of examples can be made as long as the Unicode standard, I suspect. The point is that one need a kind of knowledge that was never needed when programs only took one or two languages into account. The complexity of human languages and scripts, certainly has made this a challenge when constructing a universal standard. Supporting Unicode properly in your program \texttt{will} require effort.
1.2.3 What Unicode Is

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

Unicode code points 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 code points for all the scripts, there are a couple of encoding standards available.

It is vital to understand the difference between encodings and Unicode characters. Unicode characters are code points according to the Unicode standard, while the encodings are ways to represent such code points. An encoding is just a standard for representation, UTF-8 can for example be used to represent a very limited part of the Unicode character set (e.g. ISO-Latin-1), or the full Unicode range. It's just an encoding format.

As long as all character sets were limited to 256 characters, each character could be stored in one single byte, so there was more or less only one practical encoding for the characters. Encoding each character in one byte was so common that the encoding wasn't even named. When we now, with the Unicode system, have a lot more than 256 characters, we need a common way to represent these. The common ways of representing the code points are the encodings. This means a whole new concept to the programmer, the concept of character representation, which was before a non-issue.

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:

Bytewise representation

This is not a proper Unicode representation, but the representation used for characters before the Unicode standard. It can still be used to represent character code points in the Unicode standard that have numbers below 256, which corresponds exactly to the ISO-Latin-1 character set. In Erlang, this is commonly denoted latin1 encoding, which is slightly misleading as ISO-Latin-1 is a character code range, not an encoding.

UTF-8

Each character is stored in one to four bytes depending on code point. The encoding is backwards compatible with bytewise representation of 7-bit ASCII as all 7-bit characters are stored in one single byte in UTF-8. The characters beyond code point 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. Note that UTF-8 is not compatible with bytewise representation for code points between 128 and 255, so a ISO-Latin-1 bytewise representation is not generally compatible with UTF-8.

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, libraries 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. In Erlang, the full UTF-16 range is supported when applicable, like in the unicode module and in the bit syntax.

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 code points 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.
1.2 Using Unicode in Erlang

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 numbers are left unused in the Unicode standard and certain ranges are even deemed invalid. The most notable invalid range is 16#D800 - 16#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 hole in the Unicode range to cope with backward compatibility.

Additionally, the code point 16#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, but their use might become more widely spread as they provide the means for programs to make educated guesses about the Unicode format of a certain file.

1.2.4 Areas of Unicode Support

To support Unicode in Erlang, problems in several areas have been addressed. Each area is described briefly in this section and more thoroughly further down in this document:

Representation

To handle Unicode characters in Erlang, we have to have a common representation both in lists and binaries. The EEP (10) and the subsequent initial implementation in Erlang/OTP R13A settled a standard representation of Unicode characters in Erlang.

Manipulation

The Unicode characters need to be processed by the Erlang program, why library functions need to be able to handle them. In some cases functionality was added to already existing interfaces (as the string module now can handle lists with arbitrary code points), in some cases new functionality or options need to be added (as in the io-module, the file handling, the unicode module and the bit syntax). Today most modules in kernel and STDLIB, as well as the VM are Unicode aware.

File I/O

I/O is by far the most problematic area for Unicode. A file is an entity where bytes are stored and the lore of programming has been to treat characters and bytes as interchangeable. With Unicode characters, you need to decide on an encoding as soon as you want to store the data in a file. In Erlang you can open a text file with an encoding option, so that you can read characters from it rather than bytes, but you can also open a file for bytewise I/O. The I/O-system of Erlang has been designed (or at least used) in a way where you expect any I/O-server to be able to cope with any string data, but that is no longer the case when you work with Unicode characters. Handling the fact that you need to know the capabilities of the device where your data ends up is something new to the Erlang programmer. Furthermore, ports in Erlang are byte oriented, so an arbitrary string of (Unicode) characters can not be sent to a port without first converting it to an encoding of choice.

Terminal I/O

Terminal I/O is slightly easier than file I/O. The output is meant for human reading and is usually Erlang syntax (e.g. in the shell). There exists syntactic representation of any Unicode character without actually displaying the glyph (instead written as \x{HHH}), so Unicode data can usually be displayed even if the terminal as such do not support the whole Unicode range.

File names

File names can be stored as Unicode strings, in different ways depending on the underlying OS and file system. This can be handled fairly easy by a program. The problems arise when the file system is not consistent in it’s encodings, like for example Linux. Linux allows files to be named with any sequence of bytes, leaving to each program to interpret those bytes. On systems where these “transparent” file names are used, Erlang has to be informed about the file name encoding by a startup flag. The default is byte-wise interpretation, which is actually usually wrong, but allows for interpretation of all file names. The concept of “raw file names” can be
used to handle wrongly encoded file names if one enables Unicode file name translation (+fnu) on platforms
where this is not the default.

Source code encoding
When it comes to the Erlang source code, there is support for the UTF-8 encoding and bytewise encoding. The
default in Erlang/OTP R16B was bytewise (or latin1) encoding; in Erlang/OTP 17.0 it was changed to UTF-8.
You can control the encoding by a comment like:

```
%% -*- coding: utf-8 -*-
```

in the beginning of the file. This of course requires your editor to support UTF-8 as well. The same comment
is also interpreted by functions like file:consult/1, the release handler etc, so that you can have all text
files in your source directories in UTF-8 encoding.

The language
Having the source code in UTF-8 also allows you to write string literals containing Unicode characters with
code points > 255, although atoms, module names and function names are restricted to the ISO-Latin-1 range.
Binary literals where you use the /utf8 type, can also be expressed using Unicode characters > 255. Having
module names using characters other than 7-bit ASCII can cause trouble on operating systems with inconsistent
file naming schemes, and might also hurt portability, so it's not really recommended. It is suggested in EEP 40
that the language should also allow for Unicode characters > 255 in variable names. Whether to implement that
EEP or not is yet to be decided.

1.2.5 Standard Unicode Representation

In Erlang, strings are actually lists of integers. A string was up until Erlang/OTP R13 defined to be encoded in the
ISO-latin-1 (ISO8859-1) character set, which is, code point by code point, a sub-range of the Unicode character set.
The standard list encoding for strings was therefore easily extended to cope with the whole Unicode range: A Unicode
string in Erlang is simply a list containing integers, each integer being a valid Unicode code point and representing
one character in the Unicode character set.

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

Only if a string contains code points < 256, can it be directly converted to a binary by using i.e. erlang:iolist_to_binary/1 or can be sent directly to a port. If the string contains Unicode characters > 255, an encoding has to be decided upon and the string should be converted to a binary in the preferred encoding using unicode:characters_to_binary/1,2,3. Strings are not generally lists of bytes, as they were before
Erlang/OTP R13. They are lists of characters. Characters are not generally bytes, they are Unicode code points.

Binaries 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 ISO-Latin-1 Erlang string could be converted into a binary,
effectively using bytewise encoding - one byte per character. This was very convenient for those limited Erlang strings,
but cannot be done for arbitrary Unicode lists.

As the UTF-8 encoding is widely spread and provides some backward compatibility in the 7-bit ASCII range, it is
selected as the standard encoding for 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 iolist/s, 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 code points:

```
unicode_binary() = binary() with characters encoded in UTF-8 coding standard
chardata() = charlist() | unicode_binary()
charlist() = maybe_improper_list(char() | unicode_binary() | charlist(),
              unicode_binary() | nil())
```

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:

```
external_unicode_binary() = binary() with characters coded in
  a user specified Unicode encoding other than UTF-8 (UTF-16 or UTF-32)
external_chardata() = external_charlist() | external_unicode_binary()
external_charlist() = maybe_improper_list(char() |
                             external_unicode_binary() |
                             external_charlist(),
                             external_unicode_binary() | nil())
```

### 1.2.6 Basic Language Support

As of Erlang/OTP R16 Erlang source files can be written in either UTF-8 or bytewise encoding (a.k.a. latin1 encoding). The details on how to state the encoding of an Erlang source file can be found in `epp(3)`. Strings and comments can be written using Unicode, but functions still have to be named using characters from the ISO-latin-1 character set and atoms are restricted to the same ISO-latin-1 range. These restrictions in the language are of course independent of the encoding of the source file.

#### 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:

```
<<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:

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

#### String and Character Literals

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
1.2 Using Unicode in Erlang

for entering characters of any code point literally in a string even when the encoding of the source file is bytewise (latin1).

In the shell, if using a Unicode input device, or in source code stored in UTF-8, § can be followed directly by a Unicode character producing an integer. In the following example the code point of a Cyrillic # is output:

```
7> $с.
1089
```

Heuristic String Detection

In certain output functions and in the output of return values in the shell, Erlang tries to heuristically detect string data in lists and binaries. Typically you will see heuristic detection in a situation like this:

```
1> [97,98,99].
"abc"
2> <<97,98,99>>.
<<"abc">>
<<"åäö"/utf8>>
```

Here the shell will detect lists containing printable characters or binaries containing printable characters either in bytewise or UTF-8 encoding. The question here is: what is a printable character? One view would be that anything the Unicode standard thinks is printable, will also be printable according to the heuristic detection. The result would be that almost any list of integers will be deemed a string, resulting in all sorts of characters being printed, maybe even characters your terminal does not have in its font set (resulting in some generic output you probably will not appreciate). Another way is to keep it backwards compatible so that only the ISO-Latin-1 character set is used to detect a string. A third way would be to let the user decide exactly what Unicode ranges are to be viewed as characters. Since Erlang/OTP R16B you can select either the whole Unicode range or the ISO-Latin-1 range by supplying the startup flag `+pc Range`, where Range is either `latin1` or `unicode`. For backwards compatibility, the default is `latin1`. This only controls how heuristic string detection is done. In the future, more ranges are expected to be added, so that one can tailor the heuristics to the language and region relevant to the user.

Let's look at an example with the two different startup options:

```
$ erl +pc latin1
Erlang R16B (erts-5.10.1) [source] [async-threads:0] [hipe] [kernel-poll:false]
Eshell V5.10.1  (abort with ^G)
1> [1024].
[1024]
2> [1070,1085,1080,1082,1086,1076].
[1070,1085,1080,1082,1086,1076]
3> [229,228,246].
"åäö"
4> <<208,174,208,189,208,184,208,186,208,190,208,186>>.
<<208,174,208,189,208,184,208,186,208,190,208,186>>
5> <<229/utf8,228/utf8,246/utf8>>.
<<"åäö"/utf8>>
```

```
$ erl +pc unicode
Erlang R16B (erts-5.10.1) [source] [async-threads:0] [hipe] [kernel-poll:false]
```
1.2 Using Unicode in Erlang

In the examples, we can see that the default Erlang shell will only interpret characters from the ISO-Latin1 range as printable and will only detect lists or binaries with those “printable” characters as containing string data. The valid UTF-8 binary containing "#####", will not be printed as a string. When, on the other hand, started with all Unicode characters printable (+pc unicode), the shell will output anything containing printable Unicode data (in binaries either UTF-8 or bytewise encoded) as string data.

These heuristics are also used by io(_lib):format/2 and friends when the t modifier is used in conjunction with ~p or ~P:

In the examples, we can see that the default Erlang shell will only interpret characters from the ISO-Latin1 range as printable and will only detect lists or binaries with those “printable” characters as containing string data. The valid UTF-8 binary containing "#####", will not be printed as a string. When, on the other hand, started with all Unicode characters printable (+pc unicode), the shell will output anything containing printable Unicode data (in binaries either UTF-8 or bytewise encoded) as string data.

These heuristics are also used by io(_lib):format/2 and friends when the t modifier is used in conjunction with ~p or ~P:

1.2.7 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 XTerm, 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:

Please observe that this only affects heuristic interpretation of lists and binaries on output. For example the ~ts format sequence does always output a valid lists of characters, regardless of the +pc setting, as the programmer has explicitly requested string output.

$ echo $LANG
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 R16B (erts-5.10.1) [source] [async-threads:0] [hipe] [kernel-poll:false]
Eshell V5.10.1  (abort with ^G)
1> lists:keyfind(encoding,1,io:getopts()).
   {encoding,latin1}
2> q().
ok
$ LC_CTYPE=en_US.UTF-8 erl
Erlang R16B (erts-5.10.1) [source] [async-threads:0] [hipe] [kernel-poll:false]
Eshell V5.10.1  (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:

```
$ erl
Erlang R16B (erts-5.10.1) [source] [async-threads:0] [hipe] [kernel-poll:false]
Eshell V5.10.1  (abort with ^G)
1> lists:keyfind(encoding,1,io:getopts()).
   {encoding,unicode}
2> "Юникод".
   "Юникод"
3> io:format("~-ts-n", [v(2)]).
   Юникод
ok
4>
```

20 | Ericsson AB. All Rights Reserved.: STDLIB
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:

```erlang
$ erl
Erlang R16B (erts-5.10.1) [source] [async-threads:0] [hipe] [kernel-poll:false]
Eshell V5.10.1 (abort with ^G)
1> $ξ.
958
2> Юникод.
   * 1: illegal character
2>
```

### 1.2.8 Unicode File Names

Most modern operating systems support Unicode file names in some way or another. There are several different ways to do this and Erlang by default treats the different approaches differently:

**Mandatory Unicode file naming**

Windows and, for most common uses, MacOS X enforces Unicode support for file names. All files created in the file system have names that can consistently be interpreted. In MacOS X, all file names are retrieved in UTF-8 encoding, while Windows has selected an approach where each system call handling file names has a special Unicode aware variant, giving much the same effect. There are no file names on these systems that are not Unicode file names, why the default behavior of the Erlang VM is to work in "Unicode file name translation mode", meaning that a file name can be given as a Unicode list and that will be automatically translated to the proper name encoding for the underlying operating and file system.

Doing i.e. a `file:list_dir/1` on one of these systems may return Unicode lists with code points beyond 255, depending on the content of the actual file system.

As the feature is fairly new, you may still stumble upon non core applications that cannot handle being provided with file names containing characters with code points larger than 255, but the core Erlang system should have no problems with Unicode file names.

**Transparent file naming**

Most Unix operating systems have adopted a simpler approach, namely that Unicode file naming is not enforced, but by convention. Those systems usually use UTF-8 encoding for Unicode file names, but do not enforce it. On such a system, a file name containing characters having code points between 128 and 255 may be named either as plain ISO-Latin-1 or using UTF-8 encoding. As no consistency is enforced, the Erlang VM can do no consistent translation of all file names.

By default on such systems, Erlang starts in utf8 file name mode if the terminal supports UTF-8, otherwise in latin1 mode.

In the latin1 mode, file names are byte wise encoded. This allows for list representation of all file names in the system, but, for example, a file named "Östersund.txt", will appear in `file:list_dir/1` as either "Östersund.txt" (if the file name was encoded in byte wise ISO-Latin-1 by the program creating the file, or more probably as [195,150,115,116,101,114,115,117,110,100], which is a list containing UTF-8 bytes - not what you would want... If you on the other hand use Unicode file name translation on such a system, non-UTF-8 file names will simply be ignored by functions like `file:list_dir/1`. They can be retrieved with `file:list_dir_all/1`, but wrongly encoded file names will appear as 'raw file names'.

The Unicode file naming support was introduced with Erlang/OTP R14B01. A VM operating in Unicode file name translation mode can work with files having names in any language or character set (as long as it is supported by the underlying OS and file system). The Unicode character list is used to denote file or directory names and if the file system content is listed, you will also get Unicode lists as return value. The support lies in the Kernel and STDLIB
modules, why most applications (that does not explicitly require the file names to be in the ISO-latin-1 range) will benefit from the Unicode support without change.

On operating systems with mandatory Unicode file names, this means that you more easily conform to the file names of other (non Erlang) applications, and you can also process file names that, at least on Windows, were completely inaccessible (due to having names that could not be represented in ISO-latin-1). Also you will avoid creating incomprehensible file names on MacOS X as the vfs layer of the OS will accept all your file names as UTF-8 and will not rewrite them.

For most systems, turning on Unicode file name translation is no problem even if it uses transparent file naming. Very few systems have mixed file name encodings. A consistent UTF-8 named system will work perfectly in Unicode file name mode. It was still however considered experimental in Erlang/OTP R14B01 and is still not the default on such systems. Unicode file name translation is turned on with the +fnu switch to the On Linux, a VM started without explicitly stating the file name translation mode will default to latin1 as the native file name encoding. On Windows and MacOS X, the default behavior is that of Unicode file name translation, why the file: native_name_encoding/0 by default returns utf8 on those systems (the fact that Windows actually does not use UTF-8 on the file system level can safely be ignored by the Erlang programmer). The default behavior can, as stated before, be changed using the +fnu or +fnl options to the VM, see the erl program. If the VM is started in Unicode file name translation mode, file: native_name_encoding/0 will return the atom utf8. The +fnu switch can be followed by w, i or e, to control how wrongly encoded file names are to be reported. w means that a warning is sent to the error_logger whenever a wrongly encoded file name is "skipped" in directory listings, i means that those wrongly encoded file names are silently ignored and e means that the API function will return an error whenever a wrongly encoded file (or directory) name is encountered. w is the default. Note that file: read_link/1 will always return an error if the link points to an invalid file name.

In Unicode file name mode, file names given to the BIF open_port/2 with the option {spawn_executable, ...} are also interpreted as Unicode. So is the parameter list given in the args option available when using spawn_executable. The UTF-8 translation of arguments can be avoided using binaries, see the discussion about raw file names below.

It is worth noting that the file encoding options given when opening a file has nothing to do with the file name encoding convention. You can very well open files containing data encoded in UTF-8 but having file names in bytewise (latin1) encoding or vice versa.

Note:

Erlang drivers and NIF shared objects still can not be named with names containing code points beyond 127. This is a known limitation to be removed in a future release. Erlang modules however can, but it is definitely not a good idea and is still considered experimental.

Notes About Raw File Names

Raw file names were introduced together with Unicode file name support in erts-5.8.2 (Erlang/OTP R14B01). The reason "raw file names" was introduced in the system was to be able to consistently represent file names given in different encodings on the same system. Having the VM automatically translate a file name that is not in UTF-8 to a list of Unicode characters might seem practical, but this would open up for both duplicate file names and other inconsistent behavior. Consider a directory containing a file named "björn" in ISO-latin-1, while the Erlang VM is operating in Unicode file name mode (and therefore expecting UTF-8 file naming). The ISO-latin-1 name is not valid UTF-8 and one could be tempted to think that automatic conversion in for example file: list_dir/1 is a good idea. But what would happen if we later tried to open the file and have the name as a Unicode list (magically converted from the ISO-latin-1 file name)? The VM will convert the file name given to UTF-8, as this is the encoding expected. Effectively this means trying to open the file named "<<björn'/utf8>>". This file does not exist, and even if it existed it would not be the same file as the one that was listed. We could even create two files named "björn", one named in the
UTF-8 encoding and one not. If \texttt{file:list_dir/1} would automatically convert the ISO-latin-1 file name to a list, we would get two identical file names as the result. To avoid this, we need to differentiate between file names being properly encoded according to the Unicode file naming convention (i.e. UTF-8) and file names being invalid under the encoding. By the common \texttt{file:list_dir/1} function, the wrongly encoded file names are simply ignored in Unicode file name translation mode, but by the \texttt{file:list_dir_all/1} function, the file names with invalid encoding are returned as “raw” file names, i.e. as binaries.

The Erlang \texttt{file} module accepts raw file names as input. \texttt{open_port({spawn_executable, ...} ...)} also accepts them. As mentioned earlier, the arguments given in the option list to \texttt{open_port({spawn_executable, ...} ...)} undergo the same conversion as the file names, meaning that the executable will be provided with arguments in UTF-8 as well. This translation is avoided consistently with how the file names are treated, by giving the argument as a binary.

To force Unicode file name translation mode on systems where this is not the default was considered experimental in Erlang/OTP R14B01 due to the fact that the initial implementation did not ignore wrongly encoded file names, so that raw file names could spread unexpectedly throughout the system. Beginning with Erlang/OTP R16B, the wrongly encoded file names are only retrieved by special functions (e.g. \texttt{file:list_dir_all/1}), so the impact on existing code is much lower, why it is now supported. Unicode file name translation is expected to be default in future releases.

Even if you are operating without Unicode file naming translation automatically done by the VM, you can access and create files with names in UTF-8 encoding by using raw file names encoded as UTF-8. Enforcing the UTF-8 encoding regardless of the mode the Erlang VM is started in might, in some circumstances be a good idea, as the convention of using UTF-8 file names is spreading.

Notes About MacOS X

MacOS X’s vfs layer enforces UTF-8 file names in a quite aggressive way. Older versions did this by simply refusing to create non UTF-8 conforming file names, while newer versions replace offending bytes with the sequence ”%HH”, where HH is the original character in hexadecimal notation. As Unicode translation is enabled by default on MacOS X, the only way to come up against this is to either start the VM with the +fnl flag or to use a raw file name in bytewise (latin1) encoding. If using a raw filename, with a bytewise encoding containing characters between 127 and 255, to create a file, the file can not be opened using the same name as the one used to create it. There is no remedy for this behaviour, other than keeping the file names in the right encoding.

MacOS X also reorganizes the names of files so that the representation of accents etc is using the "combining characters", i.e. the character ö is represented as the code points [111,776], where 111 is the character o and 776 is the special accent character "combining diaeresis". This way of normalizing Unicode is otherwise very seldom used and Erlang normalizes those file names in the opposite way upon retrieval, so that file names using combining accents are not passed up to the Erlang application. In Erlang the file name “björn” is retrieved as [98,106,117,776,114,110], even though the file system might think differently. The normalization into combining accents are redone when actually accessing files, so this can usually be ignored by the Erlang programmer.

1.2.9 Unicode in Environment and Parameters

Environment variables and their interpretation is handled much in the same way as file names. If Unicode file names are enabled, environment variables as well as parameters to the Erlang VM are expected to be in Unicode.

If Unicode file names are enabled, the calls to \texttt{os:getenv/0, os:getenv/1} and \texttt{os:putenv/2} will handle Unicode strings. On Unix-like platforms, the built-in functions will translate environment variables in UTF-8 to/from Unicode strings, possibly with code points > 255. On Windows the Unicode versions of the environment system API will be used, also allowing for code points > 255.

On Unix-like operating systems, parameters are expected to be UTF-8 without translation if Unicode file names are enabled.
1.2.10 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 should not 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.

file, group, user
I/O-servers throughout the system are able 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.

re
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 perfectly 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, as there are language and locale issues as well as multi-character mappings to consider when converting text between cases. Converting case in an international environment is a big subject not yet addressed in OTP.

1.2.11 Unicode Data in Files

The fact that Erlang as such can handle Unicode data in many forms does not automatically mean that the content of any file can be Unicode text. The external entities such as ports or I/O-servers are not generally Unicode capable.

Ports are always byte oriented, so before sending data that you are not sure is byte-wise encoded to a port, make sure to encode it in a proper Unicode encoding. Sometimes this will mean that only part of the data shall be encoded as e.g. UTF-8, some parts may be binary data (like a length indicator) or something else that shall not undergo character encoding, so no automatic translation is present.

I/O-servers behave a little differently. The I/O-servers connected to terminals (or stdout) can usually cope with Unicode data regardless of the encoding option. This is convenient when one expects a modern environment but do not
want to crash when writing to an archaic terminal or pipe. Files on the other hand are more picky. A file can have an encoding option which makes it generally usable by the io-module (e.g. \{encoding,utf8\}), but is by default opened as a byte oriented file. The \texttt{file} module is byte oriented, why only ISO-Latin-1 characters can be written using that module. The \texttt{io} module is the one to use if Unicode data is to be output to a file with other encoding than latin1 (a.k.a. bytewise encoding). It is slightly confusing that a file opened with e.g. \texttt{file:open(Name, \{read,\{encoding,utf8\}\})}, cannot be properly read using \texttt{file:read(File,N)} but you have to use the \texttt{io} module to retrieve the Unicode data from it. The reason is that \texttt{file:read} and \texttt{file:write} (and friends) are purely byte oriented, and should so be, as that is the way to access files other than text files - byte by byte. Just as with ports, you can of course write encoded data into a file by "manually" converting the data to the encoding of choice (using the \texttt{unicode} module or the bit syntax) and then output it on a bytewise encoded (latin1) file.

The rule of thumb is that the \texttt{file} module should be used for files opened for byte access (\{encoding,latin1\}) and the \texttt{io} module should be used when accessing files with any other encoding (e.g. \{encoding,utf8\}).

Functions reading Erlang syntax from files generally recognize the \texttt{coding: comment} and can therefore handle Unicode data on input. When writing Erlang Terms to a file, you should insert such comments when applicable:

$ \texttt{erl +fna +pc unicode}

Erlang R16B (erts-5.10.1) [source] [async-threads:0] [hipe] [kernel-poll:false]
Eshell V5.10.1 (abort with ^G)

1> file:write_file("test.term",<<"%% coding: utf-8\n[{"Юникод",4711}]\n"/utf8>>).
ok
2> file:consult("test.term").
{ok,[[{"Юникод",4711}]]}

1.2.12 Summary of Options

The Unicode support is controlled by both command line switches, some standard environment variables and the version of OTP you are using. Most options affect mainly the way Unicode data is displayed, not the actual functionality of the APIs in the standard libraries. This means that Erlang programs usually do not need to concern themselves with these options, they are more for the development environment. An Erlang program can be written so that it works well regardless of the type of system or the Unicode options that are in effect.

Here follows a summary of the settings affecting Unicode:

The \texttt{LANG} and \texttt{LC_CTYPE} environment variables

- The language setting in the OS mainly affects the shell. The terminal (i.e. the group leader) will operate with \{encoding,unicode\} only if the environment tells it that UTF-8 is allowed. This setting should correspond to the actual terminal you are using.

- The environment can also affect file name interpretation, if Erlang is started with the +fna flag (which is default from Erlang/OTP 17.0).

You can check the setting of this by calling \texttt{io:getopts()}, which will give you an option list containing \{encoding,unicode\} or \{encoding,latin1\}.

The +pc \{unicode|latin1\} flag to \texttt{erl(1)}

This flag affects what is interpreted as string data when doing heuristic string detection in the shell and in \texttt{io/io_lib:format} with the "~tp" and "tP" formatting instructions, as described above.

You can check this option by calling \texttt{io:printable_range(0)}, which will return \texttt{unicode} or \texttt{latin1}. To be compatible with future (expected) extensions to the settings, one should rather use
io_lib:printable_list/1 to check if a list is printable according to the setting. That function will take into account new possible settings returned from io:printable_range/0.

The +fn[l|a|u] [[w|i|e]] flag to erl(1)

This flag affects how the file names are to be interpreted. On operating systems with transparent file naming, this has to be specified to allow for file naming in Unicode characters (and for correct interpretation of file names containing characters > 255.

+fnl means byte-wise interpretation of file names, which was the usual way to represent ISO-Latin-1 file names before UTF-8 file naming got widespread.

+fnu means that file names are encoded in UTF-8, which is nowadays the common scheme (although not enforced).

+fna means that you automatically select between +fnl and +fnu, based on the LANG and LC_CTYPE environment variables. This is optimistic heuristics indeed, nothing enforces a user to have a terminal with the same encoding as the file system, but usually, this is the case. This is the default on all Unix-like operating systems except MacOS X.

The file name translation mode can be read with the file:native_name_encoding/0 function, which returns latin1 (meaning byte-wise encoding) or utf8.

epp:default_encoding/0

This function returns the default encoding for Erlang source files (if no encoding comment is present) in the currently running release. In Erlang/OTP R16B latin1 was returned (meaning byte-wise encoding). In Erlang/OTP 17.0 and forward it returns utf8.

The encoding of each file can be specified using comments as described in epp(3).

io:setopts/[1,2] and the -oldshell/-noshell flags.

When Erlang is started with -oldshell or -noshell, the I/O-server for standard_io is default set to byte-wise encoding, while an interactive shell defaults to what the environment variables says.

With the io:setopts/2 function you can set the encoding of a file or other I/O-server. This can also be set when opening a file. Setting the terminal (or other standard_io server) unconditionally to the option {encoding,utf8} will for example make UTF-8 encoded characters being written to the device regardless of how Erlang was started or the users environment.

Opening files with encoding option is convenient when writing or reading text files in a known encoding. You can retrieve the encoding setting for an I/O-server using io:getopts().

1.2.13 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 code point `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),
```

26 | Ericsson AB. All Rights Reserved.: STDLIB
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/2` 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
code:open_bom_file_for_writing(File, Encoding) ->
    {ok, F} = file:open(File, [write, binary]),
    ok = file:write(F, 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 code points beyond the ISO-latin-1 range.

### Formatted I/O

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 do not 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 code points and expect binaries to be in UTF-8:

```erlang
1> io:format("~ts-n",[<<"åäö"/utf8>>]).
åäö
ok
2> io:format("~s-n",[<<"åäö"/utf8>>]).
ÅÄÖÅÄ
ok
```

Obviously the second `io:format/2` gives undesired output because the UTF-8 binary is not in latin1. For backward compatibility, the non prefixed `s` control character expects byte-wise encoded ISO-latin-1 characters in binaries and lists containing only code points < 256.

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 right choice of formatting characters. A byte-wise encoded binary will also be interpreted as a string and printed even when using `~ts`, but it might be mistaken for a valid UTF-8 string and one should therefore avoid using the `~ts` control if the binary contains byte-wise encoded characters and not UTF-8.

The function `format/2` 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/1`. 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/1` will in that case fail. However, if the I/O server you want to communicate with is Unicode-aware, the list returned can still be used directly:
1.2 Using Unicode in Erlang

The Unicode string is returned as a Unicode list, which is recognized as such since the Erlang shell uses the Unicode encoding (and is started with all Unicode characters considered printable). The Unicode list is valid input to the `io:put_chars/2` function, so data can be output on any Unicode capable device. 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 code points beyond ISO-latin-1 if encoding is set to something else than `latin1`.

### Heuristic Identification of UTF-8

While it is 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/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))
    end;
```

Heuristic Identification of UTF-8

While it is 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/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))
    end;
```

Heuristic Identification of UTF-8

While it is 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/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))
    end;
```
Another option is to try to read the whole file in UTF-8 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 code point over 255:

```erlang
define_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;
latin1;
loop_through_file2(_,{error,_Err}) ->
  latin1;
loop_through_file2(F,Bin) when is_binary(Bin) ->
  loop_through_file2(F,io:get_chars(F,'',1024)).
```

### Lists of UTF-8 Bytes

For various reasons, you may find yourself having a list of UTF-8 bytes. This is not a regular string of Unicode characters as each element in the list does not contain one character. Instead you get the "raw" UTF-8 encoding that you have in binaries. This is easily converted to a proper Unicode string by first converting byte per byte into a binary and then converting the binary of UTF-8 encoded characters back to a Unicode string:

```erlang
utf8_list_to_string(StrangeList) ->
  unicode:characters_to_list(list_to_binary(StrangeList)).
```

### Double UTF-8 Encoding

When working with binaries, you may get the horrible "double UTF-8 encoding", where strange characters are encoded in your binaries or files that you did not expect. What you may have got, is a UTF-8 encoded binary that is for the second time encoded as UTF-8. A common situation is where you read a file, byte by byte, but the actual content is already UTF-8. If you then convert the bytes to UTF-8, using i.e. the `unicode` module or by writing to a file opened with the `{encoding,utf8}` option. You will have each byte in the in the input file encoded as UTF-8, not each character of the original text (one character may have been encoded in several bytes). There is no real remedy for this other than being very sure of which data is actually encoded in which format, and never convert UTF-8 data (possibly read byte by byte from a file) into UTF-8 again.

The by far most common situation where this happens, is when you get lists of UTF-8 instead of proper Unicode strings, and then convert them to UTF-8 in a binary or on a file:

```erlang
wrong_thing_to_do() ->
  {ok,Bin} = file:read_file("an_utf8_encoded_file.txt"),
  MyList = binary_to_list(Bin), % Wrong! It is an utf8 binary!
  {ok,C} = file:open("catastrophe.txt",[write,{encoding,utf8}]),
  io:put_chars(C,MyList), % Expects a Unicode string, but get UTF-8
  file:close(C). % The file catastrophe.txt contains more or less unreadable
                 % garbage!
```

Make very sure you know what a binary contains before converting it to a string. If no other option exists, try heuristics:
if you can not know() ->
    {ok,Bin} = file:read_file("maybe_utf8_encoded_file.txt"),
    MyList = case unicode:characters_to_list(Bin) of
        L when is_list(L) ->
            L;
        _ ->
            binary_to_list(Bin) %% The file was bytewise encoded
    end,
    %% Now we know that the list is a Unicode string, not a list of UTF-8 bytes
    {ok,G} = file:open("greatness.txt",[write,[encoding,utf8]]),
    io:put_chars(G,MyList), %% Expects a Unicode string, which is what it gets!
    file:close(G). %% The file contains valid UTF-8 encoded Unicode characters!
2 Reference Manual

The Standard Erlang Libraries application, STDLIB, contains modules for manipulating lists, strings and files etc.
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 = boolean()
   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_prompt_func = {Mod, Func} | default
   where
      • Mod = atom()
      • Func = atom()
   This parameter can be used to set a customized Erlang shell prompt function.
shell_saved_results = integer() >= 0
   This parameter can be used to determine how many results are saved by the Erlang shell.
shell_strings = boolean()
   This parameter can be used to determine how the Erlang shell outputs lists of integers.

See Also
app(4), application(3), shell(3),
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). 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.

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()).
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(Type)

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.
array()

array() is equivalent to array(term()).

array_index() = integer() >= 0

array_opts() = array_opt() | [array_opt()]

array_opt() = {fixed, boolean()}
  | fixed
  | {default, Type :: term()}
  | {size, N :: integer() >= 0}
  | (N :: integer() >= 0)

indx_pairs(Type) = [indx_pair(Type)]
indx_pair(Type) = {Index :: array_index(), Type}

Exports

default(Array :: array(Type)) -> Value :: Type

Get the value used for uninitialized entries.

See also: new/2.

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

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

See also: relax/1.

foldl(Function, InitialAcc :: A, Array :: array(Type)) -> B

Types:
  Function = fun((Index :: array_index(), Value :: Type, Acc :: A) -> B)

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, map/2, sparse_foldl/3.

foldr(Function, InitialAcc :: A, Array :: array(Type)) -> B

Types:
  Function = fun((Index :: array_index(), Value :: Type, Acc :: A) -> B)

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, map/2.

from_list(List :: [Value :: Type]) -> array(Type)

Equivalent to from_list(List, undefined).

from_list(List :: [Value :: Type], Default :: term()) ->
**array**(Type)

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, to_list/1.*

```
from_orddict(Orddict :: indx_pairs(Value :: Type)) -> array(Type)
```

Equivalent to `from_orddict(Orddict, undefined)`.

```
from_orddict(Orddict :: indx_pairs(Value :: Type),
              Default :: Type) -> array(Type)
```

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 Orddict is not a proper, ordered list of pairs whose first elements are nonnegative integers, the call fails with reason **badarg**.

*See also: new/2, to_orddict/1.*

```
get(I :: array_indx(), Array :: array(Type)) -> Value :: Type
```

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.*

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

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()) -> boolean()
```

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

*See also: fix/1.*

```
map(Function, Array :: array(Type1)) -> array(Type2)
```

Types:

```
  Function = fun((Index :: array_indx(), Type1) -> Type2)
```

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, foldr/3, sparse_map/2.*

```
ew() -> array()
```

Create a new, extendible array with initial size zero.

*See also: new/1, new/2.*
new(Options :: `array_opts()`) -> 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() >= 0 or {size, N::integer() >= 0}
  
  Specifies the initial size of the array; this also implies \{fixed, true\}. If \(N\) is not a nonnegative integer, the call fails with reason \texttt{badarg}.

- \{fixed, \texttt{true}\}
  
  Creates a fixed-size array; see also \texttt{fix/1}.

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

- \{default, \texttt{Value}\}
  
  Sets the default value for the array to \texttt{Value}.

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

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

Examples:

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

creates a fixed-size array of size 100.

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

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

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

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

See also: \texttt{fix/1}, \texttt{from_list/2}, \texttt{get/2}, \texttt{new0}, \texttt{new/2}, \texttt{set/3}.

new(Size :: integer() >= 0, Options :: `array_opts()`) -> array()

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

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

Example:

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

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

See also: \texttt{new/1}.
relax(Array :: array(Type)) -> array(Type)
Make the array resizable. (Reverses the effects of fix/1.)
See also: fix/1.

reset(I :: array_index(), Array :: array(Type)) -> array(Type)
Reset entry I to the default value for the array. If the value of entry I is the default value the array will be returned unchanged. Reset will never change size of the array. Shrinking can be done explicitly by calling resize/2.
If I is not a nonnegative integer, or if the array has fixed size and I is larger than the maximum index, the call fails with reason badarg; cf. set/3
See also: new/2, set/3.

resize(Array :: array(Type)) -> array(Type)
Change the size of the array to that reported by sparse_size/1. If the given array has fixed size, the resulting array will also have fixed size.
See also: resize/2, sparse_size/1.

resize(Size :: integer() >= 0, Array :: array(Type)) -> array(Type)
Change the size of the array. If Size is not a nonnegative integer, the call fails with reason badarg. If the given array has fixed size, the resulting array will also have fixed size.

set(I :: array_index(), Value :: Type, Array :: array(Type)) -> array(Type)
Set entry I of the array to Value. If I is not a nonnegative integer, or if the array has fixed size and I is larger than the maximum index, the call fails with reason badarg.
If the array does not have fixed size, and I is greater than size(Array) - 1, the array will grow to size I + 1.
See also: get/2, reset/2.

size(Array :: array()) -> integer() >= 0
Get the number of entries in the array. Entries are numbered from 0 to size(Array) - 1; hence, this is also the index of the first entry that is guaranteed to not have been previously set.
See also: set/3, sparse_size/1.

sparse_foldl(Function, InitialAcc :: A, Array :: array(Type)) -> B
Types:
    Function = fun((Index :: array_index(), Value :: Type, Acc :: A) -> B)
Fold the elements of the array using the given function and initial accumulator value, skipping default-valued entries. The elements are visited in order from the lowest index to the highest. If Function is not a function, the call fails with reason badarg.
See also: foldl/3, sparse_foldr/3.
sparse_foldr(Function, InitialAcc :: A, Array :: array(Type)) -> B

Types:
   Function = fun((Index :: array_indx(), Value :: Type, Acc :: A) -> B)

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, sparse_foldl/3.

sparse_map(Function, Array :: array(Type1)) -> array(Type2)

Types:
   Function = fun((Index :: array_indx(), Type1) -> Type2)

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.

sparse_size(Array :: array()) -> integer() >= 0

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, size/1.

sparse_to_list(Array :: array(Type)) -> [Value :: Type]

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

See also: to_list/1.

sparse_to_orddict(Array :: array(Type)) -> indx_pairs(Value :: Type)

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

See also: to_orddict/1.

to_list(Array :: array(Type)) -> [Value :: Type]

Converts the array to a list.

See also: from_list/2, sparse_to_list/1.

to_orddict(Array :: array(Type)) -> indx_pairs(Value :: Type)

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

See also: from_orddict/2, sparse_to_orddict/1.
base64

Erlang module

Implements base 64 encode and decode, see RFC2045.

Data Types

\[
\text{ascii_string()} = [1..255] \\
\text{ascii_binary()} = \text{binary()}
\]

A binary() with ASCII characters in the range 1 to 255.

Exports

\[
\text{encode}(	ext{Data}) \rightarrow \text{Base64} \\
\text{encode_to_string}(	ext{Data}) \rightarrow \text{Base64String}
\]

Types:

\[
\begin{align*}
\text{Data} &= \text{ascii_string()} \mid \text{ascii_binary()} \\
\text{Base64} &= \text{ascii_binary()} \\
\text{Base64String} &= \text{ascii_string()}
\end{align*}
\]

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

\[
\text{decode}(	ext{Base64}) \rightarrow \text{Data} \\
\text{decode_to_string}(	ext{Base64}) \rightarrow \text{DataString}
\]

\[
\begin{align*}
\text{mime_decode}(	ext{Base64}) &\rightarrow \text{Data} \\
\text{mime_decode_to_string}(	ext{Base64}) &\rightarrow \text{DataString}
\end{align*}
\]

Types:

\[
\begin{align*}
\text{Base64} &= \text{ascii_string()} \mid \text{ascii_binary()} \\
\text{Data} &= \text{ascii_binary()} \\
\text{DataString} &= \text{ascii_string()}
\end{align*}
\]

Decodes a base64 encoded string to plain ASCII. See RFC4648. mime_decode/1 and mime_decode_to_string/1 strips away illegal characters, while decode/1 and decode_to_string/1 only strips away whitespace characters.
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, strip_files/1 and/or strip_release/1`.

**Reconstructing source code**

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

```erlang
{ok,_,[{abstract_code,_,AC}]} = beam_lib:chunks(Beam,[abstract_code]).
io:fwrite("-5-n", [erl_prettypr:format(erl_syntax:form_list(AC))]).
```

**Encrypted debug information**

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

To use encrypted debug information, a key must be provided to the compiler and `beam_lib`. The key is given as a string and it is recommended that it contains at least 32 characters and that both upper and lower case letters as well as digits and special characters are used.
The default type -- and currently the only type -- of crypto algorithm is des3_cbc, three rounds of DES. The key string will be scrambled using erlang:md5/1 to generate the actual keys used for des3_cbc.

Note:
As far as we know 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:

- Use the compiler option {debug_info, Key}, see compile(3), and the function crypto_key_fun/1 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.
- Store the key in a text file named .erlang.crypt.
  
  In this case, the compiler option encrypt_debug_info can be used, see compile(3).

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

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

{debug_info, Mode, Module, Key}

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

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

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

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

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

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

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

Ericsson AB. All Rights Reserved.: STDLIB | 41
Data Types

beam() = module() | file:filename() | binary()

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

chunkdata() = \{chunkid(), dataB()\}
   | \{abstract_code, abst_code()\}
   | \{attributes, [attrib_entry()]\}
   | \{compile_info, [compinfo_entry()]\}
   | \{exports, [[atom(), arity()]]\}
   | \{imports, [mfa()]\}
   | \{indexed_imports,
       [index(),
        module(),
        Function :: atom(),
        arity()]]\}
   | \{locals, [[atom(), arity()]]\}
   | \{labeled_locals, [labeled_entry()]\}
   | \{atoms, [[integer(), atom()]]\}

The list of attributes is sorted on Attribute (in attrib_entry()), 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() = nonempty_string()

"Abst" | "Attr" | "CInf" | "ExpT" | "ImpT" | "LocT" | "Atom"

dataB() = binary()

abst_code() = {AbstVersion :: atom(), forms()}
   | no_abstract_code

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.

forms() = [erl_parse:abstract_form()]

compinfo_entry() = {InfoKey :: atom(), term()}

attrib_entry() =
   \{Attribute :: atom(), [AttributeValue :: term()]\}

labeled_entry() = \{Function :: atom(), arity(), label()\}

index() = integer() >= 0

label() = integer()

chunkref() = chunkname() | chunkid()

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

chnk_rsn() = \{unknown_chunk, file:filename(), atom()\} |
| {key_missing_or_invalid, file:filename(), abstract_code} |
| info_rsn() |

info_rsn() = \{chunk_too_big, file:filename(), chunkid(),
                ChunkSize :: integer() >= 0,
                FileSize :: integer() >= 0\} |
| {invalid_beam_file, file:filename(),
                Position :: integer() >= 0\} |
| {invalid_chunk, file:filename(), chunkid()} |
| {missing_chunk, file:filename(), chunkid()} |
| {not_a_beam_file, file:filename()} |
| {file_error, file:filename(), file:posix()} |

Exports

chunks(Beam, ChunkRefs) ->
  \{ok, \{module(), [chunkdata()]\}\} |
  \{error, beam_lib, chnk_rsn()\}

Types:
  Beam = beam()
  ChunkRefs = \[chunkref()\]

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, ChunkRefs, Options) ->
  \{ok, \{module(), [ChunkResult]\}\} |
  \{error, beam_lib, chnk_rsn()\}

Types:
  Beam = beam()
  ChunkRefs = \[chunkref()\]
  Options = \[allow_missing_chunks\]
  ChunkResult = chunkdata() |
               \{ChunkRef :: chunkref(), missing_chunk\}

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 :: term()]\}\} |
{error, beam_lib, chnk_rsn()}

Types:

**Beam** = *beam()*

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, [275613208176997377698994100858909383631]}}

**md5(Beam)** -> `{ok, {module(), MD5}}` | `{error, beam_lib, chnk_rsn()}`

Types:

**Beam** = *beam()*

**MD5** = *binary()*

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

**info(Beam)** -> `[InfoPair]` | `{error, beam_lib, info_rsn()}`

Types:

**Beam** = *beam()*

**InfoPair** = `{file, Filename :: file:filename()}`

| `{binary, Binary :: binary()}`
| `{module, Module :: module()}`
| `{chunks, [{ChunkId :: chunkid(),
   Pos :: integer() >= 0,
   Size :: integer() >= 0}]}`

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

{file, Filename} | {binary, Binary}

The name (string) of the BEAM file, or the binary from which the information was extracted.

{module, Module}

The name (atom) of the module.

{chunks, [{ChunkId, Pos, Size}]}

For each chunk, the identifier (string) and the position and size of the chunk data, in bytes.

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

Types:
Beam1 = Beam2 = beam()
cmp_rsn() = {modules_different, module(), module()}
    | {chunks_different, chunkid()}
    | different_chunks
    | info_rsn()

Compares the contents of two BEAM files. If the module names are the same, and all chunks except for the "CInf" chunk (the chunk containing the compilation information which is returned by Module:module_info(compile)) 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, Reason}

Types:
    Dir1 = Dir2 = atom() | file:filename()
    Only1 = Only2 = [file:filename()]
    Different = [{Filename1 :: file:filename(), Filename2 :: file:filename()}]
    Reason = {not_a_directory, term()} | info_rsn()

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

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

Types:
    Dir1 = Dir2 = atom() | file:filename()
    Reason = {not_a_directory, term()} | info_rsn()

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

strip(Beam1) ->
    {ok, {module(), Beam2}} | {error, beam_lib, info_rsn()}

Types:
    Beam1 = Beam2 = beam()

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(), Beam}]} |
    {error, beam_lib, info_rsn()}

Types:
    Files = [beam()]
    Beam = beam()

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.
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())`.

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:

```
crypto_key_fun(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.

```
crypto_key_fun({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.
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.

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

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`.

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

Types:

`Result = undefined | term()`

Unregisters the crypto key fun and terminates the process holding it, started by `crypto_key_fun/1`. 
This module contains functions for manipulating byte-oriented binaries. Although the majority of functions could be implemented using bit-syntax, the functions in this library are highly optimized and are expected to either execute faster or consume less memory (or both) than a counterpart written in pure Erlang.

The module is implemented according to the EEP (Erlang Enhancement Proposal) 31.

Note:
The library handles byte-oriented data. Bitstrings that are not binaries (does not contain whole octets of bits) will result in a badarg exception being thrown from any of the functions in this module.

Data Types

cp()
Opaque data-type representing a compiled search-pattern. Guaranteed to be a tuple() to allow programs to distinguish it from non precompiled search patterns.

part() = {Start :: integer() >= 0, Length :: integer()}
A representation of a part (or range) in a binary. Start is a zero-based offset into a binary() and Length is the length of that part. As input to functions in this module, a reverse part specification is allowed, constructed with a negative Length, so that the part of the binary begins at Start + Length and is -Length long. This is useful for referencing the last N bytes of a binary as {size(Binary), -N}. The functions in this module always return part()'s with positive Length.

Exports

at(Subject, Pos) -> byte()
Types:
Subject = binary()
Pos = integer() >= 0
Returns the byte at position Pos (zero-based) in the binary Subject as an integer. If Pos >= byte_size(Subject), a badarg exception is raised.

bin_to_list(Subject) -> [byte()]
Types:
Subject = binary()
The same as bin_to_list(Subject, {0, byte_size(Subject)}).

bin_to_list(Subject, PosLen) -> [byte()]
Types:
Subject = binary()
PosLen = part()

Converts Subject to a list of byte(), each representing the value of one byte. The part() denotes which part of the binary() to convert. Example:

```
1> binary:bin_to_list(<"erlang">>, {1,3}).
"rla"
%% or [114,108,97] in list notation.
```

If PosLen in any way references outside the binary, a badarg exception is raised.

```
bin_to_list(Subject, Pos, Len) -> [byte()]
Types:
   Subject = binary()
   Pos = integer() >= 0
   Len = integer()

The same as bin_to_list(Subject, {Pos, Len}).
```

```
compile_pattern(Pattern) -> cp()
Types:
   Pattern = binary() | [binary()]

Builds an internal structure representing a compilation of a search-pattern, later to be used in the match/3, matches/3, split/3 or replace/4 functions. The cp() returned is guaranteed to be a tuple() to allow programs to distinguish it from non pre-compiled search patterns.

When a list of binaries is given, it denotes a set of alternative binaries to search for. E.g. if [<<"functional">>, <<"programming">>] is given as Pattern, this means "either <<"functional">> or <<"programming">>". The pattern is a set of alternatives; when only a single binary is given, the set has only one element. The order of alternatives in a pattern is not significant.

The list of binaries used for search alternatives shall be flat and proper.

If Pattern is not a binary or a flat proper list of binaries with length > 0, a badarg exception will be raised.

```
copy(Subject) -> binary()
Types:
   Subject = binary()

The same as copy(Subject, 1).
```

```
copy(Subject, N) -> binary()
Types:
   Subject = binary()
   N = integer() >= 0

Creates a binary with the content of Subject duplicated N times.

This function will always create a new binary, even if N = 1. By using copy/1 on a binary referencing a larger binary, one might free up the larger binary for garbage collection.
```
Note:
By deliberately copying a single binary to avoid referencing a larger binary, one might, instead of freeing up the larger binary for later garbage collection, create much more binary data than needed. Sharing binary data is usually good. Only in special cases, when small parts reference large binaries and the large binaries are no longer used in any process, deliberate copying might be a good idea.

If $N < 0$, a badarg exception is raised.

decode_unsigned(Subject) -> Unsigned
Types:
  Subject = binary()
  Unsigned = integer() >= 0
The same as decode_unsigned(Subject, big).

decode_unsigned(Subject, Endianess) -> Unsigned
Types:
  Subject = binary()
  Endianess = big | little
  Unsigned = integer() >= 0
Converts the binary digit representation, in big or little endian, of a positive integer in Subject to an Erlang integer().
Example:

1> binary:decode_unsigned(<169,138,199>,big).
11111111

encode_unsigned(Unsigned) -> binary()
Types:
  Unsigned = integer() >= 0
The same as encode_unsigned(Unsigned, big).

encode_unsigned(Unsigned, Endianess) -> binary()
Types:
  Unsigned = integer() >= 0
  Endianess = big | little
Converts a positive integer to the smallest possible representation in a binary digit representation, either big or little endian.
Example:

1> binary:encode_unsigned(11111111,big).
<<169,138,199>>
first(Subject) -> byte()
Types:
    Subject = binary()
Returns the first byte of the binary Subject as an integer. If the size of Subject is zero, a badarg exception is raised.

last(Subject) -> byte()
Types:
    Subject = binary()
Returns the last byte of the binary Subject as an integer. If the size of Subject is zero, a badarg exception is raised.

list_to_bin(ByteList) -> binary()
Types:
    ByteList = iodata()
Works exactly as erlang:list_to_binary/1, added for completeness.

longest_common_prefix(Binaries) -> integer() >= 0
Types:
    Binaries = [binary()]
Returns the length of the longest common prefix of the binaries in the list Binaries. Example:

    1> binary:longest_common_prefix([<<"erlang">>,<<"ergonomy">>]).
    2> binary:longest_common_prefix([<<"erlang">>,<<"perl">>]).
    0

If Binaries is not a flat list of binaries, a badarg exception is raised.

longest_common_suffix(Binaries) -> integer() >= 0
Types:
    Binaries = [binary()]
Returns the length of the longest common suffix of the binaries in the list Binaries. Example:

    1> binary:longest_common_suffix([<<"erlang">>,<<"fang">>]).
    2> binary:longest_common_suffix([<<"erlang">>,<<"perl">>]).
    0

If Binaries is not a flat list of binaries, a badarg exception is raised.
The same as `match(Subject, Pattern, [])`.

`match(Subject, Pattern, Options) -> Found | nomatch`

Types:

- `Subject` = `binary()`
- `Pattern` = `binary() | [binary()] | cp()`
- `Found` = `part()`
- `Options` = `[Option]`
- `Option` = `{scope, part()}`
- `part()` = `{Start :: integer() >= 0, Length :: integer()}`

Searches for the first occurrence of `Pattern` in `Subject` and returns the position and length.

The function will return `{Pos, Length}` for the binary in `Pattern` starting at the lowest position in `Subject`.

Example:

```
1> binary:match(<<"abcde">>, [<<"bcde">>,<<"cd">>],[]).
{1,4}
```

Even though `<<"cd">>` ends before `<<"bcde">>, <<"bcde">>` begins first and is therefore the first match. If two overlapping matches begin at the same position, the longest is returned.

Summary of the options:

- `{scope, {Start, Length}}`

  Only the given part is searched. Return values still have offsets from the beginning of `Subject`. A negative `Length` is allowed as described in the DATA TYPES section of this manual.

If none of the strings in `Pattern` is found, the atom `nomatch` is returned.

For a description of `Pattern`, see `compile_pattern/1`.

If `{scope, {Start, Length}}` is given in the options such that `Start` is larger than the size of `Subject`, `Start + Length` is less than zero or `Start + Length` is larger than the size of `Subject`, a badarg exception is raised.

`matches(Subject, Pattern) -> Found`

Types:

- `Subject` = `binary()`
- `Pattern` = `binary() | [binary()] | cp()`
- `Found` = `[part()]`

The same as `matches(Subject, Pattern, [])`.

`matches(Subject, Pattern, Options) -> Found`

Types:
Subject = binary()
Pattern = binary() | [binary()] | cp()
Found = [part()]
Options = [Option]
Option = {scope, part()}
  part() = {Start :: integer() >= 0, Length :: integer()}

Works like match/2, but the Subject is searched until exhausted and a list of all non-overlapping parts matching Pattern is returned (in order).
The first and longest match is preferred to a shorter, which is illustrated by the following example:

```erlang
1> binary:matches(<<"abcde">>, [<<"bcde">>, <<"bc">>, <<"de">>], []).
[{1,4}]
```

The result shows that <<"bcde">> is selected instead of the shorter match <<"bc">> (which would have given raise to one more match, <<"de">>). This corresponds to the behavior of posix regular expressions (and programs like awk), but is not consistent with alternative matches in re (and Perl), where instead lexical ordering in the search pattern selects which string matches.

If none of the strings in pattern is found, an empty list is returned.

For a description of Pattern, see compile_pattern/1 and for a description of available options, see match/3.

If {scope, {Start,Length}} is given in the options such that Start is larger than the size of Subject, Start + Length is less than zero or Start + Length is larger than the size of Subject, a badarg exception is raised.

part(Subject, PosLen) -> binary()
Types:
  Subject = binary()
  PosLen = part()

Extracts the part of the binary Subject described by PosLen.

Negative length can be used to extract bytes at the end of a binary:

```erlang
1> Bin = <<1,2,3,4,5,6,7,8,9,10>>.
2> binary:part(Bin, {byte_size(Bin), -5}).
<<6,7,8,9,10>>
```

Note:

part/2 and part/3 are also available in the erlang module under the names binary_part/2 and binary_part/3. Those BIFs are allowed in guard tests.

If PosLen in any way references outside the binary, a badarg exception is raised.
part(Subject, Pos, Len) -> binary()

Types:
Subject = binary()
Pos = integer() >= 0
Len = integer()

The same as part(Subject, {Pos, Len}).

referenced_byte_size(Binary) -> integer() >= 0

Types:
Binary = binary()

If a binary references a larger binary (often described as being a sub-binary), it can be useful to get the size of the actual referenced binary. This function can be used in a program to trigger the use of copy/1. By copying a binary, one might dereference the original, possibly large, binary which a smaller binary is a reference to.

Example:

```erlang
store(Binary, GBSet) ->
    NewBin =
    case binary:referenced_byte_size(Binary) of
        Large when Large > 2 * byte_size(Binary) ->
            binary:copy(Binary);
        _ ->
            Binary
    end,
    gb_sets:insert(NewBin, GBSet).
```

In this example, we chose to copy the binary content before inserting it in the gb_set() if it references a binary more than twice the size of the data we're going to keep. Of course different rules for when copying will apply to different programs.

Binary sharing will occur whenever binaries are taken apart, this is the fundamental reason why binaries are fast, decomposition can always be done with O(1) complexity. In rare circumstances this data sharing is however undesirable, why this function together with copy/1 might be useful when optimizing for memory use.

Example of binary sharing:

```
1> A = binary:copy(<1>,100).
<1,1,1,1,1 ... 
2> byte_size(A).
100
3> binary:referenced_byte_size(A)
100
4> << :10/binary,B:10/binary,_/binary>> = A.
<<1,1,1,1,1 ... 
5> byte_size(B).
10
6> binary:referenced_byte_size(B)
100
```
Note:
Binary data is shared among processes. If another process still references the larger binary, copying the part this process uses only consumes more memory and will not free up the larger binary for garbage collection. Use this kind of intrusive functions with extreme care, and only if a real problem is detected.

replace(Subject, Pattern, Replacement) -> Result
Types:
  Subject = binary()
  Pattern = binary() | [binary()] | cp()
  Replacement = Result = binary()
The same as replace(Subject,Pattern,Replacement,[]).

replace(Subject, Pattern, Replacement, Options) -> Result
Types:
  Subject = binary()
  Pattern = binary() | [binary()] | cp()
  Replacement = binary()
  Options = [Option]
  Option = global | {scope, part()} | {insert_replaced, InsPos}
  InsPos = OnePos | [OnePos]
  OnePos = integer() >= 0
  An integer() =< byte_size(Replacement)
  Result = binary()
Constructs a new binary by replacing the parts in Subject matching Pattern with the content of Replacement. If the matching sub-part of Subject giving raise to the replacement is to be inserted in the result, the option (insert_replaced, InsPos) will insert the matching part into Replacement at the given position (or positions) before actually inserting Replacement into the Subject. Example:

1> binary:replace(<"abcde">,<"b">,<"[]">, [{insert_replaced,1}]).
<"a[b]cde">
2> binary:replace(<"abcde">,[<"b">,<"d">],<"[]">, [
  global, {insert_replaced,1}]).
<"a[b]c[d]e">
3> binary:replace(<"abcde">,[<"b">,<"d">],<"[]">, [
  global, {insert_replaced,[1,1]}]).
<"a[bb]c[dd]e">
4> binary:replace(<"abcde">,[<"b">,<"d">],<"[-]">, [
  global, {insert_replaced,[1,2]}]).
<"a[bb-b]c[dd]e">
If any position given in InsPos is greater than the size of the replacement binary, a badarg exception is raised.
The options global and {scope, part()} work as for split/3. The return type is always a binary(). For a description of Pattern, see compile_pattern/1.
split(Subject, Pattern) -> Parts
Types:
   Subject = binary()
   Pattern = binary() | [binary()] | cp()
   Parts = [binary()]
The same as split(Subject, Pattern, []).

split(Subject, Pattern, Options) -> Parts
Types:
   Subject = binary()
   Pattern = binary() | [binary()] | cp()
   Options = [Option]
   Option = {scope, part()} | trim | global
   Parts = [binary()]
Splits Subject into a list of binaries based on Pattern. If the option global is not given, only the first occurrence of Pattern in Subject will give rise to a split. The parts of Pattern actually found in Subject are not included in the result.
Example:

1> binary:split(<<1,255,4,0,0,0,2,3>>, [<<0,0,0>>,<<2>>], []).[<<1,255,4>>, <<2,3>>]
2> binary:split(<<0,1,0,0,4,255,255,9>>, [<<0,0>>, <<255,255>>], [global]).[<<0,1>>,<<4>>,<<9>>]

Summary of options:
{scope, part()}
   Works as in match/3 and matches/3. Note that this only defines the scope of the search for matching strings, it does not cut the binary before splitting. The bytes before and after the scope will be kept in the result. See example below.
trim
   Removes trailing empty parts of the result (as does trim in re:split/3)
global
   Repeats the split until the Subject is exhausted. Conceptually the global option makes split work on the positions returned by matches/3, while it normally works on the position returned by match/3.
Example of the difference between a scope and taking the binary apart before splitting:

1> binary:split(<<"banana">>, [<<"a">>], [{scope, {2,3}}]).[<<"ban">>,<<"na">>]
2> binary:split(binary:part(<<"banana">>, {2,3}), [<<"a">>], []).[<<"n">>,<<"n">>]

The return type is always a list of binaries that are all referencing Subject. This means that the data in Subject is not actually copied to new binaries and that Subject cannot be garbage collected until the results of the split are no longer referenced.
For a description of Pattern, see compile_pattern/1.
The \texttt{c} module enables users to enter the short form of some commonly used commands.

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

\subsection*{Exports}

\begin{itemize}
\item \texttt{bt(Pid) -> ok | undefined}
\textit{Types:}
\begin{itemize}
\item \texttt{Pid = pid()}
\end{itemize}
Stack backtrace for a process. Equivalent to \texttt{erlang:process_display(Pid, backtrace)}.

\item \texttt{c(File) -> \{ok, Module\} | error}
\texttt{c(File, Options) -> \{ok, Module\} | error}
\textit{Types:}
\begin{itemize}
\item \texttt{File = file:name()}
\item \texttt{Options = [compile:option()]}\newpage
\item \texttt{Module = module()}
\end{itemize}
c/1,2 compiles and then purges and loads the code for a file. Options defaults to \texttt{[]}. Compilation is equivalent to:

\begin{verbatim}
compile:file(, ++ [report_errors, report_warnings])
\end{verbatim}

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

\item \texttt{cd(Dir) -> ok}
\textit{Types:}
\begin{itemize}
\item \texttt{Dir = file:name()}
\end{itemize}
Changes working directory to \texttt{Dir}, which may be a relative name, and then prints the name of the new working directory.

\begin{verbatim}
2> cd("../erlang").
/home/ron/erlang
\end{verbatim}

\item \texttt{flush() -> ok}
Flushes any messages sent to the shell.
\end{itemize}
help() -> ok
Displays help information: all valid shell internal commands, and commands in this module.

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

i(X, Y, Z) -> [{atom(), term()}]
Types:
  X = Y = Z = integer() >= 0
Displays information about a process, Equivalent to process_info(pid(X, Y, Z)), but location transparent.

l(Module) -> code:load_ret()
Types:
  Module = module()
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 = file:filename()
Compiles a list of files by calling compile:file(File, [report_errors, report_warnings]) for each File in Files.

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

ls(Dir) -> ok
Types:
  Dir = file:name()
Lists files in directory Dir or, if Dir is a file, only list it.

m() -> ok
Displays information about the loaded modules, including the files from which they have been loaded.

m(Module) -> ok
Types:
  Module = module()
Displays information about Module.
memory() -> [{Type, Size}]
Types:
  Type = atom()
  Size = integer() >= 0
Memory allocation information. Equivalent to erlang:memory/0.

memory(Type) -> Size
memory(Types) -> [{Type, Size}]
Types:
  Types = [Type]
  Type = atom()
  Size = integer() >= 0
Memory allocation information. Equivalent to erlang:memory/1.

nc(File) -> {ok, Module} | error
nc(File, Options) -> {ok, Module} | error
Types:
  File = filename()
  Options = [Option] | Option
  Option = compile:option()
  Module = module()
Compiles and then loads the code for a file on all nodes. Options defaults to []. Compilation is equivalent to:

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

nl(Module) -> abcast | error
Types:
  Module = module()
Loads Module on all nodes.

pid(X, Y, Z) -> pid()
Types:
  X = Y = Z = integer() >= 0
Converts X, Y, Z to the pid <X.Y.Z>. This function should only be used when debugging.

pwd() -> ok
Prints the name of the working directory.

q() -> no_return()
This function is shorthand for init:stop(), that is, it causes the node to stop in a controlled fashion.
regs() -> ok
nregs() -> ok
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:

y(File, Options) -> YeccRet
Types:
   File = name() -- see filename(3)
   Options, YeccRet = -- see yecc:file/2
Generates an LALR-1 parser. Equivalent to:

See Also
compile(3), filename(3), erlang(3), yecc(3), xref(3)
This module provides computation of local and universal time, day-of-the-week, and several time conversion functions. Time is local when it is adjusted in accordance with the current time zone and daylight saving. Time is universal when it reflects the time at longitude zero, without any adjustment for daylight saving. Universal Coordinated Time (UTC) time is also called Greenwich Mean Time (GMT).

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

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

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

There exists different definitions for the week of the year. The calendar module contains a week of the year implementation which conforms to the ISO 8601 standard. Since the week number for a given date can fall on the previous, the current or on the next year it is important to provide the information which year is it together with the week number. The function `iso_week_number/0` and `iso_week_number/1` returns a tuple of the year and the week number.

Data Types

```erlang
datetime() = {date(), time()}
datetime1970() = {[year1970()], month(), day()}, time()
date() = {year(), month(), day()}
year() = integer() >= 0
```

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.
year1970() = 1970..10000
month() = 1..12
day() = 1..31
time() = \{\text{hour()}, \text{minute()}, \text{second()}\}
hour() = 0..23
minute() = 0..59
second() = 0..59
daynum() = 1..7
ldom() = 28 | 29 | 30 | 31
yearweeknum() = \{\text{year()}, \text{weeknum()}\}
weeknum() = 1..53

Exports

date_to_gregorian_days(Date) -> Days
date_to_gregorian_days(Year, Month, Day) -> Days
Types:
\begin{align*}
\text{Date} & = \text{date()} \\
\text{Year} & = \text{year()} \\
\text{Month} & = \text{month()} \\
\text{Day} & = \text{day()}
\end{align*}

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

datetime_to_gregorian_seconds(DateTime) -> Seconds
Types:
\begin{align*}
\text{DateTime} & = \text{datetime()} \\
\text{Seconds} & = \text{integer()} \geq 0
\end{align*}

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) -> daynum()
day_of_the_week(Year, Month, Day) -> daynum()
Types:
\begin{align*}
\text{Date} & = \text{date()} \\
\text{Year} & = \text{year()} \\
\text{Month} & = \text{month()} \\
\text{Day} & = \text{day()}
\end{align*}

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:
\begin{align*}
\text{Days} & = \text{integer()} \geq 0
\end{align*}

This function computes the date given the number of gregorian days.
gregorian_seconds_to_datetime(Seconds) -> datetime()
Types:
   Seconds = integer() >= 0
This function computes the date and time from the given number of gregorian seconds.

is_leap_year(Year) -> boolean()
Types:
   Year = year()
This function checks if a year is a leap year.

iso_week_number() -> yearweeknum()
This function returns the tuple {Year, WeekNum} representing the iso week number for the actual date. For
determining the actual date, the function local_time/0 is used.

iso_week_number(Date) -> yearweeknum()
Types:
   Date = date()
This function returns the tuple {Year, WeekNum} representing the iso week number for the given date.

last_day_of_the_month(Year, Month) -> LastDay
Types:
   Year = year()
   Month = month()
   LastDay = ldom()
This function computes the number of days in a month.

local_time() -> datetime()
This function returns the local time reported by the underlying operating system.

local_time_to_universal_time(DateTime1) -> DateTime2
Types:
   DateTime1 = DateTime2 = datetime1970()
This function converts from local time to Universal Coordinated Time (UTC). DateTime1 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(DateTime1) -> [DateTime]
Types:
  DateTime1 = DateTime = datetime1970()
This function converts from local time to Universal Coordinated Time (UTC). DateTime1 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) -> datetime1970()
Types:
  Now = erlang:timestamp()
This function returns local date and time converted from the return value from erlang:now().

now_to_universal_time(Now) -> datetime1970()
now_to_datetime(Now) -> datetime1970()
Types:
  Now = erlang:timestamp()
This function returns Universal Coordinated Time (UTC) converted from the return value from erlang:now().

seconds_to_daystime(Seconds) -> {Days, Time}
Types:
  Seconds = Days = integer()
  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 = secs_per_day() = 0..86400
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}
Types:
T1 = T2 = \texttt{datetime()}
Days = \texttt{integer()}
Time = \texttt{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) \rightarrow \texttt{secs\_per\_day()}

Types:
- Time = \texttt{time()}
- secs\_per\_day() = 0..86400

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

universal_time() \rightarrow \texttt{datetime()}

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(DateTime) \rightarrow \texttt{datetime()}

Types:
- DateTime = \texttt{datetime1970()}

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

valid_date(Date) \rightarrow \texttt{boolean()}
valid_date(Year, Month, Day) \rightarrow \texttt{boolean()}

Types:
- Date = \texttt{date()}
- Year = Month = Day = \texttt{integer()}

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 \texttt{localtime/0}. Universal time is computed from the BIF \texttt{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 \texttt{dets} provides a term storage on file. The stored terms, in this module called \textit{objects}, are tuples such that one element is defined to be the key. A Dets \textit{table} is a collection of objects with the key at the same position stored on a file.

Dets is used by the Mnnesia 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. Mnnesia adds transactions, queries, and distribution. The size of Dets files cannot exceed 2 GB. If larger tables are needed, Mnnesia's table fragmentation can be used.

There are three types of Dets tables: \texttt{set}, \texttt{bag} and \texttt{duplicate\_bag}. A table of type \texttt{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 \texttt{bag} has zero or more different objects with a given key. A table of type \texttt{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.

\textbf{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 \texttt{repair} option set to \texttt{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 \texttt{first} and \texttt{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 Mnnesia 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 \{\texttt{error}, \texttt{Reason}\} if an error occurs (\texttt{first}/1 and \texttt{next}/2 are exceptions, they exit the process with the error tuple). If given badly formed arguments, all functions exit the process with a \texttt{badarg} message.
Data Types

access() = read | read_write
auto_save() = infinity | integer() >= 0
bindings_cont()
Opaque continuation used by match/1 and match/3.
cont()
Opaque continuation used by bchunk/2.
keypos() = integer() >= 1
match_spec() = ets:match_spec()
Match specifications, see the match specification documentation in the ERTS User's Guide and ms_transform(3).
no_slots() = integer() >= 0 | default
object() = tuple()
object_cont()
Opaque continuation used by match_object/1 and match_object/3.
pattern() = atom() | tuple()
See ets:match/2 for a description of patterns.
select_cont()
Opaque continuation used by select/1 and select/3.
tab_name() = term()
type() = bag | duplicate_bag | set
version() = 8 | 9 | default

Exports

all() -> [tab_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 = tab_name()
    Continuation = start | cont()
    Continuation2 = cont()
    Data = binary() | tuple()
    Reason = term()
Returns a list of objects stored in a table. The exact representation of the returned objects is not public. The lists of data can be used for initializing a table by giving the value bchunk to the format option of the init_table/3 function. The Mnesia application uses this function for copying open tables.
Unless the table is protected using safe_fixtable/2, calls to bchunk/2 may not work as expected if concurrent updates are made to the table.

Ericsson AB. All Rights Reserved.: STDLIB | 69
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 \{Continuation2, Data\}, where Data is a list of objects. 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 \{error, Reason\} if an error occurs.

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

Types:
\begin{itemize}
\item Name = \texttt{tab_name()}
\item Reason = \texttt{term()}
\end{itemize}

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.

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

Types:
\begin{itemize}
\item Name = \texttt{tab_name()}
\item Key = \texttt{term()}
\end{itemize}

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

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

Types:
\begin{itemize}
\item Name = \texttt{tab_name()}
\item Reason = \texttt{term()}
\end{itemize}

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, '_')}.

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

Types:
\begin{itemize}
\item Name = \texttt{tab_name()}
\item Object = \texttt{object()}
\item Reason = \texttt{term()}
\end{itemize}

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.

\begin{verbatim}
first(Name) -> Key | '$end_of_table'
\end{verbatim}

Types:
\begin{itemize}
\item Name = \texttt{tab_name()}
\item Key = \texttt{term()}
\end{itemize}

Returns the first key stored in the table \texttt{Name} according to the table's internal order, or \texttt{'$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) -> Acc | {error, Reason}
foldr(Function, Acc0, Name) -> Acc | {error, Reason}
```

Types:

- `Name = tab_name()`
- `Function = fun((Object :: object(), AccIn) -> AccOut)`
- `Acc0 = Acc = AccIn = AccOut = Reason = term()`

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 = tab_name()`
- `EtsTab = ets:tab()`
- `Reason = term()`

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 = tab_name()`
- `InfoList = [InfoTuple]`
- `InfoTuple = {file_size, integer() >= 0}`
  - `{filename, file:name()}`  
  - `{keypos, keypos()}`  
  - `{size, integer() >= 0}`  
  - `{type, type()}`

Returns information about the table `Name` as a list of tuples:

- `{file_size, integer() >= 0}`, the size of the file in bytes.
- `{filename, file:name()}`, the name of the file where objects are stored.
- `{keypos, keypos()}`, the position of the key.
- `{size, integer() >= 0}`, the number of objects stored in the table.
- `{type, type()}`, the type of the table.
info(Name, Item) -> Value | undefined

Types:

Name = tab_name()
Item = access
| auto_save
| bchunk_format
| hash
| file_size
| filename
| keypos
| memory
| no_keys
| no_objects
| no_slots
| owner
| ram_file
| safe_fixed
| size
| type
| version

Value = term()

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, integer() >= 0\}, the size of the file in bytes. The same value is associated with the item file_size.
- \{no_keys, integer >= 0\}, the number of different keys stored in the table. Only available for version 9 tables.
- \{no_objects, integer >= 0\}, 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, boolean()\}, 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, integer()\}, the version of the format of the table.
init_table(Name, InitFun) -> ok | {error, Reason}
init_table(Name, InitFun, Options) -> ok | {error, Reason}

Types:
   Name = tab_name()
   InitFun = fun((Arg) -> Res)
   Arg = read | close
   Res = end_of_input
       | [{object()}, InitFun]
       | {Data, InitFun}
       | term()
   Options = Option | [Option]
   Option = {min_no_slots, no_slots()} | {format, term | bchunk}
   Reason = term()
   Data = binary() | tuple()

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

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

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

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

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

- {min_no_slots, no_slots()}. Specifies the estimated number of different keys that will be stored in the table. The open_file option with the same name is ignored unless the table is created, and in that case performance can be enhanced by supplying an estimate when initializing the table.

- {format, Format}. Specifies the format of the objects returned by the function InitFun. If Format is term (the default), InitFun is assumed to return a list of tuples. If Format is bchunk, InitFun is assumed to return Data as returned by bchunk/2. This option overrides the min_no_slots option.

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

Types:
   Name = tab_name()
   Objects = object() | [object()]
   Reason = term()

Inserts one or more objects into the table Name. If there already exists an object with a key matching the key of some of the given objects and the table type is set, the old object will be replaced.
insert_new(Name, Objects) -> boolean() | {error, Reason}
Types:
   Name = tab_name()
   Objects = object() | [object()]
   Reason = term()
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) -> boolean()
Types:
   Name = tab_name()
   BchunkFormat = binary()
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) -> boolean() | {error, Reason}
Types:
   Filename = file:name()
   Reason = term()
Returns true if the file Filename is a Dets table, false otherwise.

lookup(Name, Key) -> Objects | {error, Reason}
Types:
   Name = tab_name()
   Key = term()
   Objects = [object()]
   Reason = term()
Returns a list of all objects with the key Key stored in the table Name. For example:

```erlang
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()]
  Reason = 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 = tab_name()
  Pattern = pattern()
  Match = [term()]
  Reason = term()

Returns for each object of the table Name that matches Pattern a list of bindings in some unspecified order. See ets:match/2 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 = tab_name()
  Pattern = pattern()
  N = default | integer() >= 0
  Continuation = bindings_cont()
  Match = [term()]
  Reason = term()

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:match/2 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 = tab_name()
   Pattern = pattern()
   Reason = term()

Deletes all objects that match Pattern from the table Name. See ets:match/2 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) ->
   {Objects, Continuation2} |
   'end_of_table' |
   {error, Reason}
Types:
   Continuation = Continuation2 = object_cont()
   Objects = [object()]
   Reason = term()

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) -> Objects | {error, Reason}
Types:
   Name = tab_name()
   Pattern = pattern()
   Objects = [object()]
   Reason = term()

Returns a list of all objects of the table Name that match Pattern in some unspecified order. See ets:match/2 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) ->
   {Objects, Continuation} |
   'end_of_table' |
   {error, Reason}
Types:
Name = tab_name()
Pattern = pattern()
N = default | integer() >= 0
Continuation = object_cont()
Objects = [object()]
Reason = term()

Matches some or all objects stored in the table Name and returns a non-empty list of the objects that match Pattern in some unspecified order. See ets:match/2 for a description of patterns.

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

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

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

member(Name, Key) -> boolean() | {error, Reason}
Types:
   Name = tab_name()
   Key = Reason = term()

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

next(Name, Key1) -> Key2 | '$end_of_table'
Types:
   Name = tab_name()
   Key1 = Key2 = term()

Returns the key following Key1 in the table Name according to the table's internal order, or '$end_of_table' if there is no next key.
Should an error occur, the process is exited with an error tuple {error, Reason}.
Use first/1 to find the first key in the table.

open_file(Filename) -> {ok, Reference} | {error, Reason}
Types:
   Filename = file:name()
   Reference = reference()
   Reason = term()

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

Types:
  Name = tab_name()
  Args = [OpenArg]
  OpenArg = {access, access()}
          | {auto_save, auto_save()}
          | {estimated_no_objects, integer() >= 0}
          | {file, file:name()}
          | {max_no_slots, no_slots()}
          | {min_no_slots, no_slots()}
          | {keypos, keypos()}
          | {ram_file, boolean()}
          | {repair, boolean() | force}
          | {type, type()}
          | {version, version()}
  Reason = term()

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, no_slots()\}. Equivalent to the min_no_slots option.
- \{file, file:name()\}, 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 as well as 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, boolean()\}, 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 boolean() 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:
   Pid = pid()
   Name = tab_name()
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) -> ok
Types:
Name = \textit{tab\_name()} \\
Fix = \textit{boolean()} \\

If \texttt{Fix} is \texttt{true}, the table \texttt{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 \texttt{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.

\begin{verbatim}
select(Continuation) -> \\
    \{Selection, Continuation2\} | \\
    \'\$end\_of\_table' | \\
    \{error, Reason\}

Types:
    Continuation = Continuation2 = \texttt{select\_cont()} \\
    Selection = \[\text{term()}\] \\
    Reason = \text{term()}
\end{verbatim}

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 \texttt{Continuation}, which has been returned by a prior call to \texttt{select/1} or \texttt{select/3}.

When all objects of the table have been matched, \'\$end\_of\_table' is returned.

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

Types:
    Name = \textit{tab\_name()} \\
    MatchSpec = \texttt{match\_spec()} \\
    Selection = \[\text{term()}\] \\
    Reason = \text{term()}
\end{verbatim}

Returns the results of applying the match specification \texttt{MatchSpec} to all or some 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.

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

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

\begin{verbatim}
select(Name, MatchSpec, N) -> \\
    \{Selection, Continuation\} | \\
    \'\$end\_of\_table' | \\
    \{error, Reason\}

Types:
\end{verbatim}
select_delete(Name, MatchSpec) -> N | {error, Reason}
Types:
  Name = tab_name()
  MatchSpec = match_spec()
  N = integer() >= 0
  Reason = term()

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

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

slot(Name, I) -> '$end_of_table' | Objects | {error, Reason}
Types:
  Name = tab_name()
  I = integer() >= 0
  Objects = [object()]
  Reason = term()

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

call{Name} -> ok | {error, Reason}
Types:
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.

\texttt{table(Name)} \to QueryHandle
\texttt{table(Name, Options)} \to QueryHandle

\textbf{Types:}

\begin{verbatim}
Name = \texttt{tab_name()}
Options = Option | [Option]
Option = \{n_objects, Limit\} | \{traverse, TraverseMethod\}
Limit = default | integer() \geq 1
TraverseMethod = first\_next | select | \{select, \texttt{match\_spec()}\}
QueryHandle = \texttt{qlc:query\_handle()}
\end{verbatim}

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

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}}]).

An example with implicit match specification:

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

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

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

82 | Ericsson AB. All Rights Reserved.: STDLIB
true

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

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

Types:
- $Name = tab_name()$
- $EtsTab = ets:tab()$
- $Reason = term()$

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:
- $Name = tab_name()$
- $Fun = fun((Object) -> FunReturn)$
- $Object = object()$
- $FunReturn = continue$
  - $\{continue, Val\}$
  - $\{done, Value\}$
  - OtherValue
- $Return = [term()] | OtherValue$
- $Val = Value = OtherValue = Reason = term()$

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

$continue$

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

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

$\{continue, Val\}$

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

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

$\{done, Value\}$

Terminate the traversal and return $[Value | Acc]$. Any other value $OtherValue$ returned by $Fun$ terminates the traversal and is immediately returned.
update_counter(Name, Key, Increment) -> Result

Types:

Name = tab_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), mnesia(3), qle(3)
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

dict(Key, Value)

Dictionary as returned by new/0.

dict()

dict() is equivalent to dict(term(), term()).

Exports

append(Key, Value, Dict1) -> Dict2

Types:

Dict1 = Dict2 = dict(Key, Value)

This function appends a new Value to the current list of values associated with Key.

append_list(Key, ValList, Dict1) -> Dict2

Types:

Dict1 = Dict2 = dict(Key, Value)

ValList = [Value]

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:

Dict1 = Dict2 = dict(Key, Value)

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

fetch(Key, Dict) -> Value

Types:

Dict = dict(Key, Value)

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

fetch_keys(Dict) -> Keys

Types:
Dict = \texttt{dict}(\texttt{Key}, \texttt{Value} :: \texttt{term()})

Keys = [\texttt{Key}]

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

\textbf{filter(Pred, Dict1) -> Dict2}

Types:
- \texttt{Pred} = \texttt{fun}((\texttt{Key}, \texttt{Value}) \rightarrow \texttt{boolean()})
- \texttt{Dict1} = \texttt{Dict2} = \texttt{dict}(\texttt{Key}, \texttt{Value})

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

\textbf{find(Key, Dict) -> \{\textit{ok, Value}\} | \textit{error}}

Types:
- \texttt{Dict} = \texttt{dict}(\texttt{Key}, \texttt{Value})

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.

\textbf{fold(Fun, Acc0, Dict) -> Acc1}

Types:
- \texttt{Fun} = \texttt{fun}((\texttt{Key}, \texttt{Value}, \texttt{AccIn}) \rightarrow \texttt{AccOut})
- \texttt{Dict} = \texttt{dict}(\texttt{Key}, \texttt{Value})
- \texttt{Acc0} = \texttt{Acc1} = \texttt{AccIn} = \texttt{AccOut} = \texttt{Acc}

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

\textbf{from_list(List) -> Dict}

Types:
- \texttt{Dict} = \texttt{dict}(\texttt{Key}, \texttt{Value})
- \texttt{List} = [{\texttt{Key}, \texttt{Value}]}

This function converts the Key-Value list \texttt{List} to a dictionary.

\textbf{is_key(Key, Dict) -> boolean()}

Types:
- \texttt{Dict} = \texttt{dict}(\texttt{Key}, \texttt{Value} :: \texttt{term()})

This function tests if Key is contained in the dictionary \texttt{Dict}.

\textbf{map(Fun, Dict1) -> Dict2}

Types:
- \texttt{Fun} = \texttt{fun}((\texttt{Key}, \texttt{Value1}) \rightarrow \texttt{Value2})
- \texttt{Dict1} = \texttt{dict}(\texttt{Key}, \texttt{Value1})
- \texttt{Dict2} = \texttt{dict}(\texttt{Key}, \texttt{Value2})

\texttt{map} calls \texttt{Fun} on successive keys and values of \texttt{Dict1} 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)
  Dict1 = dict(Key, Value1)
  Dict2 = dict(Key, Value2)
  Dict3 = dict(Key, Value)

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() -> dict()
This function creates a new dictionary.

size(Dict) -> integer() >= 0
Types:
  Dict = dict()
Returns the number of elements in a Dict.

is_empty(Dict) -> boolean()
Types:
  Dict = dict()
Returns true if Dict has no elements, false otherwise.

store(Key, Value, Dict1) -> Dict2
Types:
  Dict1 = Dict2 = dict(Key, Value)
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 = dict(Key, Value)
  List = [[Key, Value]]
This function converts the dictionary to a list representation.

update(Key, Fun, Dict1) -> Dict2
Types:
**Notes**

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

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

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

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

$gb_trees(3), orddict(3)$
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 V × V (the Cartesian product of V with itself). In this module, V is allowed to be empty; the so obtained unique digraph is called the empty digraph. Both vertices and edges are represented by unique Erlang terms.

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

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

### Data Types

- **d_type()** = `d_cyclicity() | d_protection()`
- **d_cyclicity()** = `acyclic | cyclic`
- **d_protection()** = `private | protected`

- **graph()**
  A digraph as returned by `new/0,1`

- **edge()**
- **label()** = `term()`
- **vertex()**

### Exports

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

**Types:**

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

`add_edge_err_rsn() = {bad_edge, Path :: [{vertex()}]}`
add_edge/5 creates (or modifies) the edge $E$ of the digraph $G$, using Label as the (new) label of the edge. The edge is emanating from $V_1$ and incident on $V_2$. Returns $E$.

$\text{add_edge}(G, V_1, V_2, \text{Label})$ is equivalent to $\text{add_edge}(G, E, V_1, V_2, \text{Label})$, where $E$ is a created edge. The created edge is represented by the term $\['e' | N\]$, where $N$ is an integer $\geq 0$.

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

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

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

Types:
$G = \text{graph()}$
$V = \text{vertex()}$
Label = \text{label()}$

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

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

$\text{add_vertex}/1$ creates a vertex using the empty list as label, and returns the created vertex. The created vertex is represented by the term $\['v' | N\]$, where $N$ is an integer $\geq 0$.

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

Types:
$G = \text{graph()}$
$E = \text{edge()}$

Deletes the edge $E$ from the digraph $G$.

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

Types:
$G = \text{graph()}$
Edges = [\text{edge()}]

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

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

Types:
$G = \text{graph()}$
$V_1 = V_2 = \text{vertex()}$

Deletes edges from the digraph $G$ until there are no paths from the vertex $V_1$ to the vertex $V_2$.

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

Ericsson AB. All Rights Reserved.: STDLIB | 91
del_vertex(G, V) -> true
Types:
   G = graph()
   V = vertex()
Deletes the vertex V from the digraph G. Any edges emanating from V or incident on V are also deleted.

del_vertices(G, Vertices) -> true
Types:
   G = graph()
   Vertices = [vertex()]
Deletes the vertices in the list Vertices from the digraph G.

delete(G) -> true
Types:
   G = graph()
Deletes the digraph G. This call is important because digraphs are implemented with ETS. There is no garbage collection of ETS tables. The digraph will, however, be deleted if the process that created the digraph terminates.

edge(G, E) -> {E, V1, V2, Label} | false
Types:
   G = graph()
   E = edge()
   V1 = V2 = vertex()
   Label = label()
Returns \{E, V1, V2, Label\} where Label is the label of the edge E emanating from V1 and incident on V2 of the digraph G. If there is no edge E of the digraph G, then false is returned.

edges(G) -> Edges
Types:
   G = graph()
   Edges = [edge()]
Returns a list of all edges of the digraph G, in some unspecified order.

edges(G, V) -> Edges
Types:
   G = graph()
   V = vertex()
   Edges = [edge()]
Returns a list of all edges emanating from or incident on V of the digraph G, in some unspecified order.

get_cycle(G, V) -> Vertices | false
Types:
\[
G = \text{graph()}
\]
\[
V = \text{vertex()}
\]
\[
\text{Vertices} = [\text{vertex()}, \ldots]
\]

If there is a simple cycle 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 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, V_1, V_2) \rightarrow \text{Vertices} | \text{false}

Types:

\[
G = \text{graph()}
\]
\[
V_1 = V_2 = \text{vertex()}
\]
\[
\text{Vertices} = [\text{vertex()}, \ldots]
\]

Tries to find a simple path 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 \text{false} if no simple path from \(V_1\) to \(V_2\) 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) \rightarrow \text{Vertices} | \text{false}

Types:

\[
G = \text{graph()}
\]
\[
V = \text{vertex()}
\]
\[
\text{Vertices} = [\text{vertex()}, \ldots]
\]

Tries to find an as short as possible simple cycle 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 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, V_1, V_2) \rightarrow \text{Vertices} | \text{false}

Types:

\[
G = \text{graph()}
\]
\[
V_1 = V_2 = \text{vertex()}
\]
\[
\text{Vertices} = [\text{vertex()}, \ldots]
\]

Tries to find an as short as possible simple path 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 \text{false} if no simple path from \(V_1\) to \(V_2\) of length one or more exists.

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

\text{in-degree}(G, V) \rightarrow \text{integer()} \geq 0

Types:

\[
G = \text{graph()}
\]
\[
V = \text{vertex()}
\]

Returns the in-degree of the vertex \(V\) of the digraph \(G\).
in_edges(G, V) -> Edges
Types:
  G = graph()
  V = vertex()
  Edges = [edge()]
Returns a list of all edges incident on V of the digraph G, in some unspecified order.

in_neighbours(G, V) -> Vertex
Types:
  G = graph()
  V = vertex()
  Vertices = [vertex()]
Returns a list of all in-neighbours of V of the digraph G, in some unspecified order.

info(G) -> InfoList
Types:
  G = graph()
  InfoList = [{cyclicity, Cyclicity :: d_cyclicity()} |
                {memory, NoWords :: integer() >= 0} |
                {protection, Protection :: d_protection()}]
  d_cyclicity() = acyclic | cyclic
  d_protection() = private | protected
Returns a list of {Tag, Value} pairs describing the digraph G. The following pairs are returned:
  • {cyclicity, Cyclicity}, where Cyclicity is cyclic or acyclic, according to the options given
to new.
  • {memory, NoWords}, where NoWords is the number of words allocated to the ETS tables.
  • {protection, Protection}, where Protection is protected or private, according to the options
given to new.

new() -> graph()
Equivalent to new([]).

new(Type) -> graph()
Types:
  Type = [d_type()]
  d_type() = d_cyclicity() | d_protection()
  d_cyclicity() = acyclic | cyclic
  d_protection() = private | protected
Returns an empty digraph with properties according to the options in Type:
  cyclic
    Allow cycles in the digraph (default).
  acyclic
    The digraph is to be kept acyclic.
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:
    $G = \text{graph()}$
Returns the number of edges of the digraph $G$.

no_vertices(G) -> integer() >= 0
Types:
    $G = \text{graph()}$
Returns the number of vertices of the digraph $G$.

out_degree(G, V) -> integer() >= 0
Types:
    $G = \text{graph()}$
    $V = \text{vertex()}$
Returns the out-degree of the vertex $V$ of the digraph $G$.

out_edges(G, V) -> Edges
Types:
    $G = \text{graph()}$
    $V = \text{vertex()}$
    $\text{Edges} = [\text{edge()}]$
Returns a list of all edges emanating from $V$ of the digraph $G$, in some unspecified order.

out_neighbours(G, V) -> Vertices
Types:
    $G = \text{graph()}$
    $V = \text{vertex()}$
    $\text{Vertices} = [\text{vertex()}]$
Returns a list of all out-neighbours of $V$ of the digraph $G$, in some unspecified order.

vertex(G, V) -> {V, Label} | false
Types:
    $G = \text{graph()}$
    $V = \text{vertex()}$
    $\text{Label} = \text{label()}$
Returns \{V, Label\} where Label is the label of the vertex $V$ of the digraph $G$, or false if there is no vertex $V$ of the digraph $G$. 
vertices(G) -> Vertices

Types:
   G = graph()
   Vertices = [vertex()]

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

See Also

digraph_utils(3), ets(3)
digraph_utils

Erlang module

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

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

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

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

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

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

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.

Data Types
digraph()
A digraph as returned by `digraph:new/0,1`.

Exports

```
arborescence_root(Digraph) -> no | {yes, Root}
```
Types:
  Digraph = `digraph:graph()`
  Root = `digraph:vertex()`
Returns `{yes, Root}` if `Root` is the root of the arborescence `Digraph`, no otherwise.
components(Digraph) -> [Component]
Types:
   Digraph = digraph:graph()
   Component = [digraph:vertex()]
Returns a list of connected components. 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:graph()
Creates a digraph where the vertices are the strongly connected components of Digraph as returned by strong_components/1. If X and Y are two different strongly connected components, and there exist vertices x and y in X and Y respectively such that there is an edge emanating from x and incident 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 [].
Each and every cycle is included in some strongly connected component, which implies that there always exists a topological ordering of the created digraph.

cyclic_strong_components(Digraph) -> [StrongComponent]
Types:
   Digraph = digraph:graph()
   StrongComponent = [digraph:vertex()]
Returns a list of strongly connected components. Each strongly component is represented by its vertices. The order of the vertices and the order of the components are arbitrary. Only vertices that are included in some cycle in Digraph are returned, otherwise the returned list is equal to that returned by strong_components/1.

is_acyclic(Digraph) -> boolean()
Types:
   Digraph = digraph:graph()
Returns true if and only if the digraph Digraph is acyclic.

is_arborescence(Digraph) -> boolean()
Types:
   Digraph = digraph:graph()
Returns true if and only if the digraph Digraph is an arborescence.

is_tree(Digraph) -> boolean()
Types:
   Digraph = digraph:graph()
Returns true if and only if the digraph Digraph is a tree.

loop_vertices(Digraph) -> Vertices
Types:
Digraph = `digraph:graph()`
Vertices = `[digraph:vertex()]`

Returns a list of all vertices of Digraph that are included in some loop.

\[\text{postorder(Digraph)} \to \text{Vertices}\]

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

Returns all vertices of the digraph Digraph. The order is given by a depth-first traversal 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.

\[\text{preorder(Digraph)} \to \text{Vertices}\]

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

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

\[\text{reachable(Vertices, Digraph)} \to \text{Reachable}\]

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

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path 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.

\[\text{reachable_neighbours(Vertices, Digraph)} \to \text{Reachable}\]

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

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path 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 are returned.

\[\text{reaching(Vertices, Digraph)} \to \text{Reaching}\]

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

Returns an unsorted list of digraph vertices such that for each vertex in the list, there is a path 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) -> Reaching
Types:
   Digraph = digraph:graph()
   Vertices = Reaching = [digraph:vertex()]

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

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

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

subgraph(Digraph, Vertices) -> SubGraph
subgraph(Digraph, Vertices, Options) -> SubGraph
Types:
   Digraph = SubGraph = digraph:graph()
   Vertices = [digraph:vertex()]
   Options = [{type, SubgraphType} | {keep_labels, boolean()}]
   SubgraphType = inherit | [digraph:d_type()]

Creates a maximal subgraph of Digraph having as vertices those vertices of Digraph that are mentioned in Vertices.

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

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

subgraph(Digraph, Vertices) is equivalent to subgraph(Digraph, Vertices, []). There will be a badarg exception if any of the arguments are invalid.

topsort(Digraph) -> Vertices | false
Types:
   Digraph = digraph:graph()
   Vertices = [digraph:vertex()]

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

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

The Erlang source file encoding is selected by a comment in one of the first two lines of the source file. The first string that matches the regular expression \( \text{coding}: \s*([a-zA-Z0-9]+) \) selects the encoding. If the matching string is not a valid encoding it is ignored. The valid encodings are Latin-1 and UTF-8 where the case of the characters can be chosen freely. Examples:

```erlang
%% coding: utf-8

%% For this file we have chosen encoding = Latin-1

%% -*- coding: latin-1 -*-
```

**Data Types**

- `macros() = [atom() | {atom(), term()}]`
- `epp_handle() = pid()`
- `source_encoding() = latin1 | utf8`

**Exports**

- `open(Options) -> {ok, Epp} | {ok, Epp, Extra} | {error, ErrorDescriptor}`

Types:

```erlang
Options = [{{default_encoding, DefEncoding :: source_encoding()} | [includes, IncludePath :: [DirectoryName :: file:name()]] | {macros, PredefMacros :: macros()} | {name, FileName :: file:name()} | extra]

Epp = epp_handle()

Extra = [{encoding, source_encoding()} | none]}

ErrorDescriptor = term()
```

Opens a file for preprocessing.

If `extra` is given in `Options`, the return value will be `{ok, Epp, Extra}` instead of `{ok, Epp}`.

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

Types:
  FileName = file:name()
  IncludePath = [DirectoryName :: file:name()]
  PredefMacros = macros()
  Epp = epp_handle()
  ErrorDescriptor = term()

Equivalent to: epp:open([{name, FileName}, {includes, IncludePath}, {macros, PredefMacros}]).

close(Epp) -> ok

Types:
  Epp = epp_handle()

Closes the preprocessing of a file.

close(Epp) -> ok

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

Types:
  Epp = epp_handle()
  AbsForm = erl_parse:abstract_form()
  Line = erl_scan:line()
  ErrorInfo = erl_scan:error_info() | erl_parse:error_info()

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.
Preprocesses and parses an Erlang source file. Note that the tuple \( \{\text{eof}, \text{Line}\} \) returned at end-of-file is included as a "form".

If extra is given in Options, the return value will be \( \{\text{ok}, \{\text{Form}\}, \text{Extra}\} \) instead of \( \{\text{ok}, \{\text{Form}\}\} \).

\[
\text{parse_file(FileName, IncludePath, PredefMacros)} \rightarrow \{\text{ok, \{Form\}}\} | \{\text{error, OpenError}\}
\]

Types:
- FileName = \( \text{file:name()} \)
- IncludePath = \[DirectoryName :: \text{file:name()}\]
- Form = \( \text{erl_parse:abstract_form()} \) | \{error, ErrorInfo\} | \{eof, Line\}
- PredefMacros = \text{macros()}
- Line = \( \text{erl_scan:line()} \)
- ErrorInfo = \( \text{erl_scan:error_info()} \) | \text{erl_parse:error_info()}\)
- OpenError = \( \text{file:posix()} \) | \text{badarg} | \text{system_limit}\)

Equivalent to \( \text{epp:parse_file(FileName, \{includes, IncludePath\}, \{macros, PredefMacros\})} \).

\[
\text{default_encoding()} \rightarrow \text{source_encoding()}
\]

Returns the default encoding of Erlang source files.

\[
\text{encoding_to_string(Encoding)} \rightarrow \text{string()}
\]

Types:
- Encoding = \( \text{source_encoding()} \)

Returns a string representation of an encoding. The string is recognized by \text{read_encoding}/1,2, \text{read_encoding_from_binary}/1,2, and \text{set_encoding}/1,2 as a valid encoding.

\[
\text{read_encoding(FileName)} \rightarrow \text{source_encoding()} | \text{none}
\]

\[
\text{read_encoding(FileName, Options)} \rightarrow \text{source_encoding()} | \text{none}
\]

Types:
- FileName = \( \text{file:name()} \)
- Options = \[Option\]
  - Option = \{in_comment_only, boolean()\}

Read the \text{encoding} from a file. Returns the read encoding, or \text{none} if no valid encoding was found.
The option `in_comment_only` is `true` by default, which is correct for Erlang source files. If set to `false` the encoding string does not necessarily have to occur in a comment.

```erlang
read_encoding_from_binary(Binary) -> source_encoding() | none
read_encoding_from_binary(Binary, Options) ->
  source_encoding() | none
```

**Types:**
- `Binary = binary()`
- `Options = [Option]`
- `Option = {in_comment_only, boolean()}`

Read the encoding from a binary. Returns the read encoding, or `none` if no valid encoding was found.

The option `in_comment_only` is `true` by default, which is correct for Erlang source files. If set to `false` the encoding string does not necessarily have to occur in a comment.

```erlang
set_encoding(File) -> source_encoding() | none
```

**Types:**
- `File = io:device()`

Reads the encoding from an IO device and sets the encoding of the device accordingly. The position of the IO device referenced by `File` is not affected. If no valid encoding can be read from the IO device the encoding of the IO device is set to the default encoding.

Returns the read encoding, or `none` if no valid encoding was found.

```erlang
set_encoding(File, Default) -> source_encoding() | none
```

**Types:**
- `Default = source_encoding()`
- `File = io:device()`

Reads the encoding from an IO device and sets the encoding of the device accordingly. The position of the IO device referenced by `File` is not affected. If no valid encoding can be read from the IO device the encoding of the IO device is set to the encoding given by `Default`.

Returns the read encoding, or `none` if no valid encoding was found.

```erlang
format_error(ErrorDescriptor) -> io_lib:chars()
```

**Types:**
- `ErrorDescriptor = term()`

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:
Module:format_error(ErrorDescriptor)

See Also

erl_parse(3)
This module provides an interpreter for Erlang expressions. The expressions are in the abstract syntax as returned by `erl_parse`, the Erlang parser, or `io`.

**Data Types**

- `bindings() = [{name(), value()}]`  
  A binding structure.
- `expression() = erl_parse:abstract_expr()`  
  A binding structure.
- `expressions() = [erl_parse:abstract_expr()]`  
  As returned by `erl_parse:parse_exprs/1` or `io:parse_erl_exprs/2`.
- `expression_list() = [expression()]`  
  Further described below.
- `func_spec() = {Module :: module(), Function :: atom()}`  
  Further described below.
- `lfun_eval_handler() = fun((Name :: atom(), Arguments :: expression_list(), Bindings :: binding_struct()) -> {value, Value :: value(), NewBindings :: binding_struct()})`  
  Further described below.
- `lfun_value_handler() = fun((Name :: atom(), Arguments :: [term()]) -> Value :: value())`  
  Further described below.
- `local_function_handler() = {value, lfun_value_handler()} | {eval, lfun_eval_handler()} | none`  
  Further described below.
- `nlfun_handler() = fun((FuncSpec :: func_spec(), Arguments :: [term()]) -> term())`  
  Further described below.
- `value() = term()`  
  Further described below.

**Exports**

- `exprs(Expressions, Bindings) -> {value, Value, NewBindings}`
- `exprs(Expressions, Bindings, LocalFunctionHandler) -> {value, Value, NewBindings}`
- `exprs(Expressions,`
expr(Expression, Bindings) -> {value, Value, NewBindings}
expr(Expression, Bindings, LocalFunctionHandler) ->
    {value, Value, NewBindings}
expr(Expression, Bindings, LocalFunctionHandler, NonLocalFunctionHandler) ->
    {value, Value, NewBindings}
expr(Expression, Bindings, LocalFunctionHandler, NonLocalFunctionHandler, ReturnFormat) ->
    {value, Value, NewBindings} | Value

eval_exprs(Expressions, Bindings) ->
    {value, Value, NewBindings}

eval_exprs(Expressions, Bindings, LocalFunctionHandler, NonLocalFunctionHandler) ->
    {value, Value, NewBindings}

eval_exprs(Expressions, Bindings, LocalFunctionHandler, NonLocalFunctionHandler, ReturnFormat) ->
    {value, Value, NewBindings}

Types:
    Expression = expression()
    Bindings = binding_struct()
    LocalFunctionHandler = local_function_handler()
    NonLocalFunctionHandler = non_local_function_handler()
    ReturnFormat = none | value
    Value = value()
    NewBindings = binding_struct()

Evaluates Expression with the set of bindings Bindings. Expression is an expression in abstract syntax. See below for an explanation of how and when to use the arguments LocalFunctionHandler and NonLocalFunctionHandler.

Returns (value, Value, NewBindings) by default. But if the ReturnFormat is value only the Value is returned.

eval_expr_list(ExpressionList, Bindings) -> {ValueList, NewBindings}
eval_expr_list(ExpressionList, Bindings, LocalFunctionHandler) ->

Bindings,
    LocalFunctionHandler,
    NonLocalFunctionHandler) ->
    {value, Value, NewBindings}

Types:
    Expressions = expressions()
    Bindings = binding_struct()
    LocalFunctionHandler = local_function_handler()
    NonLocalFunctionHandler = non_local_function_handler()
    Value = value()
    NewBindings = binding_struct()

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_expr/2. See below for an explanation of how and when to use the arguments LocalFunctionHandler and NonLocalFunctionHandler.

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

Types:
    ExpressionList = expression_list()
    Bindings = binding_struct()
    LocalFunctionHandler = local_function_handler()
    NonLocalFunctionHandler = non_local_function_handler()
    ValueList = [value()]
    NewBindings = binding_struct()

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() -> binding_struct()

Returns an empty binding structure.

bindings(BindingStruct :: binding_struct()) -> bindings()

Returns the list of bindings contained in the binding structure.

binding(Name, BindingStruct) -> \{value, value()\} | unbound

Types:
    Name = name()
    BindingStruct = binding_struct()

Returns the binding of Name in BindingStruct.

add_binding(Name, Value, BindingStruct) -> binding_struct()

Types:
    Name = name()
    Value = value()
    BindingStruct = binding_struct()

Adds the binding Name = Value to BindingStruct. Returns an updated binding structure.

del_binding(Name, BindingStruct) -> binding_struct()

Types:
    Name = name()
    BindingStruct = binding_struct()

Removes the binding of Name in BindingStruct. 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.
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}`).

Calls to functions defined by evaluating fun expressions "fun ... end" are also hidden from non-local function handlers.

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
Undocumented functions in `erl_eval` should not be used.
erl_expand_records

Erlang module

Exports

module(AbsForms, CompileOptions) -> AbsForms

Types:
  AbsForms = [erl_parse:abstract_form()]
  CompileOptions = [compile:option()]

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 \( \text{parse_transform(Module)} \) is passed to the compiler, a user written function \( \text{parse_transform/2} \) is called by the compiler before the code is checked for errors.

Exports

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

Types:

\[
\text{Forms} = [\text{erl_parse:abstract_form()}]
\]

\[
\text{Options} = [\text{compile:option()}]
\]

Performs an identity transformation on Erlang forms, as an example.

Parse Transformations

Parse transformations are used if a programmer wants to use Erlang syntax, but with different semantics. The original Erlang code is then transformed into other Erlang code.

Note:

Programmers are strongly advised not to engage in parse transformations and no support is offered for problems encountered.

See Also

\( \text{erl_parse(3), compile(3)}. \)
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) -> boolean()
Types:
   Name = atom()
   Arity = arity()
Returns true if Name/Arity is an Erlang BIF which is automatically recognized by the compiler, otherwise false.

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

type_test(Name, Arity) -> boolean()
Types:
   Name = atom()
   Arity = arity()
Returns true if Name/Arity is a valid Erlang type test, otherwise false.
arith_op(OpName, Arity) -> boolean()
Types:
   OpName = atom()
   Arity = arity()
Returns true if OpName/Arity is an arithmetic operator, otherwise false.
bool_op(OpName, Arity) -> boolean()
Types:
   OpName = atom()
   Arity = arity()
Returns true if OpName/Arity is a Boolean operator, otherwise false.
comp_op(OpName, Arity) -> boolean()
Types:
OpName = atom()
Arity = arity()

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

list_op(OpName, Arity) -> boolean()

Types:
   OpName = atom()
   Arity = arity()

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

send_op(OpName, Arity) -> boolean()

Types:
   OpName = atom()
   Arity = arity()

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

op_type(OpName, Arity) -> Type

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

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

Erlang module

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

The errors detected include:

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

Warnings include:

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

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

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

Data Types

\[
\text{error_info() = \{erl_scan:line(), module(), error_description()\}} \\
\text{error_description() = term()}
\]

Exports

\[
\text{module(AbsForms) -> \{ok, Warnings\} | \{error, Errors, Warnings\}} \\
\text{module(AbsForms, FileName) ->} \\
\quad \{ok, Warnings\} | \{error, Errors, Warnings\} \\
\text{module(AbsForms, FileName, CompileOptions) ->} \\
\quad \{ok, Warnings\} | \{error, Errors, Warnings\}
\]

Types:

\[
\text{AbsForms = [erl_parse:abstract_form()]} \\
\text{FileName = atom() | string()} \\
\text{CompileOptions = [compile:option()]} \\
\text{Warnings = [{file:filename()}, [ErrorInfo]]} \\
\text{Errors = [{FileName2 :: file:filename()}, [ErrorInfo]]} \\
\text{ErrorMessage = error_info()}
\]

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

\[
\{ok, Warnings\} \\
\quad \text{There were no errors in the module.}
\]
{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 \texttt{compile(3)}.

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

\[
\{[\{,[]}\}
\]

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) -> boolean()

Types:

\[
\begin{align*}
\text{Expr} &= \text{erl_parse:abstract_expr()} \\
\text{erl_parse:parse_exprs(Tokens)} &\text{ can be used to generate a list of Expr.}
\end{align*}
\]

format_error(ErrorDescriptor) -> \texttt{iо_lib:chars()}

Types:

\[
\begin{align*}
\text{ErrorDescriptor} &= \text{error_description()} \\
\text{Takes an ErrorDescriptor and returns a string which describes the error or warning. This function is usually called implicitly when processing an ErrorInfo structure (see below).}
\end{align*}
\]

Error Information

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

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

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

\[
\text{Module:format_error(ErrorDescriptor)}
\]

See Also

\texttt{erl_parse(3), epp(3)}
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(3)*).

**Data Types**

- **abstract_clause() = term()**
  Parse tree for Erlang clause.
- **abstract_expr() = term()**
  Parse tree for Erlang expression.
- **abstract_form() = term()**
  Parse tree for Erlang form.
- **error_description() = term()**
- **error_info() = {erl_scan:line(), module(), error_description()}**
- **token() = erl_scan:token()**

**Exports**

- **parse_form(Tokens) -> {ok, AbsForm} | {error, ErrorInfo}**
  Types:
  - Tokens = [token()]
  - AbsForm = abstract_form()
  - ErrorInfo = error_info()
  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, ExprList} | {error, ErrorInfo}**
  Types:
  - Tokens = [token()]
  - ExprList = [abstract_expr()]
  - ErrorInfo = error_info()
  This function parses Tokens as if it were a list of expressions. It returns:
  - {ok, ExprList} The parsing was successful. ExprList 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()]
   Term = term()
   ErrorInfo = error_info()
This function parses Tokens as if it were a term. It returns:
{ok, Term}
   The parsing was successful. Term is the Erlang term corresponding to the token list.
{error, ErrorInfo}
   An error occurred.

format_error(ErrorDescriptor) -> Chars
Types:
   ErrorDescriptor = error_description()
   Chars = [char() | Chars]
Uses an ErrorDescriptor and returns a string which describes the error. This function is usually called implicitly
when an ErrorInfo structure is processed (see below).

tokens(AbsTerm) -> Tokens
tokens(AbsTerm, MoreTokens) -> Tokens
Types:
   AbsTerm = abstract_expr()
   MoreTokens = Tokens = [token()]
This function generates a list of tokens representing the abstract form AbsTerm of an expression. Optionally, it
appends MoreTokens.

normalise(AbsTerm) -> Data
Types:
   AbsTerm = abstract_expr()
   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 = term()
   AbsTerm = abstract_expr()
Converts the Erlang data structure Data into an abstract form of type AbsTerm. This is the inverse of
normalise/1.
erl_parse:abstract(T) is equivalent to erl_parse:abstract(T, 0).

abstract(Data, Options) -> AbsTerm

Types:
  Data = term()
  Options = Line | [Option]
  Option = {line, Line} | {encoding, Encoding}
  Encoding = latin1 | unicode | utf8 | none | encoding_func()
  Line = erl_scan:line()
  AbsTerm = abstract_expr()
  encoding_func() = fun((integer() >= 0) -> boolean())

Converts the Erlang data structure Data into an abstract form of type AbsTerm.

The Line option is the line that will be assigned to each node of the abstract form.

The Encoding option is used for selecting which integer lists will be considered as strings. The default is to use the encoding returned by epp:default_encoding/0. The value none means that no integer lists will be considered as strings. The encoding_func() will be called with one integer of a list at a time, and if it returns true for every integer the list will be considered a string.

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:

\[ \text{Module:format_error(ErrorDescriptor)} \]

See Also

io(3), erl_scan(3), ERTS User’s Guide
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.

Data Types

hook_function() = none
| fun((Expr :: erl_parse:abstract_expr(),
      CurrentIndentation :: integer(),
      CurrentPrecedence :: integer() >= 0,
      Options :: options()) ->
      io_lib:chars())

The optional argument `HookFunction`, shown in the functions described below, defines a function which is called when an unknown form occurs where there should be a valid expression.

If `HookFunction` is equal to `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.

option() = {hook, hook_function()}
| {encoding, latin1 | unicode | utf8}

options() = hook_function() | [option()]

Exports

form(Form) -> io_lib:chars() 
form(Form, Options) -> io_lib:chars()

Types:
Form = erl_parse:abstract_form()
Options = options()

Pretty prints a `Form` which is an abstract form of a type which is returned by `erl_parse:parse_form/1`.

attribute(Attribute) -> io_lib:chars()
attribute(Attribute, Options) -> io_lib:chars()

Types:
Attribute = erl_parse:abstract_form()
Options = options()

The same as `form`, but only for the attribute `Attribute`.

function(Function) -> io_lib:chars()
function(Function, Options) -> io_lib:chars()

Types:
Function = `erl_parse:abstract_form()`  
Options = `options()`  
The same as `form`, but only for the function `Function`.

guard(Guard) -> `io_lib:chars()`  
guard(Guard, Options) -> `io_lib:chars()`  
Types:  
   Guard = `[erl_parse:abstract_expr()]`  
   Options = `options()`  
The same as `form`, but only for the guard test `Guard`.

exprs(Expressions) -> `io_lib:chars()`  
exprs(Expressions, Options) -> `io_lib:chars()`  
exprs(Expressions, Indent, Options) -> `io_lib:chars()`  
Types:  
   Expressions = `[erl_parse:abstract_expr()]`  
   Indent = integer()  
   Options = `options()`  
The same as `form`, but only for the sequence of expressions in `Expressions`.

expr(Expression) -> `io_lib:chars()`  
expr(Expression, Options) -> `io_lib:chars()`  
expr(Expression, Indent, Options) -> `io_lib:chars()`  
expr(Expression, Indent, Precedence, Options) -> `io_lib:chars()`  
Types:  
   Expression = `erl_parse:abstract_expr()`  
   Indent = integer()  
   Precedence = integer() >= 0  
   Options = `options()`  
This function prints one expression. It is useful for implementing hooks (see below).

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

See Also  
`io(3), erl_parse(3), erl_eval(3)`
This module contains functions for tokenizing characters into Erlang tokens.

### Data Types

```erl
attribute_info() = {column, column()} |
                  {length, integer() >= 1} |
                  {line, info_line()} |
                  {location, info_location()} |
                  {text, string()}
attributes() = line() | attributes_data()
attributes_data() = [{column, column()} |
                        {line, info_line()} |
                        {text, string()}] |
                  {line(), column()}
category() = atom()
column() = integer() >= 1
error_description() = term()
error_info() = {location(), module(), error_description()}
info_line() = integer() | term()
info_location() = location() | term()
line() = integer()
location() = line() | {line(), column()}
option() = return |
            return_white_spaces |
            return_comments |
            text |
            {reserved_word_fun, resword_fun()}
options() = option() |
            [option()]
symbol() = atom() | float() | integer() | string()
resword_fun() = fun((atom()) -> boolean())
token() = {category(), attributes(), symbol()} |
            {category(), attributes()}
token_info() = {category, category()} |
            {symbol, symbol()} |
            attribute_info()
tokens() = [token()]
tokens_result() = {ok, Tokens :: tokens(), EndLocation :: location()} |
                {eof, EndLocation :: location()} |
                {error, ErrorInfo :: error_info(),
```
Exports

string(String) -> Return
string(String, StartLocation) -> Return
string(String, StartLocation, Options) -> Return

Types:

  String = string()
  Options = options()
  Return = {ok, Tokens :: tokens(), EndLocation}
           | {error, ErrorInfo :: error_info(), ErrorLocation}
  StartLocation = EndLocation = ErrorLocation = location()

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, ErrorLocation} is returned if an error occurs. ErrorLocation 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 or attributes_info/1, 2.

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].
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 = return_cont() | []
CharSpec = char_spec()
StartLocation = location()
Options = options()
Return = {done,
Result :: tokens_result(),
LeftOverChars :: char_spec()}
| {more, Continuation1 :: return_cont()}
char_spec() = string() | eof

An opaque continuation

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. LeftOverChars is the remaining characters of the input data, starting from EndLocation.

{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 for a description of the various options.

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

token_info(Token) -> TokenInfo
Types:
Token = \textit{token()}
TokenInfo = [TokenInfoTuple :: \textit{token_info() }]

Returns a list containing information about the token \textit{Token}. The order of the TokenInfoTuples is not defined. See \textit{token_info/2 } for information about specific TokenInfoTuples.

Note that if \textit{token_info (Token, TokenItem) } returns undefined for some \textit{TokenItem}, the item is not included in TokenInfo.

token_info (Token, TokenItem) -> TokenInfoTuple | undefined
token_info (Token, TokenItems) -> TokenInfo

Types:
\begin{itemize}
\item \textit{Token} = \textit{token()}
\item \textit{TokenItems} = [\textit{TokenItem} :: \textit{token_item() }]
\item \textit{TokenInfo} = [\textit{TokenInfoTuple} :: \textit{token_info() }]
\item \textit{token_item()} = \textit{category} | \textit{symbol} | \textit{attribute_item()}
\item \textit{attribute_item()} = \textit{column} | \textit{length} | \textit{line} | \textit{location} | \textit{text}
\end{itemize}

Returns a list containing information about the token \textit{Token}. If one single \textit{TokenItem} is given the returned value is the corresponding TokenInfoTuple, or undefined if the \textit{TokenItem} has no value. If a list of \textit{TokenItems} is given the result is a list of TokenInfoTuple. The TokenInfoTuples will appear with the corresponding \textit{TokenItems} in the same order as the \textit{TokenItems} appear in the list of \textit{TokenItems}. \textit{TokenItems} with no value are not included in the list of TokenInfoTuple.

The following TokenInfoTuples with corresponding \textit{TokenItems} are valid:
\begin{itemize}
\item \{\textit{category}, \textit{category() }\} The category of the token.
\item \{\textit{column}, \textit{column() }\} The column where the token begins.
\item \{\textit{length}, \textit{integer()} > 0\} The length of the token's text.
\item \{\textit{line}, \textit{line() }\} The line where the token begins.
\item \{\textit{location}, \textit{location() }\} The line and column where the token begins, or just the line if the column unknown.
\item \{\textit{symbol}, \textit{symbol() }\} The token's symbol.
\item \{\textit{text}, \textit{string() }\} The token's text.
\end{itemize}

\textit{attributes_info (Attributes) -> AttributesInfo}

Types:
Attributes = \textit{attributes()}

AttributesInfo = [AttributeInfoTuple :: \textit{attribute_info()})

Returns a list containing information about the token attributes \textit{Attributes}. The order of the \textit{AttributeInfoTuples} is not defined. See \textit{attributes_info/2} for information about specific \textit{AttributeInfoTuples}.

Note that if \textit{attributes_info(Token, AttributeItem)} returns \textit{undefined} for some \textit{AttributeItem} in the list above, the item is not included in \textit{AttributesInfo}.

\texttt{attributes_info(Attributes, AttributeItem)} -> AttributeInfoTuple | undefined
\texttt{attributes_info(Attributes, AttributeItems)} -> AttributeInfo

\textbf{Types:}

\texttt{Attributes} = \textit{attributes()}
\texttt{AttributeItems} = [AttributeItem :: \textit{attribute_item()})
\texttt{AttributeInfo} = [AttributeInfoTuple :: \textit{attribute_info()})
\texttt{attribute_item()} = \texttt{column} | \texttt{length} | \texttt{line} | \texttt{location} | \texttt{text}

Returns a list containing information about the token attributes \textit{Attributes}. If one single \textit{AttributeItem} is given the returned value is the corresponding \textit{AttributeInfoTuple}, or \textit{undefined} if the \textit{AttributeItem} has no value. If a list of \textit{AttributeItem} is given the result is a list of \textit{AttributeInfoTuple}. The \textit{AttributeInfoTuples} will appear with the corresponding \textit{AttributeItems} in the same order as the \textit{AttributeItems appear in the list of AttributeItems}. \textit{AttributeItems} with no value are not included in the list of \textit{AttributeInfoTuple}.

The following \textit{AttributeInfoTuples} with corresponding \textit{AttributeItems} are valid:

\{\texttt{column}, \texttt{column()})

The column where the token begins.

\{\texttt{length}, \texttt{integer()} > 0\}

The length of the token's text.

\{\texttt{line}, \texttt{line()})

The line where the token begins.

\{\texttt{location}, \texttt{location()})

The line and column where the token begins, or just the line if the column unknown.

\{\texttt{text}, \texttt{string()})

The token's text.

\texttt{set_attribute(AttributeItem, Attributes, SetAttributeFun)} -> Attributes

\textbf{Types:}

\texttt{AttributeItem} = \texttt{line}
\texttt{Attributes} = \textit{attributes()}
\texttt{SetAttributeFun} = \texttt{fun((info_line()) -> info_line())}

Sets the value of the \texttt{line} attribute of the token attributes \textit{Attributes}. 

126 | Ericsson AB. All Rights Reserved.: STDLIB
The `SetAttributeFun` is called with the value of the `line` attribute, and is to return the new value of the `line` attribute.

```erlang
format_error(ErrorCode) -> string()
```

**Types:**

```erlang
 ErrorCode = error_description()
```

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

### Error Information

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

```erlang
{ErrorLocation, Module, ErrorCode}
```

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

```erlang
Module:format_error(ErrorCode)
```

### Notes

The continuation of the first call to the re-entrant input functions must be [ ]. Refer to Armstrong, Virding and Williams, ‘Concurrent Programming in Erlang’, Chapter 13, for a complete description of how the re-entrant input scheme works.

### See Also

`io(3), erl_parse(3)`
**erl_tar**

Erlang module

The erl_tar module archives and extract files to and from a tar file. erl_tar supports the ustar format (IEEE Std 1003.1 and ISO/IEC 9945-1). All modern tar programs (including GNU tar) can read this format. To ensure that GNU tar produces a tar file that erl_tar can read, give the --format=ustar option to GNU tar.

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 or create/3 function. Alternatively, for more control, the open, add/3,4, and close/1 functions can be used.

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

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

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

**UNICODE SUPPORT**

If file:native_name_encoding/0 returns utf8, path names will be encoded in UTF-8 when creating tar files and path names will be assumed to be encoded in UTF-8 when extracting tar files.

If file:native_name_encoding/0 returns latin1, no translation of path names will be done.

**OTHER STORAGE MEDIA**

The erl_ftp module normally accesses the tar-file on disk using the file module. When other needs arise, there is a way to define your own low-level Erlang functions to perform the writing and reading on the storage media. See init/3 for usage.

An example of this is the sftp support in ssh_sftp:open_tar/3. That function opens a tar file on a remote machine using an sftp channel.

**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 | {chunks, ChunkSize}
ChunkSize = positive_integer()
RetVal = 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.

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.

{chunks, ChunkSize}
  Read data in parts from the file. This is intended for memory-limited machines that for example builds a tar file on a remote machine over sftp.

add(TarDescriptor, FilenameOrBin, NameInArchive, Options) -> RetVal
Types:
  TarDescriptor = term()
  FilenameOrBin = filename() | binary()
  Filename = filename()
  NameInArchive = filename()
  Options = [Option]
  Option = dereference | verbose
  RetVal = 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. It accepts the same options as add/3.

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.

create(Name, FileList) -> RetVal
Types:
  Name = filename()
  FileList = [Filename | {NameInArchive, binary()}, {NameInArchive, Filename}]
  Filename = filename()
  NameInArchive = filename()
RetValue = ok | {error, {Name, Reason}}
Reason = term()

The create/2 function creates a tar file and archives the files whose names are given in FileList into it. The files may either be read from disk or given as binaries.

create(Name, FileList, OptionList)
Types:
   Name = filename()
   FileList = [Filename | {NameInArchive, binary()}, {NameInArchive, Filename}]
   Filename = filename()
   NameInArchive = filename()
   OptionList = [Option]
   Option = compressed | cooked | dereference | verbose
   RetValue = ok | {error, {Name, Reason}}
   Reason = term()

The create/3 function creates a tar file and archives the files whose names are given in FileList into it. The files may either be read from disk or given as binaries.

The options in OptionList modify the defaults as follows.

compressed
   The entire tar file will be compressed, as if it has been run through the gzip program. To abide to the convention that a compressed tar file should end in ".tar.gz" or "tgz", you’ll need to add the appropriate extension yourself.

cooked
   By default, the open/2 function will open the tar file in raw mode, which is faster but does not allow a remote (erlang) file server to be used. Adding cooked to the mode list will override the default and open the tar file without the raw option.

dereference
   By default, symbolic links will be stored as symbolic links in the tar file. Use the dereference option to override the default and store the file that the symbolic link points to into the tar file.

verbose
   Print an informational message about each file being added.

extract(Name) -> RetValue
Types:
   Name = filename()
   RetValue = ok | {error, {Name, Reason}}
   Reason = term()

The extract/1 function extracts all files from a tar archive.

If the Name argument is given as "(binary, Binary)", the contents of the binary is assumed to be a tar archive.
If the Name argument is given as "(file, Fd)", Fd is assumed to be a file descriptor returned from the file:open/2 function.
Otherwise, Name should be a filename.
extract(Name, OptionList)

Types:
Name = filename() | {binary,Binary} | {file,Fd}
Binary = binary()
Fd = file_descriptor()
OptionList = [Option]
Option = {cwd,Cwd}|{files,FileList}|keep_old_files|verbose|memory
Cwd = [dirname()]
FileList = [filename()]
RetVal = ok|MemoryRetVal|{error,(Name,Reason)}
MemoryRetVal = {ok, [{NameInArchive,binary()}]}
NameInArchive = filename()
Reason = term()

The extract/2 function extracts files from a tar archive.

If the Name argument is given as "{binary,Binary}"", the contents of the binary is assumed to be a tar archive.

If the Name argument is given as "{file,Fd}"", Fd is assumed to be a file descriptor returned from the file:open/2 function.

Otherwise, Name should be a filename.

The following options modify the defaults for the extraction as follows.

{cwd,Cwd}
Files with relative filenames will by default be extracted to the current working directory. Given the {cwd,Cwd} option, the extract/2 function will extract into the directory Cwd instead of to the current working directory.

{files,FileList}
By default, all files will be extracted from the tar file. Given the {files,Files} option, the extract/2 function will only extract the files whose names are included in FileList.

compressed
Given the compressed option, the extract/2 function will uncompress the file while extracting. If the tar file is not actually compressed, the compressed will effectively be ignored.

cooked
By default, the open/2 function will open the tar file in raw mode, which is faster but does not allow a remote (erlang) file server to be used. Adding cooked to the mode list will override the default and open the tar file without the raw option.

memory
Instead of extracting to a directory, the memory option will give the result as a list of tuples {Filename, Binary}, where Binary is a binary containing the extracted data of the file named Filename in the tar file.

keep_old_files
By default, all existing files with the same name as file in the tar file will be overwritten. Given the keep_old_files option, the extract/2 function will not overwrite any existing files.

verbose
Print an informational message as each file is being extracted.
format_error(Reason) -> string()
Types:
  Reason = term()

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

open(Name, OpenModeList) -> RetValue
Types:
  Name = filename()
  OpenModeList = [OpenMode]
  Mode = write|compressed|cooked
  RetValue = {ok,TarDescriptor}|{error,{Name,Reason}}
  TarDescriptor = term()
  Reason = term()

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 functions to add one file at the time into an opened tar file. When you are finished adding files, use the close function to close the tar file.

Warning:
The TarDescriptor term is not a file descriptor. You should not rely on the specific contents of the TarDescriptor term, as it may change in future versions as more features are added to the erl_tar module.

init(UserPrivate, AccessMode, Fun) -> {ok,TarDescriptor} | {error,Reason}
Types:
  UserPrivate = term()
  AccessMode = [write] | [read]
  Fun when AccessMode is [write] = fun(write, {UserPrivate,DataToWrite})->...; (position,{UserPrivate,Position})->...; (close, UserPrivate)->...
  Fun when AccessMode is [read] = fun(read2, {UserPrivate,Size})->...; (position,{UserPrivate,Position})->...; (close, UserPrivate)->... end
  TarDescriptor = term()
**Reason = term()**

The Fun is the definition of what to do when the different storage operations functions are to be called from the higher tar handling functions (add/3, add/4, close/1...).

The Fun will be called when the tar function wants to do a low-level operation, like writing a block to a file. The Fun is called as Fun(Op, (UserPrivate, Parameters...)) where Op is the operation name, UserPrivate is the term passed as the first argument to init/1 and Parameters... are the data added by the tar function to be passed down to the storage handling function.

The parameter UserPrivate is typically the result of opening a low level structure like a file descriptor, a sftp channel id or such. The different Fun clauses operates on that very term.

The fun clauses parameter lists are:

- (write, {UserPrivate,DataToWrite})
  - Write the term DataToWrite using UserPrivate
- (close, UserPrivate)
  - Close the access.
- (read2, {UserPrivate,Size})
  - Read using UserPrivate but only Size bytes. Note that there is only an arity-2 read function, not an arity-1
- (position, {UserPrivate,Position})
  - Sets the position of UserPrivate as defined for files in file:position/2

A complete Fun parameter for reading and writing on files using the file module could be:

```erlang
ExampleFun =
  fun(write, {Fd,Data}) ->  file:write(Fd, Data);
  (position, {Fd,Pos}) -> file:position(Fd, Pos);
  (read2, {Fd,Size}) -> file:read(Fd,Size);
  (close, Fd) -> file:close(Fd)
end
```

where Fd was given to the init/3 function as:

```erlang
{ok,Fd} = file:open(Name,...).
{ok,TarDesc} = erl_tar:init(Fd, [write], ExampleFun),
```

The TarDesc is then used:

```erlang
erl_tar:add(TarDesc, SomeValueIwantToAdd, FileNameInTarFile),
.....,
erl_tar:close(TarDesc)
```

When the erl_tar core wants to e.g. write a piece of Data, it would call ExampleFun(write, (UserPrivate,Data)).
Note:
The example above with file module operations is not necessary to use directly since that is what the open function in principle does.

Warning:
The TarDescriptor term is not a file descriptor. You should not rely on the specific contents of the TarDescriptor term, as it may change in future versions as more features are added to the erl_tar module.

table(Name) -> RetValue
Types:
   Name = filename()
   RetValue = {ok,[string()]}|{error,(Name,Reason)}
   Reason = term()

The table/1 function retrieves the names of all files in the tar file Name.

table(Name, Options)
Types:
   Name = filename()

The table/2 function retrieves the names of all files in the tar file Name.

t(Name)
Types:
   Name = filename()

The t/1 function prints the names of all files in the tar file Name to the Erlang shell. (Similar to "tar t").

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 "tar tv").
This module is an interface to the Erlang built-in term storage BIFs. These provide the ability to store very large quantities of data in an Erlang runtime system, and to have constant access time to the data. (In the case of ordered_set, see below, access time is proportional to the logarithm of the number of objects stored).

Data is organized as a set of dynamic tables, which can store tuples. Each table is created by a process. When the process terminates, the table is automatically destroyed. Every table has access rights set at creation.

Tables are divided into four different types, set, ordered_set, bag and duplicate_bag. A set or ordered_set table can only have one object associated with each key. A bag or duplicate_bag can have many objects associated with each key.

The number of tables stored at one Erlang node is limited. The current default limit is approximately 1400 tables. The upper limit can be increased by setting the environment variable ERL_MAX_ETS_TABLES before starting the Erlang runtime system (i.e. with the -env option to erl/werl). The actual limit may be slightly higher than the one specified, but never lower.

Note that there is no automatic garbage collection for tables. Even if there are no references to a table from any process, it will not automatically be destroyed unless the owner process terminates. It can be destroyed explicitly by using delete/1. The default owner is the process that created the table. Table ownership can be transferred at process termination by using the heir option or explicitly by calling give_away/3.

Some implementation details:

- In the current implementation, every object insert and look-up operation results in a copy of the object.
- '$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.

Failure

In general, the functions below will exit with reason badarg if any argument is of the wrong format, if the table identifier is invalid or if the operation is denied due to table access rights (protected or private).

Concurrency

This module provides some limited support for concurrent access. All updates to single objects are guaranteed to be both atomic and isolated. This means that an updating operation towards a single object will either succeed or fail completely without any effect at all (atomicity). Nor can any intermediate results of the update be seen by other processes (isolation). Some functions that update several objects state that they even guarantee atomicity and isolation for the entire operation. In database terms the isolation level can be seen as "serializable", as if all isolated operations were carried out serially, one after the other in a strict order.

No other support is available within ETS that would guarantee consistency between objects. However, 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 existing 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 objects that are inserted or deleted during such a traversal may be visited once or not at all. Functions that internally traverse over a table, like \texttt{select} and \texttt{match}, will give the same guarantee as \texttt{safe_fixtable}.

\section*{Match Specifications}

Some of the functions uses a \textit{match specification}, \texttt{match_spec}. A brief explanation is given in \texttt{select/2}. For a detailed description, see the chapter "Match specifications in Erlang" in \textit{ERTS User's Guide}.

\section*{Data Types}

\begin{itemize}
  \item \texttt{access()} = public | protected | private
  \item \texttt{continuation()}
  \begin{itemize}
    \item Opaque continuation used by \texttt{select/1,3, select_reverse/1,3, match/1,3,} and \texttt{match_object/1,3}.
  \end{itemize}
  \item \texttt{match_spec()} = [{\texttt{match_pattern()}, [term()], [term()]}]
  \begin{itemize}
    \item A match specification, see above.
  \end{itemize}
  \item \texttt{comp_match_spec()}
  \begin{itemize}
    \item A compiled match specification.
  \end{itemize}
  \item \texttt{match_pattern()} = atom() | tuple()
  \item \texttt{tab()} = atom() | \texttt{tid()}
  \item \texttt{tid()}
  \begin{itemize}
    \item A table identifier, as returned by \texttt{new/2}.
  \end{itemize}
  \item \texttt{type()} = set | ordered_set | bag | duplicate_bag
\end{itemize}

\section*{Exports}

\begin{itemize}
  \item \texttt{all()} -> [\texttt{Tab}]
  \begin{itemize}
    \item Types:
    \begin{itemize}
      \item \texttt{Tab} = \texttt{tab()}
    \end{itemize}
    \begin{itemize}
      \item Returns a list of all tables at the node. Named tables are given by their names, unnamed tables are given by their table identifiers.
    \end{itemize}
    \begin{itemize}
      \item There is no guarantee of consistency in the returned list. Tables created or deleted by other processes "during" the \texttt{ets:all()} call may or may not be included in the list. Only tables created/deleted \textit{before} \texttt{ets:all()} is called are guaranteed to be included/excluded.
    \end{itemize}
  \end{itemize}
  \item \texttt{delete(Tab)} -> true
  \begin{itemize}
    \item Types:
    \begin{itemize}
      \item \texttt{Tab} = \texttt{tab()}
    \end{itemize}
    \begin{itemize}
      \item Deletes the entire table \texttt{Tab}.
    \end{itemize}
  \end{itemize}
  \item \texttt{delete(Tab, Key)} -> true
  \begin{itemize}
    \item Types:
    \begin{itemize}
      \item \texttt{Tab} = \texttt{tab()}
      \item \texttt{Key} = \texttt{term()}
    \end{itemize}
    \begin{itemize}
      \item Deletes a specific object \texttt{Key} from table \texttt{Tab}.
    \end{itemize}
  \end{itemize}
\end{itemize}
Tab = \texttt{tab()}
Key = \texttt{term()}

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

\texttt{delete_all_objects(Tab) \rightarrow true}

Types:
\texttt{Tab = tab()}

Delete all objects in the ETS table \texttt{Tab}. The operation is guaranteed to be \textit{atomic and isolated}.

\texttt{delete_object(Tab, Object) \rightarrow true}

Types:
\texttt{Tab = tab()}
\texttt{Object = tuple()}

Delete the exact object \texttt{Object} from the ETS table, leaving objects with the same key but other differences (useful for type \texttt{bag}). In a duplicate\_bag, all instances of the object will be deleted.

\texttt{file2tab(Filename) \rightarrow \{ok, Tab\} | \{error, Reason\}}

Types:
\texttt{Filename = file:name()}
\texttt{Tab = tab()}
\texttt{Reason = term()}

Reads a file produced by \texttt{tab2file/2} or \texttt{tab2file/3} and creates the corresponding table \texttt{Tab}.

Equivalent to \texttt{file2tab(Filename, [])}.

\texttt{file2tab(Filename, Options) \rightarrow \{ok, Tab\} | \{error, Reason\}}

Types:
\texttt{Filename = file:name()}
\texttt{Tab = tab()}
\texttt{Options = [Option]}
\texttt{Option = \{verify, boolean()\}}
\texttt{Reason = term()}

Reads a file produced by \texttt{tab2file/2} or \texttt{tab2file/3} and creates the corresponding table \texttt{Tab}.

The currently only supported option is \texttt{\{verify, boolean()\}}. If verification is turned on (by means of specifying \texttt{\{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 \texttt{tab2file/3}.

If no extended\_info is present in the file and \texttt{\{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 \texttt{tab2file/3}, 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.

\texttt{\{verify, false\}} is the default.
first(Tab) -> Key | 'end_of_table'
Types:
  Tab = tab()
  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((Element :: term(), AccIn) -> AccOut)
  Tab = tab()
  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((Element :: term(), AccIn) -> AccOut)
  Tab = tab()
  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 = tab()
  DetsTab = dets:tab_name()
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 = function()
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. 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_ guard tests. Language constructs that have no representation in a match_spec (like if, case, receive etc) are not allowed.

The return value is the resulting match_spec.

Example:

1> ets:fun2ms(fun({M,N}) when N > 3 -> M end).
[[{{'$1','$2'},[{'>','$2',3}],['$1']}}]

Variables from the environment can be imported, so that this works:

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

The imported variables will be replaced by match_spec const expressions, which is consistent with the static scoping for Erlang funs. Local or global function calls can not be in the guard or body of the fun however. Calls to builtin match_spec functions of course is allowed:

4> ets:fun2ms(fun({M,N}) when N > X, is_atom(M) -> M end).
Error: fun containing local Erlang function calls
        (is_atom called in guard) cannot be translated into match_spec
        {error,transform_error}
5> ets:fun2ms(fun({M,N}) when N > X, is_atom(M) -> M end).
[[{{'$1','$2'},{'>','$2',{const,3}},{is_atom,'$1'}},{'$1'}]]

As can be seen by the example, the function can be called from the shell too. The fun needs to be literally in the call when used from the shell as well. Other means than the parse_transform are used in the shell case, but more or less the same restrictions apply (the exception being records, as they are not handled by the shell).
Warning:

If the parse_transform is not applied to a module which calls this pseudo function, the call will fail in runtime (with a badarg). The module ets actually exports a function with this name, but it should never really be called except for when using the function in the shell. If the parse_transform is properly applied by including the ms_transform.hrl header file, compiled code will never call the function, but the function call is replaced by a literal match_spec.

For more information, see ms_transform(3).

give_away(Tab, Pid, GiftData) -> true
Types:
   Tab = tab()
   Pid = pid()
   GiftData = term()

Make process Pid the new owner of table Tab. If successful, the message {'ETS-TRANSFER',Tab,FromPid,GiftData} will be sent to the new owner.

The process Pid must be alive, local and not already the owner of the table. The calling process must be the table owner.

Note that give_away does not at all affect the heir option of the table. A table owner can for example set the heir to itself, give the table away and then get it back in case the receiver terminates.

i() -> ok
Displays information about all ETS tables on tty.

i(Tab) -> ok
Types:
   Tab = tab()
Browses the table Tab on tty.

info(Tab) -> InfoList | undefined
Types:
   Tab = tab()
   InfoList = [InfoTuple]
   InfoTuple = {compressed, boolean()}
      | {heir, pid() | none}
      | {keypos, integer() => 1}
      | {memory, integer() => 0}
      | {name, atom()}
      | {named_table, boolean()}
      | {node, node()}
      | {owner, pid()}
      | {protection, access()}
      | {size, integer() => 0}
info(Tab, Item) -> Value | undefined
Types:
  Tab = tab()
  Item = compressed
       | fixed
       | heir
       | keypos
       | memory
       | name
       | named_table
       | node
       | owner
       | protection
       | safe_fixed
       | size
       | stats

Returns information about the table Tab as a list of 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.

- {compressed, boolean()}
  Indicates if the table is compressed or not.
- {heir, pid() | none}
  The pid of the heir of the table, or none if no heir is set.
- {keypos, integer() >= 1}
  The key position.
- {memory, integer() >= 0}
  The number of words allocated to the table.
- {name, atom()}
  The name of the table.
- {named_table, boolean()}
  Indicates if the table is named or not.
- {node, node()}
  The node where the table is stored. This field is no longer meaningful as tables cannot be accessed from other nodes.
- {owner, pid()}
  The pid of the owner of the table.
- {protection, access()}
  The table access rights.
- {size, integer() >= 0}
  The number of objects inserted in the table.
- {type, type()}
  The table type.
type
Value = term()

Returns the information associated with Item for the table Tab, or returns undefined if Tab does not refer an existing ETS table. If Tab is not of the correct type, or if Item is not one of the allowed values, this function fails with reason badarg.

Warning:
In R11B and earlier, this function would not fail but return undefined for invalid values for Item.

In addition to the (Item,Value) pairs defined for info/1, the following items are allowed:

- **Item=fixed, Value=boolean()**
  Indicates if the table is fixed by any process or not.

- **Item=safe_fixed, Value={FirstFixed,Info}|false**
  If the table has been fixed using safe_fixtable/2, the call returns a tuple where FirstFixed is the time when the table was first fixed by a process, which may or may not be one of the processes it is fixed by right now. Info is a possibly empty lists of tuples {Pid,RefCount}, one tuple for every process the table is fixed by right now. RefCount is the value of the reference counter, keeping track of how many times the table has been fixed by the process.
  If the table never has been fixed, the call returns false.

- **Item=stats, Value=tuple()**
  Returns internal statistics about set, bag and duplicate_bag tables on an internal format used by OTP test suites. Not for production use.

```erlang
init_table(Tab, InitFun) -> true

Types:
  Tab = tab()
  InitFun = fun((Arg) -> Res)
  Arg = read | close
  Res = end_of_input | {Objects :: [term()], InitFun} | term()
```

Replaces the existing objects of the table Tab with objects created by calling the input function InitFun, see below. This function is provided for compatibility with the dets module, it is not more efficient than filling a table by using ets:insert/2.

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

If the type of the table is set and there is more than one object with a given key, one of the objects is chosen. This is not necessarily the last object with the given key in the sequence of objects returned by the input functions. This holds also for duplicated objects stored in tables of type bag.

```erlang
insert(Tab, ObjectOrObjects) -> true

Types:
```
Tab = `tab()`
ObjectOrObjects = `tuple()` | `[tuple()]`

Inserts the object or all of the objects in the list ObjectOrObjects into the table Tab. If the table is a `set` and the key of the inserted objects matches the key of any object in the table, the old object will be replaced. If the table is an `ordered_set` and the key of the inserted object compares equal to the key of any object in the table, the old object is also replaced. If the list contains more than one object with matching keys and the table is a `set`, one will be inserted, which one is not defined. The same thing holds for `ordered_set`, but will also happen if the keys compare equal.

The entire operation is guaranteed to be **atomic and isolated**, even when a list of objects is inserted.

`insert_new(Tab, ObjectOrObjects) -> boolean()`  
**Types:**
- Tab = `tab()`  
- ObjectOrObjects = `tuple()` | `[tuple()]`

This function works exactly like `insert/2`, with the exception that instead of overwriting objects with the same key (in the case of `set` or `ordered_set`) or adding more objects with keys already existing in the table (in the case of `bag` and `duplicate_bag`), it simply returns `false`. If `ObjectOrObjects` is a list, the function checks every key prior to inserting anything. Nothing will be inserted if not all keys present in the list are absent from the table. Like `insert/2`, the entire operation is guaranteed to be **atomic and isolated**.

`is_compiled_ms(Term) -> boolean()`  
**Types:**
- Term = `term()`

This function is used to check if a term is a valid compiled `match_spec`. The compiled match_spec is an opaque datatype which can not be sent between Erlang nodes nor be stored on disk. Any attempt to create an external representation of a compiled match_spec will result in an empty binary (`<<>>`). As an example, the following expression:

```erlang
ets:is_compiled_ms(ets:match_spec_compile([{'_',[],[true]}])).
```

will yield `true`, while the following expressions:

```erlang
MS = ets:match_spec_compile([{'_',[],[true]}]),
Broken = binary_to_term(term_to_binary(MS)),
ets:is_compiled_ms(Broken).
```

will yield `false`, as the variable `Broken` will contain a compiled match_spec that has passed through external representation.

**Note:**

The fact that compiled match_specs has no external representation is for performance reasons. It may be subject to change in future releases, while this interface will still remain for backward compatibility reasons.

`last(Tab) -> Key | '$end_of_table'`  
**Types:**
Tab = \texttt{tab()}
Key = \texttt{term()}

Returns the last key Key according to Erlang term order in the table Tab of the \texttt{ordered_set} type. If the table is of any other type, the function is synonymous to first/2. If the table is empty, '$\texttt{end_of_table}$' is returned. Use \texttt{prev/2} to find preceding keys in the table.

\underline{lookup(Tab, Key) -> [Object]}

Types:
\begin{itemize}
  \item Tab = \texttt{tab()}
  \item Key = \texttt{term()}
  \item Object = \texttt{tuple()}
\end{itemize}

Returns a list of all objects with the key Key in the table Tab.

In the case of \texttt{set}, \texttt{bag} and \texttt{duplicate_bag}, an object is returned only if the given key \texttt{matches} the key of the object in the table. If the table is an \texttt{ordered_set} however, an object is returned if the key given \texttt{compares equal} to the key of an object in the table. The difference being the same as between \texttt{==} and \texttt{==}. As an example, one might insert an object with the \texttt{integer()} 1 as a key in an \texttt{ordered_set} and get the object returned as a result of doing a \texttt{lookup/2} with the \texttt{float()} 1.0 as the key to search for.

If the table is of type \texttt{set} or \texttt{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 \texttt{bag} or \texttt{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 \texttt{set}, \texttt{bag} and \texttt{duplicate_bag} are constant, regardless of the size of the table. For the \texttt{ordered_set} data-type, time is proportional to the (binary) logarithm of the number of objects.

\underline{lookup_element(Tab, Key, Pos) -> Elem}

Types:
\begin{itemize}
  \item Tab = \texttt{tab()}
  \item Key = \texttt{term()}
  \item Pos = \texttt{integer() >= 1}
  \item Elem = \texttt{term()} | \[\texttt{term()}\]
\end{itemize}

If the table Tab is of type \texttt{set} or \texttt{ordered_set}, the function returns the Pos:th element of the object with the key Key.

If the table is of type \texttt{bag} or \texttt{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 \texttt{set}, \texttt{bag} and \texttt{duplicate_bag} on one hand, and \texttt{ordered_set} on the other, regarding the fact that \texttt{ordered_set}’s view keys as equal when they \texttt{compare equal} whereas the other table types only regard them equal when they match, naturally holds for \texttt{lookup_element} as well as for \texttt{lookup}.

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

Types:
\begin{itemize}
\end{itemize}
Tab = tab()
Pattern = match_pattern()
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:

```erlang
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 = tab()
- Pattern = match_pattern()
- Limit = integer() >= 1
- Match = [term()]
- Continuation = continuation()

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 = continuation()

Continues a match started with ets:match/3. The next chunk of the size given in the initial ets:match/3 call is returned together with a new Continuation that can be used in subsequent calls to this function.

'$end_of_table' is returned when there are no more objects in the table.
match_delete(Tab, Pattern) -> true  
Types:  
  Tab = tab()  
  Pattern = match_pattern()  
Deletes all objects which match the pattern Pattern from the table Tab. See match/2 for a description of patterns.

match_object(Tab, Pattern) -> [Object]  
Types:  
  Tab = tab()  
  Pattern = match_pattern()  
  Object = tuple()  
Matches the objects in the table Tab against the pattern Pattern. See match/2 for a description of patterns. The function returns a list of all objects which match the pattern.  
If the key is specified in the pattern, the match is very efficient. If the key is not specified, i.e. if it is a variable or an underscore, the entire table must be searched. The search time can be substantial if the table is very large.  
On tables of the ordered_set type, the result is in the same order as in a first/next traversal.

match_object(Tab, Pattern, Limit) ->  
  {{Match}, Continuation} | '$end_of_table'  
Types:  
  Tab = tab()  
  Pattern = match_pattern()  
  Limit = integer() >= 1  
  Match = [term()]  
  Continuation = continuation()  
Works like ets:match_object/2 but only returns a limited (Limit) number of matching objects. The Continuation term can then be used in subsequent calls to ets:match_object/1 to get the next chunk of matching objects. This is a space efficient way to work on objects in a table which is still faster than traversing the table object by object using ets:first/1 and ets:next/1.  
'$end_of_table' is returned if the table is empty.

match_object(Continuation) ->  
  {{Match}, Continuation} | '$end_of_table'  
Types:  
  Match = [term()]  
  Continuation = continuation()  
Continues a match started with ets:match_object/3. The next chunk of the size given in the initial ets:match_object/3 call is returned together with a new Continuation that can be used in subsequent calls to this function.  
'$end_of_table' is returned when there are no more objects in the table.

match_spec_compile(MatchSpec) -> CompiledMatchSpec  
Types:
MatchSpec = match_spec()
CompiledMatchSpec = comp_match_spec()

This function transforms a \texttt{match\_spec} into an internal representation that can be used in subsequent calls to \texttt{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 \texttt{match\_spec}, nor can it be stored on disk). The validity of a compiled \texttt{match\_spec} can be checked using \texttt{ets:is\_compiled\_ms/1}.

If the term \texttt{MatchSpec} can not be compiled (does not represent a valid \texttt{match\_spec}), a \texttt{badarg} fault is thrown.

\textbf{Note:}

This function has limited use in normal code, it is used by Dets to perform the \texttt{dets:select} operations.

\texttt{match\_spec\_run(List, CompiledMatchSpec) -> list()}
\textbf{Types:}

\begin{itemize}
  \item \texttt{List} = \{tuple()\}
  \item CompiledMatchSpec = \texttt{comp\_match\_spec()}
\end{itemize}

This function executes the matching specified in a compiled \texttt{match\_spec} on a list of tuples. The CompiledMatchSpec term should be the result of a call to \texttt{ets:match\_spec\_compile/1} and is hence the internal representation of the \texttt{match\_spec} one wants to use.

The matching will be executed on each element in \texttt{List} and the function returns a list containing all results. If an element in \texttt{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 \texttt{List}. The two calls in the following example will give the same result (but certainly not the same execution time...):

\begin{verbatim}
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),
\end{verbatim}

\textbf{Note:}

This function has limited use in normal code, it is used by Dets to perform the \texttt{dets:select} operations and by Mnesia during transactions.

\texttt{member(Tab, Key) -> boolean()}
\textbf{Types:}

\begin{itemize}
  \item \texttt{Tab} = \texttt{tab()}
  \item \texttt{Key} = \texttt{term()}
\end{itemize}

Works like \texttt{lookup/2}, but does not return the objects. The function returns \texttt{true} if one or more elements in the table has the key \texttt{Key}, \texttt{false} otherwise.
new(Name, Options) -> tid() | atom()

Types:

Name = atom()
Options = [Option]
Option = Type
| Access
| named_table
| {keypos, Pos}
| {heir, Pid :: pid(), HeirData}
| {heir, none}
| Tweaks

Type = type()
Access = access()
Tweaks = {write_concurrency, boolean()}
| {read_concurrency, boolean()}
| compressed
Pos = integer() >= 1
HeirData = term()

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}, {heir,none}, {write_concurrency,false}, {read_concurrency,false}].

- **set** The table is a set table - one key, one object, no order among objects. This is the default table type.
- **ordered_set** The table is a ordered_set table - one key, one object, ordered in Erlang term order, which is the order implied by the < and > operators. Tables of this type have a somewhat different behavior in some situations than tables of the other types. Most notably the ordered_set tables regard keys as equal when they compare equal, not only when they match. This means that to an ordered_set, the integer() 1 and the float() 1.0 are regarded as equal. This also means that the key used to lookup an element not necessarily matches the key in the elements returned, if float()'s and integer()'s are mixed in keys of a table.
- **bag** The table is a bag table which can have many objects, but only one instance of each object, per key.
- **duplicate_bag** The table is a duplicate_bag table which can have many objects, including multiple copies of the same object, per key.
- **public** Any process may read or write to the table.
- **protected** The owner process can read and write to the table. Other processes can only read the table. This is the default setting for the access rights.
- **private** Only the owner process can read or write to the table.
- **named_table** If this option is present, the name Name is associated with the table identifier. The name can then be used instead of the table identifier in subsequent operations.
- **{keypos,Pos}** 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.

- **{heir,Pid,HeirData}** | **{heir,none}**
Set a process as heir. The heir will inherit the table if the owner terminates. The message `{'ETS-TRANSFER',tid(),FromPid,HeirData}` will be sent to the heir when that happens. The heir must be a local process. Default heir is `none`, which will destroy the table when the owner terminates.

- `{write_concurrency,boolean()}` Performance tuning. Default is `false`, in which case an operation that mutates (writes to) the table will obtain exclusive access, blocking any concurrent access of the same table until finished. If set to `true`, the table is optimized towards concurrent write access. Different objects of the same table can be mutated (and read) by concurrent processes. This is achieved to some degree at the expense of memory consumption and the performance of sequential access and concurrent reading. The `write_concurrency` option can be combined with the `read_concurrency` option. You typically want to combine these when large concurrent read bursts and large concurrent write bursts are common (see the documentation of the `read_concurrency` option for more information). Note that this option does not change any guarantees about *atomicity and isolation*. Functions that makes such promises over several objects (like `insert/2`) will gain less (or nothing) from this option.

  In current implementation, table type `ordered_set` is not affected by this option. Also, the memory consumption inflicted by both `write_concurrency` and `read_concurrency` is a constant overhead per table. This overhead can be especially large when both options are combined.

- `{read_concurrency,boolean()}` Performance tuning. Default is `false`. When set to `true`, the table is optimized for concurrent read operations. When this option is enabled on a runtime system with SMP support, read operations become much cheaper; especially on systems with multiple physical processors. However, switching between read and write operations becomes more expensive. You typically want to enable this option when concurrent read operations are much more frequent than write operations, or when concurrent reads and writes comes in large read and write bursts (i.e., lots of reads not interrupted by writes, and lots of writes not interrupted by reads). You typically do *not* want to enable this option when the common access pattern is a few read operations interleaved with a few write operations repeatedly. In this case you will get a performance degradation by enabling this option. The `read_concurrency` option can be combined with the `write_concurrency` option. You typically want to combine these when large concurrent read bursts and large concurrent write bursts are common.

- `compressed` If this option is present, the table data will be stored in a more compact format to consume less memory. The downside is that it will make table operations slower. Especially operations that need to inspect entire objects, such as `match` and `select`, will get much slower. The key element is not compressed in current implementation.

\[
\text{next}(\text{Tab}, \text{Key1}) \rightarrow \text{Key2} \mid '\$\text{end\_of\_table}'
\]

Types:

\[
\begin{align*}
\text{Tab} & = \text{tab()} \\
\text{Key1} & = \text{Key2} = \text{term()}
\end{align*}
\]

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, `'$\text{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.

\[
\text{prev}(\text{Tab}, \text{Key1}) \rightarrow \text{Key2} \mid '\$\text{end\_of\_table}'
\]

Types:
Tab = `tab()`
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 = `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 = `continuation()`
  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_spec 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:

```erlang
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:

```erlang
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, Fix) -> true
Types:
   Tab = tab()
   Fix = boolean()
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 (it depends on the internal ordering of the keys). An example:

```erlang
clean_all_with_value(Tab,X) ->
   safe_fixtable(Tab,true),
   clean_all_with_value(Tab,X,ets:first(Tab)),
   safe_fixtable(Tab,false).
```

```erlang
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 = `tab()`
- MatchSpec = `match_spec()`
- Match = `term()`

Matches the objects in the table Tab using a `match_spec`. This is a more general call than the `ets:match/2` and `ets:match_object/2` calls. In its simplest forms the `match_specs` look like this:

- MatchSpec = [MatchFunction]
- MatchFunction = {MatchHead, [Guard], [Result]}
- MatchHead = "Pattern as in ets:match"
- Guard = {"Guardtest name", ...}
- Result = "Term construct"

This means that the `match_spec` is always a list of one or more tuples (of arity 3). The tuples first element should be a pattern as described in the documentation of `ets:match/2`. The second element of the tuple should be a list of 0 or more guard tests (described below). The third element of the tuple should be a list containing a description of the value to actually return. In almost all normal cases the list contains exactly one term which fully describes the value to return for each object.

The return value is constructed using the "match variables" bound in the MatchHead or using the special match variables '$_' (the whole matching object) and '$_$' (all match variables in a list), so that the following `ets:match/2` expression:

```erlang
ets:match(Tab,{'$1','$2','$3'})
```

is exactly equivalent to:

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

- and the following `ets:match_object/2` call:

```erlang
ets:match_object(Tab,{'$1','$2','$1'})
```

is exactly equivalent to

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

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

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

gives the same output as:

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

i.e. all the bound variables in the match head as a list. If tuples are to be constructed, one has to write a tuple of arity 1 with the single element in the tuple being the tuple one wants to construct (as an ordinary tuple could be mistaken for a Guard). Therefore the following call:

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

gives the same output as:

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

- this syntax is equivalent to the syntax used in the trace patterns (see dbg(3)).

The Guards are constructed as tuples where the first element is the name of the test and the rest of the elements are the parameters of the test. To check for a specific type (say a list) of the element bound to the match variable '$1', one would write the test as (is_list, '$1'). If the test fails, the object in the table will not match and the next MatchFunction (if any) will be tried. Most guard tests present in Erlang can be used, but only the new versions prefixed is_ are allowed (like is_float, is_atom etc).

The Guard section can also contain logic and arithmetic operations, which are written with the same syntax as the guard tests (prefix notation), so that a guard test written in Erlang looking like this:

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

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

\[
[\text{is_integer, '$1'\}, \text{is_integer, '$2'\}, \{'\ll', \{'+', '$1', '$2', 4711\}}]
\]

On tables of the ordered_set type, objects are visited in the same order as in a first/next traversal. This means that the match specification will be executed against objects with keys in the first/next order and the corresponding result list will be in the order of that execution.

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

Types:
   Tab = tab()
   MatchSpec = match_spec()
   Limit = integer() >= 1
   Match = term()
   Continuation = continuation()

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 = continuation()
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_count(Tab, MatchSpec) -> NumMatched
Types:
    Tab = tab()
    MatchSpec = match_spec()
    NumMatched = integer() >= 0
Matches the objects in the table Tab using a match_spec. 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.

select_delete(Tab, MatchSpec) -> NumDeleted
Types:
    Tab = tab()
    MatchSpec = match_spec()
    NumDeleted = integer() >= 0
Matches the objects in the table Tab using a match_spec. 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_reverse(Tab, MatchSpec) -> [Match]
Types:
    Tab = tab()
    MatchSpec = match_spec()
    Match = term()
Works like select/2, but returns the list in reverse order for the ordered_set table type. For all other table types, the return value is identical to that of select/2.
select_reverse(Tab, MatchSpec, Limit) ->
    {{[Match], Continuation} | '$end_of_table'}

Types:
    Tab = tab()
    MatchSpec = match_spec()
    Limit = integer() >= 1
    Match = term()
    Continuation = continuation()

Works like select/3, but for the ordered_set table type, traversing is done starting at the last object in Erlang
term order and moves towards the first. For all other table types, the return value is identical to that of select/3.

Note that this is not equivalent to reversing the result list of a select/3 call, as the result list is not only reversed,
but also contains the last Limit matching objects in the table, not the first.

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

Types:
    Continuation = continuation()
    Match = term()

Continues a match started with ets:select_reverse/3. If the table is an ordered_set, the traversal of the
table will continue towards objects with keys earlier in the Erlang term order. The returned list will also contain objects
with keys in reverse order.

For all other table types, the behaviour is exactly that of select/1.

Example:

```erlang
1> T = ets:new(x,[ordered_set]).
2> [ ets:insert(T,{N}) || N <- lists:seq(1,10) ].
...     
3> {R0,C0} = ets:select_reverse(T,{{'_'},[],['$_']},4).
...     
4> R0.
    [{10},{9},{8},{7}]
5> {R1,C1} = ets:select_reverse(C0).
...     
6> R1.
    [{6},{5},{4},{3}]
7> {R2,C2} = ets:select_reverse(C1).
...     
8> R2.
    [{2},{1}]
9> '$end_of_table' = ets:select_reverse(C2).
...     
```

setopts(Tab, Opts) -> true

Types:
Tab = `tab()`

Opts = Opt | [Opt]

Opt = {heir, pid(), HeirData} | {heir, none}

HeirData = term()

Set table options. The only option that currently is allowed to be set after the table has been created is `heir`. The calling process must be the table owner.

slot(Tab, I) -> [Object] | `$end_of_table`

Types:

Tab = `tab()`
I = integer() >= 0
Object = tuple()

This function is mostly for debugging purposes. Normally one should use `first/next` or `last/prev` instead.

Returns all objects in the I:th slot of the table Tab. A table can be traversed by repeatedly calling the function, starting with the first slot I=0 and ending when `$end_of_table` is returned. The function will fail with reason `badarg` if the I argument is out of range.

Unless a table of type `set`, `bag` or `duplicate_bag` is protected using `safe_fixtable/2`, see above, a traversal may fail if concurrent updates are made to the table. If the table is of type `ordered_set`, the function returns a list containing the I:th object in Erlang term order.

tab2file(Tab, Filename) -> ok | {error, Reason}

Types:

Tab = `tab()`
Filename = `file:name()`
Reason = term()

Dumps the table Tab to the file Filename.

Equivalent to `tab2file(Tab, Filename, [])`

tab2file(Tab, Filename, Options) -> ok | {error, Reason}

Types:

Tab = `tab()`
Filename = `file:name()`
Options = [Option]
Option = {extended_info, [ExtInfo]}
ExtInfo = `md5sum` | `object_count`
Reason = term()

Dumps the table Tab to the file Filename.

When dumping the table, certain information about the table is dumped to a header at the beginning of the dump. This information contains data about the table type, name, protection, size, version and if it's a named table. It also contains notes about what extended information is added to the file, which can be a count of the objects in the file or a MD5 sum of the header and records in the file.

The size field in the header might not correspond to the actual number of records in the file if the table is public and records are added or removed from the table during dumping. Public tables updated during dump, and that one
wants to verify when reading, needs at least one field of extended information for the read verification process to be reliable later.

The `extended_info` option specifies what extra information is written to the table dump:

- `object_count`
  
The number of objects actually written to the file is noted in the file footer, why verification of file truncation is possible even if the file was updated during dump.

- `md5sum`
  
The header and objects in the file are checksummed using the built in MD5 functions. The MD5 sum of all objects is written in the file footer, so that verification while reading will detect the slightest bitflip in the file data. Using this costs a fair amount of CPU time.

Whenever the `extended_info` option is used, it results in a file not readable by versions of ets prior to that in stdlib-1.15.1

```
tab2list(Tab) -> [Object]
Types:
  Tab = tab()
  Object = tuple()

Returns a list of all objects in the table Tab.
```

```
tabfile_info(Filename) -> {ok, TableInfo} | {error, Reason}
Types:
  Filename = file:name()
  TableInfo = [InfoItem]
  InfoItem = {name, atom()}
    | {type, Type}
    | {protection, Protection}
    | {named_table, boolean()}
    | {keypos, integer() >= 0}
    | {size, integer() >= 0}
    | {extended_info, [ExtInfo]}
    | {version, {Major :: integer() >= 0, Minor :: integer() >= 0}}
  ExtInfo = md5sum | object_count
  Type = bag | duplicate_bag | ordered_set | set
  Protection = private | protected | public
  Reason = term()

Returns information about the table dumped to file by `tab2file/2` or `tab2file/3`
```

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`. 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. 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 or tab2file/3.

table(Tab) -> QueryHandle

table(Tab, Options) -> QueryHandle

Types:
   Tab = tab()
   QueryHandle = qlc:query_handle()
   Options = [Option] | Option
   Option = {n_objects, NoObjects} | {traverse, TraverseMethod}
   NoObjects = default | integer() >= 1
   TraverseMethod = first_next
                  | last_prev
                  | select
                  | {select, 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:next/2.
   • last_prev. The table is traversed one key at a time by calling ets:prev/2.
   • select. The table is traversed by calling ets:select/1. The option n_objects
determines the number of objects returned (the third argument of select/3); the default is to return 100
objects at a time. The match_spec (the second argument of select/3) is assembled by QLC: simple filters are
translated into equivalent match_specs while more complicated filters have to be applied to all objects returned by select/3 given a match_spec that matches all objects.

- {select, MatchSpec}. As for select the table is traversed by calling ets:select/3 and ets:select/1. The difference is that the match_spec is explicitly given. This is how to state match_specs that cannot easily be expressed within the syntax provided by QLC.

The following example uses an explicit match_spec to traverse the table:

```erlang
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:

```erlang
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:

```erlang
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 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,Result} where Result 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) -> DetsTab

Types:
- Tab = tab()
- DetsTab = dets:tab_name()

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 :: [UpdateOp]) -> [Result]
update_counter(Tab, Key, Incr) -> Result

Types:
    Tab = tab()
    Key = term()
    UpdateOp = {Pos, Incr} | {Pos, Incr, Threshold, SetValue}
    Pos = Incr = Threshold = SetValue = Result = integer()

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 and new/2 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, ElementSpec :: {Pos, Value}) -> boolean()
update_element(Tab, Key, ElementSpec :: [{Pos, Value}]) -> boolean()

Types:
    Tab = tab()
    Key = term()
    Value = term()
    Pos = integer() >= 1

This function provides an efficient way to update one or more elements within an object, without the hassle of having to look up, update and write back the entire object.
It will destructively update the object with key Key in the table Tab. The element at the Pos:th position will be given the value Value.

A list of \( \{\text{Pos}, \text{Value}\} \) can be supplied to update several elements within the same object. If the same position occurs more than one in the list, the last value in the list will be written. If the list is empty or the function fails, no updates will be done at all. The function is also atomic in the sense that other processes can never see any intermediate results.

The function returns true if an object with the key Key was found, false otherwise.

The given Key is used to identify the object by either matching the key of an object in a set table, or compare equal to the key of an object in an ordered_set table (see lookup/2 and new/2 for details on the difference).

The function will fail with reason badarg if:

- the table is not of type set or ordered_set,
- Pos is less than 1 or greater than the object arity, or,
- the element to update is also the key
The functions of this module sort terms on files, merge already sorted files, and check files for sortedness. Chunks containing binary terms are read from a sequence of files, sorted internally in memory and written on temporary files, which are merged producing one sorted file as output. Merging is provided as an optimization; it is faster when the files are already sorted, but it always works to sort instead of merge.

On a file, a term is represented by a header and a binary. Two options define the format of terms on files:

- `{header, HeaderLength}`. HeaderLength determines the number of bytes preceding each binary and containing the length of the binary in bytes. Default is 4. The order of the header bytes is defined as follows: if \( B \) is a binary containing a header only, the size \( Size \) of the binary is calculated as \(<\langle Size:HeaderLength/unit:8\rangle\) = \( B \).
- `{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. An example of a typical ordering function is less than or equal to, `=</2`. Using an ordering function will slow down the sort considerably. The `keysort`, `keymerge` and `keycheck` functions do not accept ordering functions.
- `{unique, boolean()}`. When sorting or merging files, only the first of a sequence of terms that compare equal (`==`) is output if this option is set to `true`. The default value is `false` which implies that all terms that compare equal are output. When checking files for sortedness, a check that no pair of consecutive terms compares equal is done if this option is set to `true`.
- `{tmpdir, TempDirectory}`. The directory where temporary files are put can be chosen explicitly. The default, implied by the value `""`, is to put temporary files on the same directory as the sorted output file. If output is a function (see below), the directory returned by `file:get_cwd()` is used instead. The names of temporary files are derived from the Erlang nodename (`node()`), the process identifier of the current Erlang emulator (`os:getpid()`), and a timestamp (`erlang:now()`); a typical name would be `fs_mynode@myhost_1763_1043_337000_266005.17`, where `17` is a sequence number. Existing files will be overwritten. Temporary files are deleted unless some uncaught EXIT signal occurs.
- `{compressed, boolean()}`. Temporary files and the output file may be compressed. The default value `false` implies that written files are not compressed. Regardless of the value of the `compressed` option, compressed files can always be read. Note that reading and writing compressed files is significantly slower than reading and writing uncompressed files.
- `{size, Size}`. By default approximately 512*1024 bytes read from files are sorted internally. This option should rarely be needed.
- `{no_files, NoFiles}`. By default 16 files are merged at a time. This option should rarely be needed.

As an alternative to sorting files, a function of one argument can be given as input. When called with the argument `read` the function is assumed to return `end_of_input` or `{end_of_input, Value}` when there is no more
input (Value is explained below), or \{Objects, Fun\}, where Objects is a list of binaries or terms depending on the format and Fun is a new input function. Any other value is immediately returned as value of the current call to sort or keysort. Each input function will be called exactly once, and should an error occur, the last function is called with the argument close, the reply of which is ignored.

A function of one argument can be given as output. The results of sorting or merging the input is collected in a non-empty sequence of variable length lists of binaries or terms depending on the format. The output function is called with one list at a time, and is assumed to return a new output function. Any other return value is immediately returned as value of the current call to the sort or merge function. Each output function is called exactly once. When some output function has been applied to all of the results or an error occurs, the last function is called with the argument close, and the reply is returned as value of the current call to the sort or merge function. If a function is given as input and the last input function returns \{end_of_input, Value\}, the function given as output will be called with the argument \{value, Value\}. This makes it easy to initiate the sequence of output functions with a value calculated by the input functions.

As an example, consider sorting the terms on a disk log file. A function that reads chunks from the disk log and returns a list of binaries is used as input. The results are collected in a list of terms.

```erlang
sort(Log) ->
    \{ok, _\} = disk_log:open([{name,Log}, {mode,read_only}]),
    Input = input(Log, start),
    Output = output([],
    Reply = file_sorter:sort(Input, Output, \{format,term\}),
    ok = disk_log:close(Log),
    Reply.

input(Log, Cont) ->
    fun(close) ->
        ok;
        (read) ->
            case disk_log:chunk(Log, Cont) of
                \{error, Reason\} ->
                    \{error, Reason\};
                \{Cont2, Terms\} ->
                    \{Terms, input(Log, Cont2)};
                \{Cont2, Terms, _Badbytes\} ->
                    \{Terms, input(Log, Cont2)};
                eof ->
                    end_of_input
            end.
        end.
    end.

output(L) ->
    fun(close) ->
        lists:append(lists:reverse(L));
        \{Terms\} ->
            output([Terms | L])
        end.
    end.
```

Further examples of functions as input and output can be found at the end of the file_sorter module; the term format is implemented with functions.

The possible values of Reason returned when an error occurs are:

- \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, file:posix()\}}. See file(3) for an explanation of file:posix().
• {premature_eof, FileName}. End-of-file was encountered inside some binary term.

Data Types

file_name() = file:name()
file_names() = [file:name()]
i_command() = read | close
i_reply() = end_of_input
  | {end_of_input, value()}
  | {[object()], infun()}
  | input_reply()
infun() = fun((i_command()) -> i_reply())
input() = file_names() | infun()
input_reply() = term()
o_command() = {value, value()} | [object()] | close
o_reply() = outfun() | output_reply()
object() = term() | binary()
outfun() = fun((o_command()) -> o_reply())
output() = file_name() | outfun()
output_reply() = term()
value() = term()
options() = [option()] | option()
option() = {compressed, boolean()}
  | {header, header_length()}
  | {format, format()}
  | {no_files, no_files()}
  | {order, order()}
  | {size, size()}
  | {tmpdir, tmp_directory()}
  | {unique, boolean()}
format() = binary_term | term | binary | format_fun()
format_fun() = fun((binary()) -> term())
header_length() = integer() >= 1
key_pos() = integer() >= 1 | [integer() >= 1]
no_files() = integer() >= 1
order() = ascending | descending | order_fun()
order_fun() = fun((term(), term()) -> boolean())
size() = integer() >= 0
tmp_directory() = [] | file:name()
reason() = bad_object
  | {bad_object, file:name()}
  | {bad_term, file:name()}
  | {file_error,
      file:name(),
      file:posix()} | badarg | system_limit}
| {premature.eof, file_name()}

Exports

sort(FileName) -> Reply
Types:
  FileName = file_name()
  Reply = ok | {error, reason()} | input_reply() | output_reply()
Sorts terms on files. sort(FileName) is equivalent to sort([FileName], FileName).

sort(Input, Output) -> Reply
sort(Input, Output, Options) -> Reply
Types:
  Input = input()
  Output = output()
  Options = options()
  Reply = ok | {error, reason()} | input_reply() | output_reply()
Sorts terms on files. sort(Input, Output) is equivalent to sort(Input, Output, []).

keysort(KeyPos, FileName) -> Reply
Types:
  KeyPos = key_pos()
  FileName = file_name()
  Reply = ok | {error, reason()} | input_reply() | output_reply()
Sorts tuples on files. keysort(N, FileName) is equivalent to keysort(N, [FileName], FileName).

keysort(KeyPos, Input, Output) -> Reply
keysort(KeyPos, Input, Output, Options) -> Reply
Types:
  KeyPos = key_pos()
  Input = input()
  Output = output()
  Options = options()
  Reply = ok | {error, reason()} | input_reply() | output_reply()
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.

merge(FileNames, Output) -> Reply
merge(FileNames, Output, Options) -> Reply
Types:
FileNames = file_names()
Output = output()
Options = options()
Reply = ok | {error, reason()} | output_reply()

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:
  KeyPos = key_pos()
  FileNames = file_names()
  Output = output()
  Options = options()
  Reply = ok | {error, reason()} | output_reply()

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:
  FileNames = file_names()
  Options = options()
  Reply = {ok, [Result]} | {error, reason()}
  Result = {FileName, TermPosition, term()}
  FileName = file_name()
  TermPosition = integer() >= 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) -> Reply
keycheck(KeyPos, FileNames, Options) -> Reply
Types:
KeyPos = \texttt{key\_pos()}
FileNames = \texttt{file\_names()}
Options = \texttt{options()}
Reply = \{ok, [Result]\} \mid \{error, \texttt{reason()}\}
Result = \{FileName, TermPosition, term()\}
FileName = \texttt{file\_name()}
TermPosition = integer() \geq 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.

\texttt{keycheck(KeyPos, FileName)} is equivalent to \texttt{keycheck(KeyPos, [FileName], [])}.
This module contains utilities on a higher level than the \texttt{file} module. This module does not support "raw" file names (i.e. files whose names do not comply with the expected encoding). Such files will be ignored by the functions in this module.

For more information about raw file names, see the \texttt{file} module.

### Data Types

- \texttt{filename()} = \texttt{file:name()}
- \texttt{dirname()} = \texttt{filename()}
- \texttt{dirname_all()} = \texttt{filename_all()}
- \texttt{filename_all()} = \texttt{file:name_all()}

### Exports

- \texttt{ensure_dir(Name) -> ok | \{error, Reason\}}

  **Types:**
  - \texttt{Name} = \texttt{filename_all()} \| \texttt{dirname_all()}
  - \texttt{Reason} = \texttt{file:posix()}

  The \texttt{ensure_dir/1} function ensures that all parent directories for the given file or directory name \texttt{Name} exist, trying to create them if necessary.

  Returns \texttt{ok} if all parent directories already exist or could be created, or \{\texttt{error}, \texttt{Reason}\} if some parent directory does not exist and could not be created for some reason.

- \texttt{file_size(Filename) -> integer() \geq 0}

  **Types:**
  - \texttt{Filename} = \texttt{filename_all()}

  The \texttt{file_size} function returns the size of the given file.

- \texttt{fold_files(Dir, RegExp, Recursive, Fun, AccIn) -> AccOut}

  **Types:**
  - \texttt{Dir} = \texttt{dirname()}
  - \texttt{RegExp} = \texttt{string()}
  - \texttt{Recursive} = \texttt{boolean()}
  - \texttt{Fun} = fun((\texttt{F} :: \texttt{file:filename()}, \texttt{AccIn}) -> \texttt{AccOut})
  - \texttt{AccIn} = \texttt{AccOut} = \texttt{term()}

  The \texttt{fold_files/5} function folds the function \texttt{Fun} over all (regular) files \texttt{F} in the directory \texttt{Dir} that match the regular expression \texttt{RegExp} (see the \texttt{re} module for a description of the allowed regular expressions). If \texttt{Recursive} is true all sub-directories to \texttt{Dir} are processed. The regular expression matching is done on just the filename without the directory part.
If Unicode file name translation is in effect and the file system is completely transparent, file names that cannot be interpreted as Unicode may be encountered, in which case the \texttt{fun()} must be prepared to handle raw file names (i.e. binaries). If the regular expression contains codepoints beyond 255, it will not match file names that do not conform to the expected character encoding (i.e. are not encoded in valid UTF-8).

For more information about raw file names, see the \texttt{file} module.

\begin{verbatim}
is_dir(Name) -> boolean()
Types:
    Name = filename_all() | dirname_all()
The \texttt{is_dir/1} function returns \texttt{true} if \texttt{Name} refers to a directory, and \texttt{false} otherwise.

is_file(Name) -> boolean()
Types:
    Name = filename_all() | dirname_all()
The \texttt{is_file/1} function returns \texttt{true} if \texttt{Name} refers to a file or a directory, and \texttt{false} otherwise.

is_regular(Name) -> boolean()
Types:
    Name = filename_all()
The \texttt{is_regular/1} function returns \texttt{true} if \texttt{Name} refers to a file (regular file), and \texttt{false} otherwise.

last_modified(Name) -> file:date_time() | 0
Types:
    Name = filename_all() | dirname_all()
The \texttt{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.

wildcard(Wildcard) -> [file:filename()]
Types:
    Wildcard = filename() | dirname()
The \texttt{wildcard/1} function returns a list of all files that match Unix-style wildcard-string \texttt{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:

\begin{itemize}
    \item {?} Matches one character.
    \item {* *} Matches any number of characters up to the end of the filename, the next dot, or the next slash.
    \item {**} Two adjacent *'s used as a single pattern will match all files and zero or more directories and subdirectories.
    \item {[Character1,Character2,...]} Matches any of the characters listed. Two characters separated by a hyphen will match a range of characters. Example: \texttt{[A-Z]} will match any uppercase letter.
\end{itemize}
\end{verbatim}
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:

```erlang
filelib:wildcard("lib/*/ebin/*.beam").
```

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

```erlang
filelib:wildcard("lib/*/src/*.?rl")
```

or the following line

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

can be used.

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

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

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

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

To find all `.erl` or `.hrl` files in any subdirectory, use:

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

wildcard(Wildcard, Cwd) -> [file:filename()]

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.

The module supports raw file names in the way that if a binary is present, or the file name cannot be interpreted according to the return value of `file:native_name_encoding/0`, a raw file name will also be returned. For example `filename:join/1` provided with a path component being a binary (and also not being possible to interpret under the current native file name encoding) will result in a raw file name being returned (the join operation will have been performed of course). For more information about raw file names, see the `file` module.

**Exports**

```erlang
absname(Filename) -> file:filename_all()
```

Types:

- `Fname = file:name_all()`

Converts a relative `Fname` and returns an absolute name. No attempt is made to create the shortest absolute name, because this can give incorrect results on file systems which allow links.

Unix examples:

```
1> pwd().
"/usr/local"
2> filename:absname("foo").
"/usr/local/foo"
3> filename:absname("../x").
"/usr/local/../../x"
4> filename:absname("/").
"/
```

Windows examples:

```
1> pwd().
"D:/usr/local"
2> filename:absname("foo").
"D:/usr/local/foo"
3> filename:absname("../x").
"D:/usr/local/../../x"
4> filename:absname("/").
"D:/
```

```erlang
absname(Filename, Dir) -> file:filename_all()
```

Types:
Filename = Dir = \textit{file:filename\_all()}

This function works like \texttt{absname/1}, except that the directory to which the file name should be made relative is given explicitly in the \texttt{Dir} argument.

\texttt{absname\_join(Dir, Filename) \rightarrow file:filename\_all()}

\textbf{Types:}

\begin{itemize}
  \item \texttt{Dir = Filename = file:filename\_all()}
\end{itemize}

Joins an absolute directory with a relative filename. Similar to \texttt{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 \texttt{Filename} are matched against trailing directory components in \texttt{Dir} so they can be removed from the result - minimizing its length.

\texttt{basename(Filename) \rightarrow file:filename\_all()}

\textbf{Types:}

\begin{itemize}
  \item \texttt{Filename = file:filename\_all()}
\end{itemize}

Returns the last component of \texttt{Filename}, or \texttt{Filename} itself if it does not contain any directory separators.

\texttt{basename(Filename, Ext) \rightarrow file:filename\_all()}

\textbf{Types:}

\begin{itemize}
  \item \texttt{Filename = Ext = file:filename\_all()}
\end{itemize}

Returns the last component of \texttt{Filename} with the extension \texttt{Ext} stripped. This function should be used to remove a specific extension which might, or might not, be there. Use \texttt{rootname(basename(Filename))} to remove an extension that exists, but you are not sure which one it is.

\texttt{dirname(Filename) \rightarrow file:filename\_all()}

\textbf{Types:}

\begin{itemize}
  \item \texttt{Filename = file:filename\_all()}
\end{itemize}

Returns the directory part of \texttt{Filename}.
extension(Filename) -> file:filename_all()
Types:
    Filename = file:name_all()
Returns the file extension of Filename, including the period. Returns an empty string if there is no extension.

flatten(Filename) -> file:filename_all()
Types:
    Filename = file:name_all()
Converts a possibly deep list filename consisting of characters and atoms into the corresponding flat string filename.

join(Components) -> file:filename_all()
Types:
    Components = [file:name_all()]
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.

join(Name1, Name2) -> file:filename_all()
Types:
    Name1 = Name2 = file:name_all()
Joins two file name components with directory separators. Equivalent to join([Name1, Name2]).
filename

nativenamex(Path) -> file:filename_all()
Types:
    Path = file:name_all()
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
Types:
    Path = file:name_all()
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) -> file:filename_all()
rootname(Filename, Ext) -> file:filename_all()
Types:
    Filename = Ext = file:name_all()
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 = file:name_all()
Components = [file:name_all()]

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
Filename = atom() | string()
Rules = [{BinSuffix :: string(), SourceSuffix :: string()}]
Module = module()
SourceFile = string()
Options = [Option]
Option = {i, Path :: string()}
    | {outdir, Path :: 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.

**Warning:**

We don’t recommend using this function. If possible, use beam_lib(3) to extract the abstract code format from the BEAM file and compile that instead.

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:
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. \texttt{SourceFile} is the absolute path to the source file without the ".erl" extension. \texttt{Options} include the options which are necessary to recompile the file with \texttt{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.
An implemenation 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.

This module considers two elements as different if and only if they do not compare equal (==).

**Complexity note**

The complexity on set operations is bounded by either \(O(|S|)\) or \(O(|T| \times \log(|S|))\), where \(S\) is the largest given set, depending on which is fastest for any particular function call. For operating on sets of almost equal size, this implementation is about 3 times slower than using ordered-list sets directly. For sets of very different sizes, however, this solution can be arbitrarily much faster; in practical cases, often between 10 and 100 times. This implementation is particularly suited for accumulating elements a few at a time, building up a large set (more than 100-200 elements), and repeatedly testing for membership in the current set.

As with normal tree structures, lookup (membership testing), insertion and deletion have logarithmic complexity.

**Compatibility**

All of the following functions in this module also exist and do the same thing in the `sets` and `ordsets` modules. That is, by only changing the module name for each call, you can try out different set representations.

- add_element/2
- del_element/2
- filter/2
- fold/3
- from_list/1
- intersection/1
- intersection/2
- is_element/2
- is_set/1
- is_subset/2
- new/0
- size/1
- subtract/2
- to_list/1
- union/1
- union/2

**Data Types**

- `set(Element)`
  
  A GB set.

- `set()`

  `set()` is equivalent to `set(term())`. 

iter(Element)
A GB set iterator.
iter()
iter() is equivalent to iter(term()).

Exports

add(Element, Set1) -> Set2
add_element(Element, Set1) -> Set2
Types:
  Set1 = Set2 = set(Element)
Returns a new set formed from Set1 with Element inserted. If Element is already an element in Set1, nothing is changed.

balance(Set1) -> Set2
Types:
  Set1 = Set2 = set(Element)
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:
  Set1 = Set2 = set(Element)
Returns a new 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:
  Set1 = Set2 = set(Element)
Returns a new 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 = set(Element)
Returns only the elements of Set1 which are not also elements of Set2.

empty() -> Set
new() -> Set
Types:
Set = gb_sets:set()
Returns a new empty set.

filter(Pred, Set1) -> Set2
Types:
   Pred = fun((Element) -> boolean())
   Set1 = Set2 = set(Element)
Filters elements in Set1 using predicate function Pred.

fold(Function, Acc0, Set) -> Acc1
Types:
   Function = fun((Element, AccIn) -> AccOut)
   Acc0 = Acc1 = AccIn = AccOut = Acc
   Set = set(Element)
Folds Function over every element in Set returning the final value of the accumulator.

from_list(List) -> Set
Types:
   List = [Element]
   Set = set(Element)
Returns a set of the elements in List, where List may be unordered and contain duplicates.

from_ordset(List) -> Set
Types:
   List = [Element]
   Set = set(Element)
Turns an ordered-set list List into a set. The list must not contain duplicates.

insert(Element, Set1) -> Set2
Types:
   Set1 = Set2 = set(Element)
Returns a new set formed from Set1 with Element inserted. Assumes that Element is not present in Set1.

intersection(Set1, Set2) -> Set3
Types:
   Set1 = Set2 = Set3 = set(Element)
Returns the intersection of Set1 and Set2.
SetList = [set(Element), ...]
Set = set(Element)
Returns the intersection of the non-empty list of sets.

is_disjoint(Set1, Set2) -> boolean()
Types:
  Set1 = Set2 = set(Element)
Returns true if Set1 and Set2 are disjoint (have no elements in common), and false otherwise.

is_empty(Set) -> boolean()
Types:
  Set = gb_sets:set()
Returns true if Set is an empty set, and false otherwise.

is_member(Element, Set) -> boolean()
is_element(Element, Set) -> boolean()
Types:
  Set = set(Element)
Returns true if Element is an element of Set, otherwise false.

is_set(Term) -> boolean()
Types:
  Term = term()
Returns true if Term appears to be a set, otherwise false.

is_subset(Set1, Set2) -> boolean()
Types:
  Set1 = Set2 = set(Element)
Returns true when every element of Set1 is also a member of Set2, otherwise false.

iterator(Set) -> Iter
Types:
  Set = set(Element)
  Iter = iter(Element)
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) -> Element
Types:
  Set = set(Element)
Returns the largest element in Set. Assumes that Set is nonempty.
next(Iter1) -> {Element, Iter2} | none
Types:
  Iter1 = Iter2 = iter(Element)
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) -> set(Element)
Returns a set containing only the element Element.

size(Set) -> integer() >= 0
Types:
  Set = gb_sets:set()
Returns the number of elements in Set.

smallest(Set) -> Element
Types:
  Set = set(Element)
Returns the smallest element in Set. Assumes that Set is nonempty.

take_largest(Set1) -> {Element, Set2}
Types:
  Set1 = Set2 = set(Element)
Returns {Element, Set2}, where Element is the largest element in Set1, and Set2 is this set with Element deleted. Assumes that Set1 is nonempty.

take_smallest(Set1) -> {Element, Set2}
Types:
  Set1 = Set2 = set(Element)
Returns {Element, Set2}, where Element is the smallest element in Set1, and Set2 is this set with Element deleted. Assumes that Set1 is nonempty.

to_list(Set) -> List
Types:
  Set = set(Element)
  List = [Element]
Returns the elements of Set as a list.

union(Set1, Set2) -> Set3
Types:
  Set1 = Set2 = Set3 = set(Element)
Returns the merged (union) set of Set1 and Set2.
union(\text{SetList}) \rightarrow \text{Set}

Types:
\begin{align*}
\text{SetList} &= [\text{set}(\text{Element}), \ldots] \\
\text{Set} &= \text{set}(\text{Element})
\end{align*}

Returns the merged (union) set of the list of sets.

\textbf{SEE ALSO}
\begin{itemize}
\item \textit{gb_trees}(3), \textit{ordsets}(3), \textit{sets}(3)
\end{itemize}
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. This module considers two keys as different if and only if they do not compare equal (=).

Data structure

Data structure:

- {Size, Tree}, where 'Tree' is composed of nodes of the form:
  - {Key, Value, Smaller, Bigger}, and the "empty tree" node:
  - nil.

There is no attempt to balance trees after deletions. Since deletions do not increase the height of a tree, this should be OK.

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

Performance is comparable to the AVL trees in the Erlang book (and faster in general due to less overhead); the difference is that deletion works for these trees, but not for the book's trees. Behaviour is logarithmic (as it should be).

Data Types

tree(Key, Value)
A GB tree.
tree()

```
tree() is equivalent to tree(term(), term()).
```

iter(Key, Value)
A GB tree iterator.
iter()

```
iter() is equivalent to iter(term(), term()).
```

Exports

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

Types:

```
Tree1 = Tree2 = \text{tree(Key, Value)}
```

Rebalances \text{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.
gb_trees

delete(Key, Tree1) -> Tree2
Types:
   Tree1 = Tree2 = tree(Key, Value)
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:
   Tree1 = Tree2 = tree(Key, Value)
Removes the node with key Key from Tree1 if the key is present in the tree, otherwise does nothing; returns new tree.

empty() -> tree()
Returns a new empty tree

enter(Key, Value, Tree1) -> Tree2
Types:
   Tree1 = Tree2 = tree(Key, Value)
Inserts Key with value Value into Tree1 if the key is not present in the tree, otherwise updates Key to value Value in Tree1. Returns the new tree.

from_orddict(List) -> Tree
Types:
   List = [{Key, Value}]
   Tree = tree(Key, Value)
Turns an ordered list List of key-value tuples into a tree. The list must not contain duplicate keys.

get(Key, Tree) -> Value
Types:
   Tree = tree(Key, Value)
Retrieves the value stored with Key in Tree. Assumes that the key is present in the tree, crashes otherwise.

insert(Key, Value, Tree1) -> Tree2
Types:
   Tree1 = Tree2 = tree(Key, Value)
Inserts Key with value Value into Tree1; returns the new tree. Assumes that the key is not present in the tree, crashes otherwise.

is_defined(Key, Tree) -> boolean()
Types:
   Tree = tree(Key, Value :: term())
Returns true if Key is present in Tree, otherwise false.
is_empty(Tree) -> boolean()
Types:
  Tree = tree()
Returns true if Tree is an empty tree, and false otherwise.

iterator(Tree) -> Iter
Types:
  Tree = tree(Key, Value)
  Iter = iter(Key, Value)
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 = tree(Key, Value :: term())
Returns the keys in Tree as an ordered list.

largest(Tree) -> {Key, Value}
Types:
  Tree = tree(Key, Value)
Returns {Key, Value}, where Key is the largest key in Tree, and Value is the value associated with this key. Assumes that the tree is nonempty.

lookup(Key, Tree) -> none | {value, Value}
Types:
  Tree = tree(Key, Value)
Looks up Key in Tree; returns {value, Value}, or none if Key is not present.

map(Function, Tree1) -> Tree2
Types:
  Function = fun((K :: Key, V1 :: Value1) -> V2 :: Value2)
  Tree1 = tree(Key, Value1)
  Tree2 = tree(Key, Value2)
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) -> none | {Key, Value, Iter2}
Types:
  Iter1 = Iter2 = iter(Key, Value)
Returns {Key, Value, 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) -> integer() >= 0
Types:
    Tree = tree()
Returns the number of nodes in Tree.

smallest(Tree) -> {Key, Value}
Types:
    Tree = tree(Key, Value)
Returns {Key, Value}, where Key is the smallest key in Tree, and Value is the value associated with this key. Assumes that the tree is nonempty.

take_largest(Tree1) -> {Key, Value, Tree2}
Types:
    Tree1 = Tree2 = tree(Key, Value)
Returns {Key, Value, Tree2}, where Key is the largest key in Tree1, Value 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, Value, Tree2}
Types:
    Tree1 = Tree2 = tree(Key, Value)
Returns {Key, Value, Tree2}, where Key is the smallest key in Tree1, Value 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, Value}]
Types:
    Tree = tree(Key :: term(), Value)
Converts a tree into an ordered list of key-value tuples.

update(Key, Value, Tree1) -> Tree2
Types:
    Tree1 = Tree2 = tree(Key, Value)
Updates Key to value Value in Tree1; returns the new tree. Assumes that the key is present in the tree.

values(Tree) -> [Value]
Types:
    Tree = tree(Key :: term(), Value)
Returns the values in Tree as an ordered list, sorted by their corresponding keys. Duplicates are not removed.

SEE ALSO
    gb_sets(3), dict(3)
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>gen_event:add_sup_handler</td>
</tr>
<tr>
<td>gen_event:notify</td>
<td>gen_event:sync_notify</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>gen_event:swap_sup_handler</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.

A *gen_event* process handles system messages as documented in *sys(3)*. 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.

Data Types

handler() = atom() | {atom(), term()}
handler_args() = term()
add_handler_ret() = ok | term() | {'EXIT', term()}
del_handler_ret() = ok | term() | {'EXIT', term()}

Exports

start_link() -> Result
start_link(EventMgrName) -> Result
Types:
   EventMgrName = {local,Name} | {global,GlobalName} | {via,Module,ViaName}
      Name = atom()
      GlobalName = ViaName = term()
      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,GlobalName}, the event manager is registered globally as GlobalName using global:register_name/2. If no name is provided, the event manager is not registered. If EventMgrName={via,Module,ViaName}, the event manager will register with the registry represented by Module. The Module callback should export the functions register_name/2, unregister_name/1, whereis_name/1 and send/2, which should behave like the corresponding functions in global. Thus, {via,global,GlobalName} is a valid reference.

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,GlobalName} | {via,Module,ViaName}
      Name = atom()
      GlobalName = ViaName = term()
      Result = {ok,Pid} | {error,{already_started,Pid}}
      Pid = pid()

Creates a stand-alone event manager process, i.e. an event manager which is not part of a supervision tree and thus has no supervisor.

See start_link/0,1 for a description of arguments and return values.
add_handler(EventMgrRef, Handler, Args) -> Result

Types:

EventMgr = Name | {Name, Node} | {global, GlobalName} | {via, Module, ViaName} | pid()
  Name = Node = atom()
  GlobalName = ViaName = term()
  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, GlobalName}, if the event manager is globally registered.
- {via, Module, ViaName}, if the event manager is registered through an alternative process registry.

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 indicating successful completion, the event manager adds the event handler and this function returns ok. If Module:init/1 fails with Reason or returns {error, Reason}, the event handler is ignored and this function returns {'EXIT', Reason} or {error, Reason}, respectively.

add_sup_handler(EventMgrRef, Handler, Args) -> Result

Types:

EventMgr = Name | {Name, Node} | {global, GlobalName} | {via, Module, ViaName} | pid()
  Name = Node = atom()
  GlobalName = ViaName = term()
  Handler = Module | {Module, Id}
    Module = atom()
    Id = term()
  Args = term()
  Result = ok | {'EXIT', Reason} | term()
  Reason = term()

Adds a new event handler in the same way as add_handler/3 but will also supervise the connection between the event handler and the calling process.
• If the calling process later terminates with Reason, the event manager will delete the event handler by calling Module:terminate/2 with {stop,Reason} as argument.

• If the event handler later is deleted, the event manager sends a message {gen_event_EXIT,Handler,Reason} to the calling process. Reason is one of the following:
  • normal, if the event handler has been removed due to a call to delete_handler/3, or remove_handler has been returned by a callback function (see below).
  • shutdown, if the event handler has been removed because the event manager is terminating.
  • {swapped,NewHandler,Pid}, if the process Pid has replaced the event handler with another event handler NewHandler using a call to swap_handler/3 or swap_sup_handler/3.
  • a term, if the event handler is removed due to an error. Which term depends on the error.

See add_handler/3 for a description of the arguments and return values.

notify(EventMgrRef, Event) -> ok
sync_notify(EventMgrRef, Event) -> ok

Types:
  EventMgrRef = Name | {Name,Node} | {global,GlobalName} | {via,Module,ViaName} | pid()
  Name = Node = atom()
  GlobalName = ViaName = term()
  Event = term()

Sends an event notification to the event manager EventMgrRef. The event manager will call Module:handle_event/2 for each installed event handler to handle the event.

notify is asynchronous and will return immediately after the event notification has been sent. sync_notify is synchronous in the sense that it will return ok after the event has been handled by all event handlers.

See add_handler/3 for a description of EventMgrRef.

Event is an arbitrary term which is passed as one of the arguments to Module:handle_event/2.

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

call(EventMgrRef, Handler, Request) -> Result
call(EventMgrRef, Handler, Request, Timeout) -> Result

Types:
  EventMgrRef = Name | {Name,Node} | {global,GlobalName} | {via,Module,ViaName} | pid()
  Name = Node = atom()
  GlobalName = ViaName = term()
  Handler = Module | {Module,Id}
  Module = atom()
  Id = term()
  Request = term()
  Timeout = int()>0 | infinity
  Result = Reply | {error,Error}
  Reply = term()
  Error = bad_module | {'EXIT',Reason} | term()
  Reason = term()
Makes a synchronous call to the event handler Handler installed in the event manager EventMgrRef by sending a request and waiting until a reply arrives or a timeout occurs. The event manager will call Module:handle_call/2 to handle the request.

See add_handler/3 for a description of EventMgrRef and Handler.

Request is an arbitrary term which is passed as one of the arguments to Module:handle_call/2.

Timeout is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom infinity to wait indefinitely. Default value is 5000. If no reply is received within the specified time, the function call fails.

The return value Reply is defined in the return value of Module:handle_call/2. If the specified event handler is not installed, the function returns {error,bad_module}. If the callback function fails with Reason or returns an unexpected value Term, this function returns {error,{'EXIT',Reason}} or {error,Term}, respectively.

delete_handler(EventMgrRef, Handler, Args) -> Result

Types:

- EventMgrRef = Name | (Name,Node) | {global,GlobalName} | {via,Module,ViaName} | pid()
  - Name = Node = atom()
  - GlobalName = ViaName = term()
  - Handler = Module | {Module,Id}
  - Module = atom()
  - Id = term()
- Args = term()
- Result = term() | {error,module_not_found} | {'EXIT',Reason}
- Reason = term()

Deletes an event handler from the event manager EventMgrRef. The event manager will call Module:terminate/2 to terminate the event handler.

See add_handler/3 for a description of EventMgrRef and Handler.

Args is an arbitrary term which is passed as one of the arguments to Module:terminate/2.

The return value is the return value of Module:terminate/2. If the specified event handler is not installed, the function returns {error,module_not_found}. If the callback function fails with Reason, the function returns {'EXIT',Reason}.

swap_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result

Types:

- EventMgrRef = Name | (Name,Node) | {global,GlobalName} | {via,Module,ViaName} | pid()
  - Name = Node = atom()
  - GlobalName = ViaName = term()
  - Handler1 = Handler2 = Module | {Module,Id}
  - Module = atom()
  - Id = term()
- Args1 = Args2 = term()
- Result = ok | {error,Error}
- Error = {'EXIT',Reason} | term()
Reason = term()

Replaces an old event handler with a new event handler in the event manager EventMgrRef.

See add_handler/3 for a description of the arguments.

First the old event handler Handler1 is deleted. The event manager calls Module1:terminate(Args1, ...), where Module1 is the callback module of Handler1, and collects the return value.

Then the new event handler Handler2 is added and initiated by calling Module2:init({Args2,Term}), where Module2 is the callback module of Handler2 and Term the return value of Module1:terminate/2. This makes it possible to transfer information from Handler1 to Handler2.

The new handler will be added even if the the specified old event handler is not installed in which case Term=error, or if Module1:terminate/2 fails with Reason in which case Term=('EXIT',Reason). The old handler will be deleted even if Module2:init/1 fails.

If there was a supervised connection between Handler1 and a process Pid, there will be a supervised connection between Handler2 and Pid instead.

If Module2:init/1 returns a correct value, this function returns ok. If Module2:init/1 fails with Reason or returns an unexpected value Term, this function returns {error,{'EXIT',Reason}} or {error,Term}, respectively.

swap_sup_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result

Types:
- EventMgrRef = Name | {Name,Node} | {global,GlobalName} | {via,Module,ViaName} | pid()
- Name = Node = atom()
- GlobalName = ViaName = term()
- 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,GlobalName} | {via,Module,ViaName} | pid()
- Name = Node = atom()
- GlobalName = ViaName = term()
- Handler = Module | {Module,Id}
- Module = atom()
- Id = term()

Returns a list of all event handlers installed in the event manager EventMgrRef.
See `add_handler/3` for a description of `EventMgrRef` and `Handler`.

`stop(EventMgrRef) -> ok`

Types:

```
EventMgrRef = Name | {Name,Node} | {global,GlobalName} |
             {via,Module,ViaName} | pid()
Name = Node = atom()
GlobalName = ViaName = term()
```

Terminates the event manager `EventMgrRef`. Before terminating, the event manager will call `Module:terminate(stop,...)` for each installed event handler.

See `add_handler/3` for a description of the argument.

**CALLBACK FUNCTIONS**

The following functions should be exported from a `gen_event` callback module.

**Exports**

`Module:init(InitArgs) -> {ok,State} | {ok,State,hibernate} | {error,Reason}`

Types:

```
InitArgs = Args | {Args,Term}
Args = Term = term()
State = term()
Reason = 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`.

If successful, 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`), waiting for the next event to occur.

`Module:handle_event(Event, State) -> Result`

Types:

```
Event = term()
State = term()
Result = {ok,NewState} | {ok,NewState,hibernate}
          | {swap_handler,Args1,NewState,Handler2,Args2} | remove_handler
NewState = term()
Args1 = Args2 = term()
Handler2 = Module2 | {Module2,Id}
```
Whenever an event manager receives an event sent using \texttt{gen_event:notify/2} or \texttt{gen_event:sync_notify/2}, this function is called for each installed event handler to handle the event.\texttt{Event} is the \texttt{Event} argument of \texttt{notify/sync_notify}.
\texttt{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 \texttt{proc_lib:hibernate/3}), 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 \texttt{Module:terminate(Args1,NewState)} and then \texttt{Module2:init({Args2,Term})} where Term is the return value of \texttt{Module:terminate/2}. See \texttt{gen_event:swap_handler/3} for more information.

If the function returns \texttt{remove_handler} the event handler will be deleted by calling \texttt{Module:terminate(remove_handler,State)}.

\texttt{Module:handle_call(Request, State) -> Result}

\begin{itemize}
  \item \texttt{Request = term()}
  \item \texttt{State = term()}
  \item \texttt{Result} = \{ok,Reply,NewState\} | \{ok,Reply,NewState,hibernate\} | \{swap_handler,Reply,Args1,NewState,Handler2,Args2\} | \{remove_handler, Reply\}
  \item \texttt{Reply = term()}
  \item \texttt{NewState = term()}
  \item \texttt{Args1 = Args2 = term()}
  \item \texttt{Handler2 = Module2 | \{Module2,Id\}}
  \item \texttt{Module2 = atom()}
  \item \texttt{Id = term()}
\end{itemize}

Whenever an event manager receives a request sent using \texttt{gen_event:call/3,4}, this function is called for the specified event handler to handle the request.

\texttt{Request} is the \texttt{Request} argument of \texttt{call}.
\texttt{State} is the internal state of the event handler.

The return values are the same as for \texttt{handle_event/2} except they also contain a term \texttt{Reply} which is the reply given back to the client as the return value of \texttt{call}.

\texttt{Module:handle_info(Info, State) -> Result}

\begin{itemize}
  \item \texttt{Info = term()}
  \item \texttt{State = term()}
  \item \texttt{Result} = \{ok,NewState\} | \{ok,NewState,hibernate\}
\end{itemize}
This function is called for each installed event handler when an event manager receives any other message than an event or a synchronous request (or a system message).

Info is the received message.

See Module:handle_event/2 for a description of State and possible return values.

Module:terminate(Arg, State) -> term()

Types:
  Arg = Args | (stop,Reason) | stop | remove_handler
  | {error,('EXIT',Reason)} | {error,Term}
  Args = Reason = Term = term()

Whenever an event handler is deleted from an event manager, this function is called. It should be the opposite of Module:init/1 and do any necessary cleaning up.

If the event handler is deleted due to a call to gen_event:delete_handler, gen_event:swap_handler/3 or gen_event:swap_sup_handler/3, Arg is the Args argument of this function call.

Arg=(stop,Reason) if the event handler has a supervised connection to a process which has terminated with reason Reason.

Arg=stop if the event handler is deleted because the event manager is terminating.

The event manager will terminate if it is part of a supervision tree and it is ordered by its supervisor to terminate. Even if it is not part of a supervision tree, it will terminate if it receives an 'EXIT' message from its parent.

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/ down grade, i.e. when the instruction \{update,Module,Change,...\} where Change=\{advanced,Extra\} is given in the .appup file. See OTP Design Principles for more information.

In the case of an upgrade, OldVsn is Vsn, and in the case of a downgrade, OldVsn is \{down,Vsn\}. Vsn is defined by the \texttt{vsn} attribute(s) of the old version of the callback module Module. If no such attribute is defined, the version is the checksum of the BEAM file.

\texttt{State} is the internal state of the event handler.

\texttt{Extra} is passed as-is from the \{advanced,Extra\} part of the update instruction.

The function should return the updated internal state.

\texttt{Module:format_status(Opt, \[PDict, State\]) -> Status}

Types:
- \texttt{Opt = normal | terminate}
- \texttt{PDict = \{Key, Value\}}
- \texttt{State = term()}
- \texttt{Status = term()}

\textbf{Note:}
This callback is optional, so event handler modules need not export it. If a handler does not export this function, the gen_event module uses the handler state directly for the purposes described below.

This function is called by a gen_event process when:
- One of \texttt{sys:get_status/1,2} is invoked to get the gen_event status. Opt is set to the atom \texttt{normal} for this case.
- The event handler terminates abnormally and gen_event logs an error. Opt is set to the atom \texttt{terminate} for this case.

This function is useful for customising the form and appearance of the event handler state for these cases. An event handler callback module wishing to customise the \texttt{sys:get_status/1,2} return value as well as how its state appears in termination error logs exports an instance of \texttt{format_status/2} that returns a term describing the current state of the event handler.

\texttt{PDict} is the current value of the gen_event's process dictionary.

\texttt{State} is the internal state of the event handler.

The function should return \texttt{Status}, a term that customises the details of the current state of the event handler. Any term is allowed for \texttt{Status}. The gen_event module uses \texttt{Status} as follows:
- When \texttt{sys:get_status/1,2} is called, gen_event ensures that its return value contains \texttt{Status} in place of the event handler's actual state term.
- When an event handler terminates abnormally, gen_event logs \texttt{Status} in place of the event handler's actual state term.

One use for this function is to return compact alternative state representations to avoid having large state terms printed in logfiles.

\textbf{SEE ALSO}
\texttt{supervisor(3), sys(3)}
A behaviour module for implementing a finite state machine. A generic finite state machine process (gen_fsm) implemented using this module will have a standard set of interface functions and include functionality for tracing and error reporting. It will also fit into an OTP supervision tree. Refer to OTP Design Principles for more information.

A gen_fsm assumes all specific parts to be located in a callback module exporting a pre-defined set of functions. The relationship between the behaviour functions and the callback functions can be illustrated as follows:

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

If a callback function fails or returns a bad value, the gen_fsm will terminate.

A gen_fsm handles system messages as documented in sys(3). The sys module can be used for debugging a gen_fsm. Note that a gen_fsm does not trap exit signals automatically, this must be explicitly initiated in the callback module.

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

The gen_fsm process can go into hibernation (see erlang(3)) if a callback function specifies 'hibernate' instead of a timeout value. This might be useful if the server is expected to be idle for a long time. However this feature should be used with care as hibernation implies at least two garbage collections (when hibernating and shortly after waking up) and is not something you’d want to do between each call to a busy state machine.

**Exports**

```
start_link(Module, Args, Options)  -> Result
start_link(FsmName, Module, Args, Options)  -> Result
```

Types:

- `FsmName` = `{local,Name}` | `{global,GlobalName}` | `{via,Module,ViaName}`
- `Name` = atom()
- `GlobalName` = ViaName = term()
- `Module` = atom()
- `Args` = term()
- `Options` = [Option]
Option = (debug,Dbgs) | (timeout,Time) | (spawn_opt,SOpts)

Dbgs = [Dbg]

Dbg = trace | log | statistics
   | {log_to_file,FileName} | {install,(Func,FuncState)}

SOpts = [SOpt]

SOpt = see erlang:spawn_opt/2,3,4,5

Result = {ok,Pid} | ignore | {error,Error}

Pid = pid()

Error = {already_started,Pid} | term()

Creates a gen_fsm process as part of a supervision tree. The function should be called, directly or indirectly, by the supervisor. It will, among other things, ensure that the gen_fsm is linked to the supervisor.

The gen_fsm process calls Module:init/1 to initialize. To ensure a synchronized start-up procedure, start_link/3,4 does not return until Module:init/1 has returned.

If FsmName={local,Name}, the gen_fsm is registered locally as Name using register/2.
If FsmName={global,GlobalName}, the gen_fsm is registered globally as GlobalName using global:register_name/2.
If FsmName={via,Module,ViaName}, the gen_fsm will register with the registry represented by Module. The Module callback should export the functions register_name/2, unregister_name/1, whereis_name/1 and send/2, which should behave like the corresponding functions in global. Thus, (via,global,GlobalName) is a valid reference.

If no name is provided, the gen_fsm is not registered.

Module is the name of the callback module.

Args is an arbitrary term which is passed as the argument to Module:init/1.

If the option {timeout,Time} is present, the gen_fsm is allowed to spend Time milliseconds initializing or it will be terminated and the start function will return {error,timeout}.

If the option {debug,Dbgs} is present, the corresponding sys function will be called for each item in Dbgs. See sys(3).

If the option {spawn_opt,SOpts} is present, SOpts will be passed as option list to the spawn_opt BIF which is used to spawn the gen_fsm process. See erlang(3).

Note:
Using the spawn option monitor is currently not allowed, but will cause the function to fail with reason badarg.

If the gen_fsm is successfully created and initialized the function returns {ok,Pid}, where Pid is the pid of the gen_fsm. If there already exists a process with the specified FsmName, the function returns {error, {already_started,Pid}} where Pid is the pid of that process.

If Module:init/1 fails with Reason, the function returns {error,Reason}. If Module:init/1 returns {stop,Reason} or ignore, the process is terminated and the function returns {error,Reason} or ignore, respectively.

start(Module, Args, Options) -> Result
start(FsmName, Module, Args, Options) -> Result

Types:
FsmName = {local,Name} | {global,GlobalName} | {via,Module,ViaName}
Name = atom()
GlobalName = ViaName = term()
Module = atom()
Args = term()
Options = [Option]
Option = {debug,Dbgs} | {timeout,Time} | {spawn_opt,SOpts}
Dbgs = [Dbg]
  Dbg = trace | log | statistics
  | {log_to_file,FileName} | {install,{Func,FuncState})
SOpts = [term()]
Result = {ok,Pid} | ignore | {error,Error}
Pid = pid()
Error = {already_started,Pid} | term()

Creates a stand-alone gen_fsm process, i.e. a gen_fsm which is not part of a supervision tree and thus has no supervisor.

See start_link/3,4 for a description of arguments and return values.

send_event(FsmRef, Event) -> ok
Types:
  FsmRef = Name | (Name,Node) | {global,GlobalName} | {via,Module,ViaName} | pid()
  Name = Node = atom()
  GlobalName = ViaName = term()
  Event = term()

Sends an event asynchronously to the gen_fsm FsmRef and returns ok immediately. The gen_fsm will call Module:StateName/2 to handle the event, where StateName is the name of the current state of the gen_fsm.

FsmRef can be:
• the pid,
• Name, if the gen_fsm is locally registered,
• {Name,Node}, if the gen_fsm is locally registered at another node, or
• {global,GlobalName}, if the gen_fsm is globally registered.
• {via,Module,ViaName}, if the gen_fsm is registered through an alternative process registry.

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} | {via,Module,ViaName} | pid()
  Name = Node = atom()
  GlobalName = ViaName = term()
  Event = term()

Sends an event asynchronously to the gen_fsm FsmRef and returns ok immediately. The gen fsm will call Module:handle_event/3 to handle the event.
See send_event/2 for a description of the arguments.

The difference between send_event and send_all_state_event is which callback function is used to handle the event. This function is useful when sending events that are handled the same way in every state, as only one handle_event clause is needed to handle the event instead of one clause in each state name function.

\[
\begin{align*}
sync\_send\_event(FsmRef, Event) & \to \text{Reply} \\
sync\_send\_event(FsmRef, Event, Timeout) & \to \text{Reply}
\end{align*}
\]

Types:

\[
\begin{align*}
FsmRef & = \text{Name} \mid \{\text{Name}, \text{Node}\} \mid \{\text{global}, \text{GlobalName}\} \mid \{\text{via}, \text{Module}, \text{ViaName}\} \mid \text{pid}() \\
\text{Name} & = \text{Node} = \text{atom}() \\
\text{GlobalName} & = \text{ViaName} = \text{term}() \\
\text{Event} & = \text{term}() \\
\text{Timeout} & = \text{int}() > 0 \mid \text{infinity} \\
\text{Reply} & = \text{term}()
\end{align*}
\]

Sends an event to the gen_fsm FsmRef and waits until a reply arrives or a timeout occurs. The gen_fsm will call Module:StateName/3 to handle the event, where StateName is the name of the current state of the gen_fsm. See send_event/2 for a description of FsmRef and Event.

Timeout is an integer greater than zero which specifies how many milliseconds to wait for a reply, or the atom infinity to wait indefinitely. Default value is 5000. If no reply is received within the specified time, the function call fails.

The return value Reply is defined in the return value of Module:StateName/3.

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.

\[
\begin{align*}
sync\_send\_all\_state\_event(FsmRef, Event) & \to \text{Reply} \\
sync\_send\_all\_state\_event(FsmRef, Event, Timeout) & \to \text{Reply}
\end{align*}
\]

Types:

\[
\begin{align*}
FsmRef & = \text{Name} \mid \{\text{Name}, \text{Node}\} \mid \{\text{global}, \text{GlobalName}\} \mid \{\text{via}, \text{Module}, \text{ViaName}\} \mid \text{pid}() \\
\text{Name} & = \text{Node} = \text{atom}() \\
\text{GlobalName} & = \text{ViaName} = \text{term}() \\
\text{Event} & = \text{term}() \\
\text{Timeout} & = \text{int}() > 0 \mid \text{infinity} \\
\text{Reply} & = \text{term}()
\end{align*}
\]

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 for a description of FsmRef and Event. See sync_send_event/3 for a description of Timeout and Reply.

See send_all_state_event/2 for a discussion about the difference between sync_send_event and sync_send_all_state_event.

\[
\begin{align*}
\text{reply}(\text{Caller}, \text{Reply}) & \to \text{true}
\end{align*}
\]

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 or sync_send_all_state_event/2,3, 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.

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.

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 or start_timer/2.

If the timer has already timed out, but the event not yet been delivered, it is cancelled as if it had not timed out, so there will be no false timer event after returning from this function.

Returns the remaining time in ms until the timer would have expired if Ref referred to an active timer, false otherwise.
enter_loop(Module, Options, StateName, StateData)
enter_loop(Module, Options, StateName, StateData, FsmName)
enter_loop(Module, Options, StateName, StateData, Timeout)
enter_loop(Module, Options, StateName, StateData, FsmName, Timeout)

Types:

- **Module** = atom()
- **Options** = [Option]
- **Option** = (debug, Dbgs)
- **Dbgs** = [Dbg]
  - **Dbg** = trace | log | statistics
  - {log_to_file, FileName} | {install, {Func, FuncState}}
- **StateName** = atom()
- **StateData** = term()
- **FsmName** = {local, Name} | {global, GlobalName} | {via, Module, ViaName}
- **Name** = atom()
- **GlobalName** = ViaName = 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). 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. 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. 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()
- **Result** = (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 \texttt{gen_fsm:start/3,4} or \texttt{gen_fsm:start_link/3,4}, this function is called by the new process to initialize.

\texttt{Args} is the \texttt{Args} argument provided to the start function.

If initialization is successful, the function should return \texttt{\{ok,StateName,StateData\}}, \texttt{\{ok,StateName,StateData,Timeout\}} or \texttt{\{ok,StateName,StateData,hibernate\}}, where \texttt{StateName} is the initial state name and \texttt{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 \texttt{Timeout} milliseconds. A timeout is represented by the atom \texttt{timeout} and should be handled by the Module:StateName/2 callback functions. The atom \texttt{infinity} can be used to wait indefinitely, this is the default value.

If \texttt{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).

If something goes wrong during the initialization the function should return \texttt{\{stop,Reason\}}, where \texttt{Reason} is any term, or \texttt{ignore}.

\texttt{Module:StateName(Event, StateData) -> Result}

Types:
\begin{itemize}
\item \texttt{Event} = \texttt{timeout} | \texttt{term()}
\item \texttt{StateData} = \texttt{term()}
\item \texttt{Result} = \texttt{\{next_state,NextStateName,NewStateData\}}
\item \texttt{\{next_state,NextStateName,NewStateData,Timeout\}}
\item \texttt{\{next_state,NextStateName,NewStateData,hibernate\}}
\item \texttt{\{stop,Reason,NewStateData\}}
\end{itemize}

\texttt{NextStateName} = \texttt{atom()}
\texttt{NewStateData} = \texttt{term()}
\texttt{Timeout} = int()>0 | infinity
\texttt{Reason} = \texttt{term()}

There should be one instance of this function for each possible state name. Whenever a gen_fsm receives an event sent using \texttt{gen_fsm:send_event/2}, the instance of this function with the same name as the current state name \texttt{StateName} is called to handle the event. It is also called if a timeout occurs.

\texttt{Event} is either the atom \texttt{timeout}, if a timeout has occurred, or the \texttt{Event} argument provided to \texttt{send_event/2}.

\texttt{StateData} is the state data of the gen_fsm.

If the function returns \texttt{\{next_state,NextStateName,NewStateData\}}, \texttt{\{next_state,NextStateName,NewStateData,Timeout\}} or \texttt{\{next_state,NextStateName,NewStateData,hibernate\}}, the gen_fsm will continue executing with the current state name set to \texttt{NextStateName} 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{\{stop,Reason,NewStateData\}}, the \texttt{gen_fsm} will call Module:terminate(Reason,NewStateData) and terminate.

\texttt{Module:handle_event(Event, StateName, StateData) -> Result}

Types:
\begin{itemize}
\item \texttt{Event} = \texttt{term()}
\end{itemize}
Whenever a `gen_fsm` receives an event sent using `gen_fsm:send_all_state_event/2`, this function is called to handle the event.

`StateName` is the current state name of the `gen_fsm`.

See `Module:StateName/2` for a description of the other arguments and possible return values.

```
Module:StateName(Event, From, StateData) -> Result
```

Types:
- `Event` = term()
- `From` = {pid(),Tag}
- `StateData` = term()
- `Result` = {reply,Reply,NextStateName,NewStateData} | {reply,Reply,NextStateName,NewStateData,Timeout} | {reply,Reply,NextStateName,NewStateData,hibernate} | {next_state,NextStateName,NewStateData} | {next_state,NextStateName,NewStateData,Timeout} | {next_state,NextStateName,NewStateData,hibernate} | {stop,Reason,Reply,NewStateData} | {stop,Reason,NewStateData}
- `Reply` = term()
- `NextStateName` = atom()
- `NewStateData` = term()
- `Timeout` = int()>0 | infinity
- `Reason` = normal | term()

There should be one instance of this function for each possible state name. Whenever a `gen_fsm` receives an event sent using `gen_fsm:sync_send_event/2,3`, the instance of this function with the same name as the current state name `StateName` is called to handle the event.

`Event` is the `Event` argument provided to `sync_send_event/2,3`.

`From` is a tuple `{Pid,Tag}` where `Pid` is the pid of the process which called `sync_send_event/2,3` and `Tag` is a unique tag.

`StateData` is the state data of the `gen_fsm`.

If the function returns `{reply,Reply,NextStateName,NewStateData}`, `{reply,Reply,NextStateName,NewStateData,Timeout}` or `{reply,Reply,NextStateName,NewStateData,hibernate}`, `Reply` will be given back to `From` as the return value of `sync_send_event/2,3`. The `gen_fsm` then continues executing with the current state name.
set to NextStateName and with the possibly updated state data NewStateData. See Module:init/1 for a description of Timeout and hibernate.

If the function returns (next_state,NextStateName,NewStateData), (next_state,NextStateName,NewStateData,Timeout) or (next_state,NextStateName,NewStateData,hibernate), the gen_fsm will continue executing in NextStateName with NewStateData. Any reply to From must be given explicitly using gen_fsm:reply/2.

If the function returns (stop,Reason,Reply,NewStateData), Reply will be given back to From. If the function returns (stop,Reason,NewStateData), any reply to From must be given explicitly using gen_fsm:reply/2. The gen_fsm will then call Module:terminate(Reason,NewStateData) and terminate.

Module:handle_sync_event(Event, From, StateName, StateData) -> Result

Types:

Event  = term()
From   = {pid(),Tag}
StateName  = atom()
StateData  = term()

Result  = {reply,Reply,NextStateName,NewStateData} |
          | {reply,Reply,NextStateName,NewStateData,Timeout}
          | {reply,Reply,NextStateName,NewStateData,hibernate}
          | {next_state,NextStateName,NewStateData} |
          | {next_state,NextStateName,NewStateData,Timeout} |
          | {next_state,NextStateName,NewStateData,hibernate} |
          | {stop,Reason,Reply,NewStateData} | {stop,Reason,NewStateData}

Reply   = term()
NextStateName  = atom()
NewStateData   = term()
Timeout       = int()>0 | infinity
Reason        = term()

Whenever a gen_fsm receives an event sent using gen_fsm:sync_send_all_state_event/2,3, this function is called to handle the event.

StateName is the current state name of the gen_fsm.

See Module:StateName/3 for a description of the other arguments and possible return values.

Module:handle_info(Info, StateName, StateData) -> Result

Types:

Info    = term()
StateName  = atom()
StateData  = term()

Result  = {next_state,NextStateName,NewStateData} |
          | {next_state,NextStateName,NewStateData,Timeout}
          | {next_state,NextStateName,NewStateData,hibernate} |
          | {stop,Reason,NewStateData}

NextStateName  = atom()
NewStateData = term()
Timeout = int()>0 | infinity
Reason = normal | term()

This function is called by a gen_fsm when it receives any other message than a synchronous or asynchronous event (or a system message).

Info is the received message.
See Module:StateName/2 for a description of the other arguments and possible return values.

Module:terminate(Reason, StateName, StateData)
Types:
  Reason = normal | shutdown | {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.

Even if the gen_fsm is not part of a supervision tree, this function will be called if it receives an 'EXIT' message from its parent. Reason will be the same as in the 'EXIT' message.

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.

Module:format_status(Opt, [PDict, StateData]) -> Status

Types:
- Opt = normal | terminate
- PDict = [{Key, Value}]
- StateData = term()
- Status = term()

Note:
This callback is optional, so callback modules need not export it. The gen_fsm module provides a default implementation of this function that returns the callback module state data.

This function is called by a gen_fsm process when:
- One of sys:get_status/1,2 is invoked to get the gen_fsm status. Opt is set to the atom normal for this case.
- The gen_fsm terminates abnormally and logs an error. Opt is set to the atom terminate for this case.

This function is useful for customising the form and appearance of the gen_fsm status for these cases. A callback module wishing to customise the sys:get_status/1,2 return value as well as how its status appears in termination error logs exports an instance of format_status/2 that returns a term describing the current status of the gen_fsm.

PDict is the current value of the gen_fsm's process dictionary.

StateData is the internal state data of the gen_fsm.

The function should return Status, a term that customises the details of the current state and status of the gen_fsm. There are no restrictions on the form Status can take, but for the sys:get_status/1,2 case (when Opt is normal), the recommended form for the Status value is [{data, [{"StateData", Term}]}) where Term provides relevant details of the gen_fsm state data. Following this recommendation isn't required, but doing so will make the callback module status consistent with the rest of the sys:get_status/1,2 return value.

One use for this function is to return compact alternative state data representations to avoid having large state terms printed in logfiles.

SEE ALSO
- gen_event(3), gen_server(3), supervisor(3), proc_lib(3), sys(3)
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></td>
</tr>
<tr>
<td>gen_server:multi_call</td>
<td>Module:handle_call/3</td>
</tr>
<tr>
<td>gen_server:cast</td>
<td>Module:handle_cast/2</td>
</tr>
<tr>
<td>gen_server:abcast</td>
<td>Module:handle_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.

A gen_server handles system messages as documented in *sys(3)*. 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**

```erlang
start_link(Module, Args, Options) -> Result
start_link(ServerName, Module, Args, Options) -> Result
```

Types:

- `ServerName` = `{local,Name} | {global,GlobalName} | {via,Module,ViaName}`
  - `Name` = `atom()`
  - `GlobalName` = `ViaName` = `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. If ServerName=\{via,Module,ViaName\}, the gen_server will register with the registry represented by Module. The Module callback should export the functions register_name/2, unregister_name/1, whereis_name/1 and send/2, which should behave like the corresponding functions in global. Thus, \{via,global,GlobalName\} is a valid reference.

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).

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.

\[\text{start(Module, Args, Options)} \rightarrow \text{Result} \]
\[\text{start(ServerName, Module, Args, Options)} \rightarrow \text{Result} \]

Types:
\[
\text{ServerName} = \{\text{local,Name}\} | \{\text{global,GlobalName}\} | \{\text{via,Module,ViaName}\}
\]
Name = atom()
GlobalName = ViaName = term()
Module = atom()
Args = term()
Options = [Option]
  Option = (debug,Dbgs) | (timeout,Time) | (spawn_opt,SOpts)
  Dbgs = [Dbg]
    Dbg = trace | log | statistics | (log_to_file,FileName) | (install,
                  {Func,FuncState})
  SOpts = [term()]
Result = {ok,Pid} | ignore | {error,Error}
Pid = pid()
Error = {already_started,Pid} | term()

Creates a stand-alone gen_server process, i.e. a gen_server which is not part of a supervision tree and thus has no
supervisor.

See start_link/3,4 for a description of arguments and return values.

call(ServerRef, Request) -> Reply
call(ServerRef, Request, Timeout) -> Reply
Types:
  ServerRef = Name | (Name,Node) | (global,GlobalName) |
              (via,Module,ViaName) | pid()
    Node = atom()
    GlobalName = ViaName = 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.
• (via,Module,ViaName), if the gen_server is registered through an alternative process registry.

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. If the caller catches the failure and continues running, and the server is just late with the reply, it may arrive
at any time later into the caller's message queue. The caller must in this case be prepared for this and discard any such
garbage messages that are two element tuples with a reference as the first element.

The return value Reply is defined in the return value of Module:handle_call/3.

The call may fail for several reasons, including timeout and the called gen_server dying before or during the call.
The ancient behaviour of sometimes consuming the server exit message if the server died during the call while linked to the client has been removed in OTP R12B/Erlang 5.6.

multi_call(Name, Request) -> Result
multi_call(Nodes, Name, Request) -> Result
multi_call(Nodes, Name, Request, Timeout) -> Result

Types:
- Nodes = [Node]
- Node = atom()
- Name = atom()
- Request = term()
- Timeout = int()>=0 | infinity
- Result = {Replies,BadNodes}
- Replies = [{Node,Reply}]
- Reply = term()
- BadNodes = [Node]

makes a synchronous call to all gen_servers locally registered as Name at the specified nodes by first sending a request to every node and then waiting for the replies. The gen_servers will call Module:handle_call/3 to handle the request.

The function returns a tuple {Replies,BadNodes} where Replies is a list of {Node,Reply} and BadNodes is a list of node that either did not exist, or where the gen_server Name did not exist or did not reply.

Nodes is a list of node names to which the request should be sent. Default value is the list of all known nodes [node()|nodes()].

Name is the locally registered name of each gen_server.

Request is an arbitrary term which is passed as one of the arguments to Module:handle_call/3.

Timeout is an integer greater than zero which specifies how many milliseconds to wait for each reply, or the atom infinity to wait indefinitely. Default value is infinity. If no reply is received from a node within the specified time, the node is added to BadNodes.

When a reply Reply is received from the gen_server at a node Node, {Node,Reply} is added to Replies. Reply is defined in the return value of Module:handle_call/3.

Warning:

If one of the nodes is not capable of process monitors, for example C or Java nodes, and the gen_server is not started when the requests are sent, but starts within 2 seconds, this function waits the whole Timeout, which may be infinity.

This problem does not exist if all nodes are Erlang nodes.

To avoid that late answers (after the timeout) pollutes the caller's message queue, a middleman process is used to do the actual calls. Late answers will then be discarded when they arrive to a terminated process.

cast(ServerRef, Request) -> ok

Types:
ServerRef = Name | (Name,Node) | (global,GlobalName) | (via,Module,ViaName) | pid()
    Node = atom()
    GlobalName = ViaName = term()
    Request = term()

Sends an asynchronous request to the gen_server ServerRef and returns ok immediately, ignoring if the destination node or gen_server does not exist. The gen_server will call Module:handle_cast/2 to handle the request.

See call/2,3 for a description of ServerRef.

Request is an arbitrary term which is passed as one of the arguments to Module:handle_cast/2.

abcast(Name, Request) -> abcast
abcast(Nodes, Name, Request) -> abcast

Types:
    Nodes = [Node]
    Node = atom()
    Name = atom()
    Request = term()

Sends an asynchronous request to the gen_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 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
**gen_server**

<table>
<thead>
<tr>
<th>{log_to_file,FileName}</th>
<th>{install,{Func,FuncState}}</th>
</tr>
</thead>
</table>

 State = term()

 ServerName = {local,Name} | {global,GlobalName} | {via,Module,ViaName}
 Name = atom()
 GlobalName = ViaName = 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). 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/3,4`. 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. 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` or `gen_server:start_link/3,4`, 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).

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 or gen_server:multi_call/2,3,4, this function is called to handle the request.

Request is the Request argument provided to call or multi_call.

From is a tuple {Pid,Tag} where Pid is the pid of the client and Tag is a unique tag.

State is the internal state of the gen_server.

If the function returns {reply,Reply,NewState}, {reply,Reply,NewState,Timeout} or {reply,Reply,NewState,hibernate}, Reply will be given back to From as the return value of call/2,3 or included in the return value of multi_call/2,3,4. The gen_server then continues executing with the possibly updated internal state NewState. See Module:init/1 for a description of Timeout and hibernate.

If the functions returns {noreply,NewState}, {noreply,NewState,Timeout} or {noreply,NewState,hibernate}, the gen_server will continue executing with NewState. Any reply to From must be given explicitly using gen_server:reply/2.

If the function returns {stop,Reason,Reply,NewState}, Reply will be given back to From. If the function returns {stop,Reason,NewState}, any reply to From must be given explicitly using gen_server:reply/2. The gen_server will then call Module:terminate(Reason,NewState) and terminate.

Module:handle_cast(Request, State) -> Result
Types:
  Request = term()
  State = term()
  Result = {noreply,NewState} | {noreply,NewState,Timeout}
  | {noreply,NewState,hibernate}
  | {stop,Reason,NewState}
  NewState = term()
  Timeout = int()>=0 | infinity
  Reason = term()

Whenever a gen_server receives a request sent using gen_server:cast/2 or gen_server:abcast/2,3, this function is called to handle the request.
See Module:handle_call/3 for a description of the arguments and possible return values.

Module:handle_info(Info, State) -> Result
Types:
  Info = timeout | term()
  State = term()
  Result = (noreply, NewState) | (noreply, NewState, Timeout)
            | (noreply, NewState, hibernate)
            | (stop, Reason, NewState)
  NewState = term()
  Timeout = int()>=0 | infinity
  Reason = normal | term()

This function is called by a gen_server when a timeout occurs or when it receives any other message than a synchronous or asynchronous request (or a system message).

Info is either the atom timeout, if a timeout has occurred, or the received message.

See Module:handle_call/3 for a description of the other arguments and possible return values.

Module:terminate(Reason, State)
Types:
  Reason = normal | shutdown | {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.

Even if the gen_server is not part of a supervision tree, this function will be called if it receives an 'EXIT' message from its parent. Reason will be the same as in the 'EXIT' message.

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} | {error, Reason}
Types:
  OldVsn = Vsn | {down, Vsn}
  Vsn = term()
  State = NewState = term()
Extra = term()
Reason = 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.

If successful, the function shall return the updated internal state.
If the function returns {error, Reason}, the ongoing upgrade will fail and roll back to the old release.

Module:format_status(Opt, [PDict, State]) -> Status

Types:
Opt = normal | terminate
PDict = [{Key, Value}]
State = term()
Status = term()

Note:
This callback is optional, so callback modules need not export it. The gen_server module provides a default implementation of this function that returns the callback module state.

This function is called by a gen_server process when:
• One of sys:get_status/1,2 is invoked to get the gen_server status. Opt is set to the atom normal for this case.
• The gen_server terminates abnormally and logs an error. Opt is set to the atom terminate for this case.

This function is useful for customising the form and appearance of the gen_server status for these cases. A callback module wishing to customise the sys:get_status/1,2 return value as well as how its status appears in termination error logs exports an instance of format_status/2 that returns a term describing the current status of the gen_server.

PDict is the current value of the gen_server's process dictionary.
State is the internal state of the gen_server.

The function should return Status, a term that customises the details of the current state and status of the gen_server. There are no restrictions on the form Status can take, but for the sys:get_status/1,2 case (when Opt is normal), the recommended form for the Status value is [{data, [{"State", Term}]}] where Term provides relevant details of the gen_server state. Following this recommendation isn't required, but doing so will make the callback module status consistent with the rest of the sys:get_status/1,2 return value.

One use for this function is to return compact alternative state representations to avoid having large state terms printed in logfiles.
SEE ALSO

gen_event(3), gen_fsm(3), supervisor(3), proc_lib(3), sys(3)
Erlang module

This module provides an interface to standard Erlang I/O servers. The output functions all return \texttt{ok} if they are successful, or \texttt{exit} if they are not.

In the following description, all functions have an optional parameter \texttt{IoDevice}. If included, it must be the pid of a process which handles the IO protocols. Normally, it is the \texttt{IoDevice} returned by \texttt{file:open/2}.

For a description of the IO protocols refer to the \textit{STDLIB User's Guide}.

\begin{center}
\textbf{Warning:}
\end{center}

As of R13A, data supplied to the \texttt{put_chars} function should be in the \texttt{unicode:chardata()} format. 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 \texttt{get_chars} and \texttt{get_line} 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 \texttt{file} module instead.

For conversion functions between character encodings, see the \texttt{unicode} module.

\begin{center}
\textbf{Data Types}
\end{center}

\begin{itemize}
    \item \texttt{device()} = \texttt{atom()} | \texttt{pid()}
    \item \texttt{opt_pair()} = \{\texttt{binary, boolean()}
    \item \{\texttt{echo, boolean()}
    \item \{\texttt{expand_fun, expand_fun()}
    \item \{\texttt{encoding, encoding()}
    \end{itemize}

\begin{itemize}
    \item \texttt{expand_fun()} = \texttt{fun((term()) -> \{yes | no, string(), [string(), ...]})}
    \item \texttt{encoding()} = latin1
    \item unicode
    \item utf8
    \item utf16
    \item utf32
    \item \{utf16, big | little\}
    \item \{utf32, big | little\}
    \item \texttt{setopt()} = \texttt{binary | list | opt_pair()}
    \item \texttt{format()} = \texttt{atom()} | \texttt{string()} | \texttt{binary()}
    \item \texttt{location()} = \texttt{erl_scan:location()}
    \item \texttt{prompt()} = \texttt{atom()} | \texttt{unicode:chardata()}
    \item \texttt{server_no_data()} = \{\texttt{error, ErrorDescription :: term()}} | \texttt{eof}
\end{itemize}

What the I/O-server sends when there is no data.
Exports

columns() -> {ok, integer() >= 1} | {error, enotsup}
columns(IoDevice) -> {ok, integer() >= 1} | {error, enotsup}

Types:
   IoDevice = 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 IO devices the function returns {error, enotsup}

put_chars(CharData) -> ok
put_chars(IoDevice, CharData) -> ok

Types:
   IoDevice = device()
   CharData = unicode:chardata()

Writes the characters of CharData to the I/O server (IoDevice).

nl() -> ok
nl(IoDevice) -> ok

Types:
   IoDevice = device()

Writes new line to the standard output (IoDevice).

get_chars(Prompt, Count) -> Data | server_no_data()
get_chars(IoDevice, Prompt, Count) -> Data | server_no_data()

Types:
   IoDevice = device()
   Prompt = prompt()
   Count = integer() >= 0
   Data = string() | unicode:unicode_binary()
   server_no_data() = {error, ErrorDescription :: term()} | eof

Reads Count characters from standard input (IoDevice), prompting it with Prompt. It returns:

Data

   The input characters. If the IO device supports Unicode, the data may represent codepoints larger than 255 (the latin1 range). If the I/O server is set to deliver binaries, they will be encoded in UTF-8 (regardless of if the IO device actually supports Unicode or not).

eof

   End of file was encountered.

{error, ErrorDescription}

   Other (rare) error condition, for instance {error, estale} if reading from an NFS file system.
get_line(Prompt) -> Data | server_no_data()
get_line(IoDevice, Prompt) -> Data | server_no_data()

Types:
   IoDevice = device()
   Prompt = prompt()
   Data = string() | unicode:unicode_binary()
   server_no_data() = {error, ErrorDescription :: term()} | eof

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 IO device supports Unicode, the data may represent codepoints larger than 255 (the latin1 range). If the I/O server is set to deliver binaries, they will be encoded in UTF-8 (regardless of if the IO device actually supports Unicode or not).

eof
End of file was encountered.

{error, ErrorDescription}
Other (rare) error condition, for instance {error, estale} if reading from an NFS file system.

getopts() -> [opt_pair()] | {error, Reason}
getopts(IoDevice) -> [opt_pair()] | {error, Reason}

Types:
   IoDevice = device()
   Reason = 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.

printable_range() -> unicode | latin1
Return the user requested range of printable Unicode characters.

The user can request a range of characters that are to be considered printable in heuristic detection of strings by the shell and by the formatting functions. This is done by supplying +pc <range> when starting Erlang.
Currently the only valid values for `<range>` are `latin1` and `unicode.latin1` means that only code points below 256 (with the exception of control characters etc) will be considered printable. `unicode` means that all printable characters in all unicode character ranges are considered printable by the `io` functions.

By default, Erlang is started so that only the `latin1` range of characters will indicate that a list of integers is a string. The simplest way to utilize the setting is to call `io_lib:printable_list/1`, which will use the return value of this function to decide if a list is a string of printable characters or not.

Note:
In the future, this function may return more values and ranges. It is recommended to use the `io_lib:printable_list/1` function to avoid compatibility problems.

```erlang
default(Opts) -> ok | {error, Reason}
default(IoDevice, Opts) -> ok | {error, Reason}
Types:
   IoDevice = device()
   Opts = [setopt()]
   Reason = term()
Set options for the standard 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 IO device, use the `getopts/1` function.
The options and values supported by the current OTP IO devices are:

- `binary, list or {binary, boolean()}`
  If set in binary mode (binary or `{binary, true}`), the I/O server sends binary data (encoded in UTF-8) as answers to the `get_line`, `get_chars` and, if possible, `get_until` requests (see the I/O protocol description in `STDLIB User's Guide` for details). The immediate effect is that `get_chars/2,3` and `get_line/1,2` return UTF-8 binaries instead of lists of chars for the affected IO device.
  By default, all IO devices in OTP are set in list mode, but the I/O 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 (`group.erl`), the 'oldshell' (`user.erl`) and the file I/O servers.
- `{echo, boolean()}`
  Denotes if the terminal should echo input. Only supported for the standard shell I/O-server (`group.erl`)
- `{expand_fun, expand_fun()}`
  Provide a function for tab-completion (expansion) like the Erlang shell. This function is called when the user presses the TAB key. The expansion is active when calling line-reading functions such as `get_line/1,2`.
  The function is called with the current line, 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 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"):

  ```erlang
  fun("" ) -> {yes, "quit", [[]];
  ```
This option is supported by the standard shell only (group.erl).

\{encoding, latin1 | unicode\}

Specifies how characters are input or output from or to the actual IO 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 I/O functions or how it is sent in the I/O-protocol, it only affects how the 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 IO device will be in \{encoding, unicode\} mode if the IO device supports it. The mode can be changed, if the assumption of the runtime system is wrong, by setting this option.

The 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 (group.erl including werl on Windows®), the 'oldshell' (user.erl) and the file I/O servers.

\{encoding, utf8 | utf16 | utf32 | \{utf16,big\} | \{utf16,little\} | \{utf32,big\} | \{utf32,little\}\}

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)

\texttt{write(Term) -> ok}
\texttt{write(IoDevice, Term) -> ok}

Types:

\texttt{IoDevice = device()}
\texttt{Term = term()}

 Writes the term Term to the standard output (IoDevice).

\texttt{read(Prompt) -> Result}
\texttt{read(IoDevice, Prompt) -> Result}

Types:

\texttt{IoDevice = device()}
\texttt{Prompt = prompt()}
\texttt{Result = \{ok, Term :: term()\} | server_no_data()}

(\_) -> \{no, ",", ["quit"]\} end
| {error, ErrorInfo}
ErrorInfo = `erl_scan:error_info()` | `erl_parse:error_info()`
server_no_data() = {error, ErrorDescription :: term()} | eof

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.

{error, ErrorDescription}
Other (rare) error condition, for instance {error, estale} if reading from an NFS file system.

read(IoDevice, Prompt, StartLocation) -> Result
read(IoDevice, Prompt, StartLocation, Options) -> Result

Types:

IoDevice = `device()`
Prompt = `prompt()`
StartLocation = `location()`
Options = `erl_scan:options()`

Result = {ok, Term :: term(), EndLocation :: `location()`}
| {eof, EndLocation :: `location()`}
| server_no_data()
| {error, ErrorInfo, ErrorLocation :: `location()`}

ErrorInfo = `erl_scan:error_info()` | `erl_parse:error_info()`
server_no_data() = {error, ErrorDescription :: term()} | eof

Reads a term Term from IoDevice, prompting it with Prompt. Reading starts at location StartLocation. The argument Options is passed on as the Options argument of the `erl_scan:tokens/4` function. It returns:

{ok, Term, EndLocation}
The parsing was successful.

{eof, EndLocation}
End of file was encountered.

{error, ErrorInfo, ErrorLocation}
The parsing failed.

{error, ErrorDescription}
Other (rare) error condition, for instance {error, estale} if reading from an NFS file system.
writes the items in \( \text{Data} \) on the standard output (\( \text{IoDevice} \)) in accordance with \( \text{Format} \). \( \text{Format} \) contains plain characters which are copied to the output device, and control sequences for formatting, see below. If \( \text{Format} \) is an atom or a binary, it is first converted to a list with the aid of \text{atom_to_list}/1 or \text{binary_to_list}/1.

The following control sequences are available:

\~

The character \~ is written.

c

The argument is a number that will be interpreted as an ASCII code. The precision is the number of times the character is printed and it defaults to the field width, which in turn defaults to 1. The following example illustrates:

```
1> io:fwrite("\~10.5c\~\~5c\~n", [$a, $b, $c]).
   aaaaa|bbbbb   |ccccc|
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:

```
2> io:fwrite("~tc-n", [1024]).
\x{400}
ok
3> io:fwrite("~c-n", [1024]).
^@
ok
```

**f**

The argument is a float which is written as \[-\]ddd,ddd, where the precision is the number of digits after the decimal point. The default precision is 6 and it cannot be less than 1.

**e**

The argument is a float which is written as \[-\]d.ddde+-ddd, where the precision is the number of digits written. The default precision is 6 and it cannot be less than 2.

**g**

The argument is a float which is written as \[\] if it is \(\geq 0.1\) and \(< 10000.0\). Otherwise, it is written in the e format. The precision is the number of significant digits. It defaults to 6 and should not be less than 2. If the absolute value of the float does not allow it to be written in the f format with the desired number of significant digits, it is also written in the e format.

**s**

Prints the argument with the string syntax. The argument is, if no Unicode translation modifier is present, an iolist(), a binary(), or an atom(). If the Unicode translation modifier (t) is in effect, the argument is unicode:chardata(), meaning that binaries are in UTF-8. The characters are printed without quotes. The string is first truncated by the given precision and then padded and justified to the given field width. The default precision is the field width.

This format can be used for printing any object and truncating the output so it fits a specified field:

```
1> io:fwrite("|~10w|~n", [{hey, hey, hey}]).
|**********|
ok
2> io:fwrite("|~10s|~n", [io_lib:write({hey, hey, hey})]).
|{hey,hey,h|
ok
3> io:fwrite("|~-.8s|~n", [io_lib:write({hey, hey, hey})]).
|{hey,hey  |
ok
```

A list with integers larger than 255 is considered an error if the Unicode translation modifier is not given:

```
4> io:fwrite("~ts-n", [[1024]]).
\x{400}
ok
5> io:fwrite("~s-n", [[1024]])..
** exception exit: {badarg,[{io,format,[<0.26.0>,"~s-n",[[1024]]]},...
```
w Writes data with the standard syntax. This is used to output Erlang terms. Atoms are printed within quotes if they contain embedded non-printable characters, and floats are printed accurately as the shortest, correctly rounded string.

p  Writes the data with standard syntax in the same way as ~w, but breaks terms whose printed representation is longer than one line into many lines and indents each line sensibly. It also tries to detect lists of printable characters and to output these as strings. The Unicode translation modifier is used for determining what characters are printable. For example:

1> T = [{attributes,[[{id,age,1.50000},{mode,explicit},
{typename,"INTEGER"}], [{id,cho},{mode,explicit},{typename,'Cho'}]}],
{typename,'Person'},{tag,{'PRIVATE',3}},{mode,implicit}].
2> io:fwrite("-w-n", [T]).
    
1> io:fwrite("-62p-n", [T]).
    
3> io:fwrite("-62p-n", [T]).
ok
3> io:fwrite("-62p-n", [T]).
ok

The field width specifies the maximum line length. It defaults to 80. The precision specifies the initial indentation of the term. It defaults to the number of characters printed on this line in the same call to io:fwrite or io:format. For example, using T above:

4> io:fwrite("Here T = ~62p-n", [T]).
Here T = [{attributes,[[{id,age,1.5},
{mode,explicit},
    {typename,"INTEGER"}],
    [{id,cho},{mode,explicit},{typename,'Cho'}]}],
{typename,'Person'},{tag,{'PRIVATE',3}},{mode,implicit}].
ok

When the modifier 1 is given no detection of printable character lists will take place. For example:

5> S = [{a,"a"}, {b, "b"}].
6> io:fwrite("-15p-n", [S]).
    
7> io:fwrite("-15lp-n", [S]).
Binaries that look like UTF-8 encoded strings will be output with the string syntax if the Unicode translation modifier is given:

```
9> io:fwrite("~p-n",[[1024]]).
1024
10> io:fwrite("~tp-n",[[1024]]).
"\x{400}"
11> io:fwrite("~tp-n", [<<128,128>>]).
<<128,128>>
12> io:fwrite("~tp-n", [<<208,128>>]).
"\x{400}"/utf8>>
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:

```
8> 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:

```
9> io:fwrite("~62P-n", [T,9]).
[{attributes,[[[id,age,1.5],[mode,explicit],[typename,...]]],
  [[id,cho],[mode,...],[...]]},
  {typename,'Person'},
  {tag,{'PRIVATE',3}},{mode,implicit}]
ok
```

**B**

Writes an integer in base 2 .. 36, the default base is 10. A leading dash is printed for negative integers.

The precision field selects base. For example:

```
1> io:fwrite("~.16B-n", [31]).
1F
ok
2> io:fwrite("~.2B-n", [-19]).
-10011
```
3> 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.

1> io:fwrite("~X-n", [31,"10#"]).
10#31
ok
2> io:fwrite("~.16X-n", [-31,"0x"]).
-0x1F
ok

#
Like B, but prints the number with an Erlang style #-separated base prefix.

1> io:fwrite("~.10#-n", [31]).
10#31
ok
2> io:fwrite("~.16#-n", [-31]).
-16#1F
ok

b
Like B, but prints lowercase letters.

x
Like X, but prints lowercase letters.

+
Like #, but prints lowercase letters.

n
Writes a new line.

i
Ignores the next term.

Returns:
ok
The formatting succeeded.
If an error occurs, there is no output. For example:

1> 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
2> io:fwrite("~s", [65]).

228 | Ericsson AB. All Rights Reserved.: STDLIB
In this example, an attempt was made to output the single character 65 with the aid of the string formatting directive "~s".

```erlang
fread(Prompt, Format) -> Result
fread(IoDevice, Prompt, Format) -> Result
```

Types:

- `IoDevice = device()`
- `Prompt = prompt()`
- `Format = format()`

Result = {ok, Terms :: [term()]}  
| {error, {fread, FreadError :: io_lib:fread_error()}}  
| `server_no_data()`

`server_no_data() = {error, ErrorDescription :: term()} | eof`

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.
- `d` A decimal integer is expected.
- `u` An unsigned integer in base 2..36 is expected. The field width parameter is used to specify base. Leading white-space characters are not skipped.
- `~` An optional sign character is expected. A sign character ~ gives the return value -1. Sign character + or none gives 1. The field width parameter is ignored. Leading white-space characters are not skipped.
- `#` An integer in base 2..36 with Erlang-style base prefix (for example "16#ffff") is expected.
f
A floating point number is expected. It must follow the Erlang floating point number syntax.

s
A string of non-white-space characters is read. If a field width has been specified, this number of characters
are read and all trailing white-space characters are stripped. An Erlang string (list of characters) is returned.
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}}
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, FreadError}
The reading failed and FreadError gives a hint about the error.

{error, ErrorDescription}
The read operation failed and the parameter ErrorDescription 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() -> {ok, integer() >= 1} | {error, enotsup}
rows(IoDevice) -> {ok, integer() >= 1} | {error, enotsup}

Types:

     IoDevice = 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 IO devices the function returns {error, enotsup}

scan_erl_exprs(Prompt) -> Result
scan_erl_exprs(Device, Prompt) -> Result
scan_erl_exprs(Device, Prompt, StartLocation) -> Result
scan_erl_exprs(Device, Prompt, StartLocation, Options) -> Result

Types:

     Device = device()
     Prompt = prompt()
     StartLocation = location()
     Options = erl_scan:options()
     Result = erl_scan:tokens_result() | server_no_data()

server_no_data() = {error, ErrorDescription :: term()} | eof

Reads data from the standard input (IoDevice), prompting it with Prompt. Reading starts at location StartLocation (1). The argument Options is passed on as the Options argument of the erl_scan:tokens/4 function. The data is tokenized as if it were a sequence of Erlang expressions until a final dot (.) is reached. This token is also returned. It returns:

{ok, Tokens, EndLocation}

The tokenization succeeded.

{eof, EndLocation}

End of file was encountered by the tokenizer.

eof

End of file was encountered by the I/O-server.

{error, ErrorInfo, ErrorLocation}

An error occurred while tokenizing.

{error, ErrorDescription}

Other (rare) error condition, for instance {error, estale} if reading from an NFS file system.

Example:
io

23> io:scan_erl_exprs('enter>').
   enter>abc(), "hey".
   {ok, [{atom, 1, abc}, {',', 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) -> Result
scan_erl_form(IoDevice, Prompt) -> Result
scan_erl_form(IoDevice, Prompt, StartLocation) -> Result
scan_erl_form(IoDevice, Prompt, StartLocation, Options) -> Result

Types:
   IoDevice = device()
   Prompt = prompt()
   StartLocation = location()
   Options = erl_scan:options()
   Result = erl_scan:tokens_result() | server_no_data()
   server_no_data() = {error, ErrorDescription :: term()} | eof

Reads data from the standard input (IoDevice), prompting it with Prompt. Starts reading at location StartLocation (1). The argument Options is passed on as the Options argument of the erl_scan:tokens/4 function. 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 dot (.) is reached. This last token is also returned. The return values are the same as for scan_erl_exprs/1,2,3 above.

parse_erl_exprs(Prompt) -> Result
parse_erl_exprs(IoDevice, Prompt) -> Result
parse_erl_exprs(IoDevice, Prompt, StartLocation) -> Result
parse_erl_exprs(IoDevice, Prompt, StartLocation, Options) -> Result

Types:
   IoDevice = device()
   Prompt = prompt()
   StartLocation = location()
   Options = erl_scan:options()
   Result = parse_ret()
   parse_ret() = {ok,
                 ExprList :: erl_parse:abstract_expr(),
                 EndLocation :: location()}
               | {eof, EndLocation :: location()}
               | {error,
                 ErrorInfo :: erl_scan:error_info()
                 | erl_parse/error_info(),
                 ErrorLocation :: location()}

232 | Ericsson AB. All Rights Reserved.: STDLIB
server_no_data() = {error, ErrorDescription :: term()} | eof

Reads data from the standard input (IoDevice), prompting it with Prompt. Starts reading at location StartLocation (1). The argument Options is passed on as the Options argument of the erl_scan:tokens/4 function. The data is tokenized and parsed as if it were a sequence of Erlang expressions until a final dot (.) is reached. It returns:

{ok, ExprList, EndLocation}

The parsing was successful.

eof

End of file was encountered by the I/O-server.

{error, ErrorInfo, ErrorLocation}

An error occurred while tokenizing or parsing.

{error, ErrorDescription}

Other (rare) error condition, for instance {error, estale} if reading from an NFS file system.

Example:

25> io:parse_erl_exprs('enter>').
enter>abc(), "hey".
{ok, [{call,1,\{atom,1,abc\},[]},{string,1,\"hey\"}],2}
26> io:parse_erl_exprs('enter>').
enter>abc("hey".
{error,\{1,erl_parse,\"syntax error before: ",[".'\"]]\},2}

parse_erl_form(Prompt) -> Result
parse_erl_form(IoDevice, Prompt) -> Result
parse_erl_form(IoDevice, Prompt, StartLocation) -> Result
parse_erl_form(IoDevice, Prompt, StartLocation, Options) -> Result

Types:

IoDevice = device()
Prompt = prompt()
StartLocation = location()
Options = erl_scan:options()
Result = parse_form_ret()

parse_form_ret() = {ok,
    AbsForm :: erl_parse:abstract_form(),
    EndLocation :: location()}
| {eof, EndLocation :: location()}
| {error,
    ErrorInfo :: erl_scan:error_info()
    | erl_parse:error_info(),
    ErrorLocation :: location()}
server_no_data() = {error, ErrorDescription :: term()} | eof

Reads data from the standard input (IoDevice), prompting it with Prompt. Starts reading at location StartLocation (1). The argument Options is passed on as the Options argument of the erl_scan:tokens/4 function. 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 dot (.) is reached. It returns:

{ok, AbsForm, EndLocation}  
The parsing was successful.
{eof, EndLocation}  
End of file was encountered by the tokenizer.
eof  
End of file was encountered by the I/O-server.
{error, ErrorInfo, ErrorLocation}  
An error occurred while tokenizing or parsing.
{error, ErrorDescription}  
Other (rare) error condition, for instance {error, estale} if reading from an NFS file system.

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 I/O-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 IO device for error output. Example on a Unix-like operating system:

$ erl -noshell -noinput -eval 'io:format(standard_error,"Error: ~s~n",["error 11"]),\'init:stop().' > /dev/null
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:
A string which describes the error is obtained with the following call:

`Module:format_error(ErrorDescriptor)`
Erlang module

This module contains functions for converting to and from strings (lists of characters). They are used for implementing the functions in the io module. There is no guarantee that the character lists returned from some of the functions are flat, they can be deep lists. lists:flatten/1 can be used for flattening deep lists.

Data Types
chars() = [char() | chars()]
continuation()
A continuation as returned by fread/3.
depth() = -1 | integer() >= 0
fread_error() = atom
| based
| character
| float
| format
| input
| integer
| string
| unsigned
fread_item() = string() | atom() | integer() | float()
latin1_string() = [unicode:latin1_char()]

Exports

nl() -> string()
Returns a character list which represents a new line character.

write(Term) -> chars()
write(Term, Depth) -> chars()
Types:
Term = term()
Depth = depth()
Returns a character list which represents Term. The Depth (-1) argument controls the depth of the structures written. When the specified depth is reached, everything below this level is replaced by "...". For example:

1> lists:flatten(io_lib:write([{1,[2],[3],[4,5],6,7,8,9]})).
"{1,[2],[3],[4,5],6,7,8,9}"
2> lists:flatten(io_lib:write([{1,[2],[3],[4,5],6,7,8,9}, 5]).
"{1,[2],[3],[...],...}"
print(Term) -> \texttt{chars()}
print(Term, Column, LineLength, Depth) -> \texttt{chars()}

Types:
  \begin{itemize}
    \item \texttt{Term} = term()
    \item \texttt{Column} = LineLength = integer() >= 0
    \item \texttt{Depth} = depth()
  \end{itemize}

Also returns a list of characters which represents \texttt{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. \texttt{Column} is the starting column (1), \texttt{LineLength} the maximum line length (80), and \texttt{Depth} (-1) the maximum print depth.

fwrite(Format, Data) -> \texttt{chars()}
format(Format, Data) -> \texttt{chars()}

Types:
  \begin{itemize}
    \item \texttt{Format} = \texttt{io:format()}
    \item \texttt{Data} = [term()]
  \end{itemize}

Returns a character list which represents \texttt{Data} formatted in accordance with \texttt{Format}. See \texttt{io:fwrite/1,2,3} 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. \texttt{~ts} or \texttt{~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 \texttt{string()}, but can well be used in any context where Unicode data is allowed.

fread(Format, String) -> Result

Types:
  \begin{itemize}
    \item \texttt{Format} = \texttt{String} = string()
    \item \texttt{Result} = \{ok, InputList :: [fread_item()], LeftOverChars :: string()\}
    \item \{more, RestFormat :: string(), Nchars :: integer() >= 0, InputStack :: \texttt{chars()}\}
    \item \{error, \{fread, What :: \texttt{fread_error()}\}\}
  \end{itemize}

Tries to read \texttt{String} in accordance with the control sequences in \texttt{Format}. See \texttt{io:fread/3} for a detailed description of the available formatting options. It is assumed that \texttt{String} contains whole lines. It returns:

\{ok, InputList, LeftOverChars\}

The string was read. \texttt{InputList} is the list of successfully matched and read items, and \texttt{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. \texttt{RestFormat} is the remaining format string, \texttt{Nchars} the number of characters scanned, and \texttt{InputStack} is the reversed list of inputs matched up to that point.

\{error, What\}

The read operation failed and the parameter \texttt{What} gives a hint about the error.
Example:

3> io_lib:fread("~f~f~f", "15.6 17.3e-6 24.5").
{ok,[15.6,1.73e-5,24.5],[]}

`fread(Continuation, CharSpec, Format) -> Return`

Types:
- **Continuation** = `continuation()` | `[]`
- **CharSpec** = `string()` | `eof`
- **Format** = `string()`
- **Return** = `{more, Continuation1 :: `continuation()`}
  | `{done, Result, LeftOverChars :: `string()`}
- **Result** = `{ok, InputList :: [fread_item()]}
  | `eof`
  | `{error, {fread, What :: fread_error()}}`

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_string_as_latin1(String) -> latin1_string()
Types:
   String = string()
Returns the list of characters needed to print String as a string. Non-Latin-1 characters are escaped.

write_latin1_string(Latin1String) -> latin1_string()
Types:
   Latin1String = latin1_string()
Returns the list of characters needed to print Latin1String as a string.

write_char(Char) -> chars()
Types:
   Char = char()
Returns the list of characters needed to print a character constant in the Unicode character set.

write_char_as_latin1(Char) -> latin1_string()
Types:
   Char = char()
Returns the list of characters needed to print a character constant in the Unicode character set. Non-Latin-1 characters are escaped.

write_latin1_char(Latin1Char) -> latin1_string()
Types:
   Latin1Char = unicode:latin1_char()
Returns the list of characters needed to print a character constant in the ISO-latin-1 character set.

indentation(String, StartIndent) -> integer()
Types:
   String = string()
   StartIndent = integer()
Returns the indentation if String has been printed, starting at StartIndent.

car_list(Term) -> boolean()
Types:
   Term = term()
Returns true if Term is a flat list of characters in the Unicode range, otherwise it returns false.

latin1_char_list(Term) -> boolean()
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) -> boolean()
Types:
   Term = term()
Returns true if Term is a, possibly deep, list of characters in the Unicode range, otherwise it returns false.

deep_latin1_char_list(Term) -> boolean()
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) -> boolean()
Types:
   Term = term()
Returns true if Term is a flat list of printable characters, otherwise it returns false.
What is a printable character in this case is determined by the +pc start up flag to the Erlang VM. See io:printable_range/0 and erl(1).

printable_latin1_list(Term) -> boolean()
Types:
   Term = term()
Returns true if Term is a flat list of printable ISO-latin-1 characters, otherwise it returns false.

printable_unicode_list(Term) -> boolean()
Types:
   Term = term()
Returns true if Term is a flat list of printable Unicode 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() -> ok
Flushes the message buffer of the current process.

error_message(Format, Args) -> ok
Types:
  Format = io:format()
  Args = [term()]
Prints error message Args in accordance with Format. Similar to io:format/2, see io(3).

progname() -> atom()
Returns the name of the script that started the current Erlang session.

nonl(String1) -> String2
Types:
  String1 = String2 = string()
Removes the last newline character, if any, in String1.

send(To, Msg) -> Msg
Types:
  To = pid() | atom() | {atom(), node()}
  Msg = term()
This function to makes it possible to send a message using the apply/3 BIF.

sendw(To, Msg) -> Msg
Types:
  To = pid() | atom() | {atom(), node()}
  Msg = term()
As send/2, but waits for an answer. It is implemented as follows:

sendw(To, Msg) ->
  To ! {self(),Msg},
The message returned is not necessarily a reply to the message sent.
lists

Erlang module

This module contains functions for list processing.

Unless otherwise stated, all functions assume that position numbering starts at 1. That is, the first element of a list is at position 1.

Two terms T1 and T2 compare equal if T1 == T2 evaluates to true. They match if T1 =:= T2 evaluates to true.

Whenever an ordering function F is expected as argument, it is assumed that the following properties hold of F for all x, y and z:

- if x F y and y F x then x = y (F is antisymmetric);
- if x F y and y F z then x F z (F is transitive);
- x F y or y F x (F is total).

An example of a typical ordering function is less than or equal to, =<>/2.

Exports

all(Pred, List) -> boolean()

Types:
    Pred = fun((Elem :: T) -> boolean())
    List = [T]
    T = term()

Returns true if Pred(Elem) returns true for all elements Elem in List, otherwise false.

any(Pred, List) -> boolean()

Types:
    Pred = fun((Elem :: T) -> boolean())
    List = [T]
    T = term()

Returns true if Pred(Elem) returns true for at least one element Elem in List.

append(ListOfLists) -> List1

Types:
    ListOfLists = [List]
    List = List1 = [T]
    T = 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 = [T]
T = term()

Returns a new list List3 which is made from the elements of List1 followed by the elements of List2. For example:

```prolog
> 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.

```prolog
> lists:concat([doc, '/', file, '.', 3]).
"doc/file.3"
```

delete(Elem, List1) -> List2

Types:
Elem = T
List1 = List2 = [T]
T = term()

Returns a copy of List1 where the first element matching Elem is deleted, if there is such an element.

droplast(List) -> InitList

Types:
List = [T, ...]
InitList = [T]
T = term()

Drops the last element of a List. The list should be non-empty, otherwise the function will crash with a function_clause.

dropwhile(Pred, List1) -> List2

Types:
Pred = fun((Elem :: T) -> boolean())
List1 = List2 = [T]
T = term()

Drops elements Elem from List1 while Pred(Elem) returns true and returns the remaining list.

duplicate(N, Elem) -> List

Types:
   N = integer() >= 0
   Elem = T
   List = [T]
   T = term()

Returns a list which contains N copies of the term Elem. For example:

> lists:duplicate(5, xx).
[xx,xx,xx,xx,xx]

filter(Pred, List1) -> List2

Types:
   Pred = fun((Elem :: T) -> boolean())
   List1 = List2 = [T]
   T = term()

List2 is a list of all elements Elem in List1 for which Pred(Elem) returns true.

filtermap(Fun, List1) -> List2

Types:
   Fun = fun((Elem) -> boolean() | {true, Value})
   List1 = [Elem]
   List2 = [Elem | Value]
   Elem = Value = term()

Calls Fun(Elem) on successive elements Elem of List1. Fun/2 must return either a boolean or a tuple {true, Value}. The function returns the list of elements for which Fun returns a new value, where a value of true is synonymous with {true, Elem}.

That is, filtermap behaves as if it had been defined as follows:

filtermap(Fun, List1) ->
   lists:foldr(fun(Elem, Acc) ->
               case Fun(Elem) of
               false -> Acc;
               true -> [Elem|Acc];
               {true,Value} -> [Value|Acc]
               end,
               end, [], List1).

Example:
```
> lists:filtermap(fun(X) -> case X rem 2 of 0 -> {true, X div 2}; _ -> false end end, [1,2,3,4,5]).
[{true, 1}, {true, 2}]
```

```erlang
deflatlength(DeepList) -> integer() >= 0

Types:
    DeepList = [term() | DeepList]

Equivalent to length(flatten(DeepList)), but more efficient.

deflatmap(Fun, List1) -> List2

Types:
    Fun = fun((A) -> [B])
    List1 = [A]
    List2 = [B]
    A = B = term()

Takes a function from As to lists of Bs, and a list of As (List1) and produces a list of Bs by applying the function to every element in List1 and appending the resulting lists.

That is, `flatmap` behaves as if it had been defined as follows:

```
deflatmap(Fun, List1) ->
    append(map(Fun, List1)).
```

Example:

```
> lists:flatmap(fun(X)->[X,X] end, [a,b,c]).
[a,a,b,b,c,c]
```

deflaten(DeepList) -> List

Types:
    DeepList = [term() | DeepList]
    List = [term()]

Returns a flattened version of DeepList.

deflaten(DeepList, Tail) -> List

Types:
    DeepList = [term() | DeepList]
    Tail = List = [term()]

Returns a flattened version of DeepList with the tail Tail appended.

defoldl(Fun, Acc0, List) -> Acc1

Types:
```
Fun = fun((Elem :: T, AccIn) -> AccOut)
Acc0 = Acc1 = AccIn = AccOut = term()
List = [T]
T = term()

Calls Fun(Elem, AccIn) on successive elements A of List, starting with AccIn == Acc0. Fun/2 must return a new accumulator which is passed to the next call. The function returns the final value of the accumulator. Acc0 is returned if the list is empty. For example:

```
> lists:foldl(fun(X, Sum) -> X + Sum end, 0, [1,2,3,4,5]).
15
> lists:foldl(fun(X, Prod) -> X * Prod end, 1, [1,2,3,4,5]).
120
```

foldr(Fun, Acc0, List) -> Acc1
Types:
Fun = fun((Elem :: T, AccIn) -> AccOut)
Acc0 = Acc1 = AccIn = AccOut = term()
List = [T]
T = term()

Like foldl/3, but the list is traversed from right to left. For example:

```
> P = fun(A, AccIn) -> io:format("~p ", [A]), AccIn end.
#Fun<erl_eval.12.2225172>
> lists:foldl(P, void, [1,2,3]).
1 2 3 void
> lists:foldr(P, void, [1,2,3]).
3 2 1 void
```

foldl/3 is tail recursive and would usually be preferred to foldr/3.

foreach(Fun, List) -> ok
Types:
Fun = fun((Elem :: T) -> term())
List = [T]
T = term()

Calls Fun(Elem) for each element Elem in List. This function is used for its side effects and the evaluation order is defined to be the same as the order of the elements in the list.

keydelete(Key, N, TupleList1) -> TupleList2
Types:
Key = term()
N = integer() >= 1
1..tuple_size(Tuple)
**TupleList1** = **TupleList2** = [Tuple]
Tuple = tuple()

Returns a copy of **TupleList1** where the first occurrence of a tuple whose Nth element compares equal to **Key** is deleted, if there is such a tuple.

**keyfind(Key, N, TupleList)** -> Tuple | false

Types:
Key = term()
N = integer() >= 1
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.

**keymap(Fun, N, TupleList1)** -> TupleList2

Types:
Fun = fun((Term1 :: term()) -> Term2 :: term())
N = integer() >= 1
1..tuple_size(Tuple)
TupleList1 = TupleList2 = [Tuple]
Tuple = tuple()

Returns a list of tuples where, for each tuple in **TupleList1**, the Nth element **Term1** of the tuple has been replaced with the result of calling **Fun(Term1)**.

Examples:

```
> Fun = fun(Atom) -> atom_to_list(Atom) end.
#Fun<erl_eval.6.10732646>
2> lists:keymap(Fun, 2, [{name,jane,22},{name,lizzie,20},{name,lydia,15}]).
[{name,"jane",22},{name,"lizzie",20},{name,"lydia",15}]
```

**keymember(Key, N, TupleList)** -> boolean()

Types:
Key = term()
N = integer() >= 1
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 = integer() >= 1
1..tuple_size(Tuple)
TupleList1 = [T1]
TupleList2 = [T2]
TupleList3 = [(T1 | T2)]
T1 = T2 = 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 = integer() >= 1
  1..tuple_size(Tuple)
  TupleList1 = TupleList2 = [Tuple]
  NewTuple = Tuple
  Tuple = tuple()

Returns a copy of TupleList1 where the first occurrence of a T tuple whose Nth element compares equal to Key is replaced with NewTuple, if there is such a tuple T.

keysearch(Key, N, TupleList) -> {value, Tuple} | false
Types:
  Key = term()
  N = integer() >= 1
  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 = integer() >= 1
  1..tuple_size(Tuple)
TupleList1 = TupleList2 = [Tuple]
Tuple = tuple()

Returns a list containing the sorted elements of the list TupleList1. Sorting is performed on the \(N\)th element of the tuples. The sort is stable.

keystore(Key, N, TupleList1, NewTuple) -> TupleList2

Types:
- Key = term()
- N = integer() \(\geq 1\)
- 1..tuple_size(Tuple)
- TupleList1 = [Tuple]
- TupleList2 = [Tuple, ...]
- NewTuple = Tuple
- Tuple = tuple()

Returns a copy of TupleList1 where the first occurrence of a tuple \(T\) whose \(N\)th element compares equal to Key is replaced with NewTuple, if there is such a tuple \(T\). If there is no such tuple \(T\) a copy of TupleList1 where [NewTuple] has been appended to the end is returned.

keytake(Key, N, TupleList1) -> \{value, Tuple, TupleList2\} | false

Types:
- Key = term()
- N = integer() \(\geq 1\)
- 1..tuple_size(Tuple)
- TupleList1 = TupleList2 = [tuple()]
- Tuple = tuple()

Searches the list of tuples TupleList1 for a tuple whose \(N\)th element compares equal to Key. Returns \{value, Tuple, TupleList2\} if such a tuple is found, otherwise false. TupleList2 is a copy of TupleList1 where the first occurrence of Tuple has been removed.

last(List) -> Last

Types:
- List = [T, ...]
- Last = T
- T = term()

Returns the last element in List.

map(Fun, List1) -> List2

Types:
- Fun = fun((A) -> B)
- List1 = [A]
- List2 = [B]
- A = B = term()

Takes a function from As to Bs, and a list of As and produces a list of Bs by applying the function to every element in the list. This function is used to obtain the return values. The evaluation order is implementation dependent.
mapfoldl(Fun, Acc0, List1) -> {List2, Acc1}
Types:
   Fun = fun((A, AccIn) -> {B, AccOut})
   Acc0 = Acc1 = AccIn = AccOut = term()
   List1 = [A]
   List2 = [B]
   A = B = term()

mapfoldl 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:

```erlang
> 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()

mapfoldr combines the operations of map/2 and foldr/3 into one pass.

max(List) -> Max
Types:
   List = [T, ...]
   Max = T
   T = term()

Returns the first element of List that compares greater than or equal to all other elements of List.

member(Elem, List) -> boolean()
Types:
   Elem = T
   List = [T]
   T = term()

Returns true if Elem matches some element of List, otherwise false.

merge(ListOfLists) -> List1
Types:
ListOfLists = [List]
List = List1 = [T]
T = 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 = [X]
   List2 = [Y]
   List3 = [(X | Y)]
   X = Y = 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) -> boolean())
   List1 = [A]
   List2 = [B]
   List3 = [(A | B)]
   A = B = term()

Returns the sorted list formed by merging List1 and List2. Both List1 and List2 must be sorted according to the ordering function Fun prior to evaluating this function. Fun(A, B) should return true if A compares less than or equal to B in the ordering, false otherwise. When two elements compare equal, the element from List1 is picked before the element from List2.

merge3(List1, List2, List3) -> List4
Types:
   List1 = [X]
   List2 = [Y]
   List3 = [Z]
   List4 = [(X | Y | Z)]
   X = Y = Z = term()

Returns the sorted list formed by merging List1, List2 and List3. All of List1, List2 and List3 must be sorted prior to evaluating this function. When two elements compare equal, the element from List1, if there is such an element, is picked before the other element, otherwise the element from List2 is picked before the element from List3.

min(List) -> Min
Types:
List = [T, ...]
Min = T
T = term()

Returns the first element of List that compares less than or equal to all other elements of List.

nth(N, List) -> Elem
Types:
   N = integer() >= 1
   1..length(List)
   List = [T, ...]
   Elem = T
   T = term()

Returns the Nth element of List. For example:

> lists:nth(3, [a, b, c, d, e]).
c

nthtail(N, List) -> Tail
Types:
   N = integer() >= 0
   0..length(List)
   List = [T, ...]
   Tail = [T]
   T = term()

Returns the Nth tail of List, that is, the sublist of List starting at N+1 and continuing up to the end of the list. For example:

> lists:nthtail(3, [a, b, c, d, e]).
[d,e]
> tl(tl(tl([a, b, c, d, e]))).
[d,e]
> lists:nthtail(0, [a, b, c, d, e]).
[a,b,c,d,e]
> lists:nthtail(5, [a, b, c, d, e]).
[]

partition(Pred, List) -> {Satisfying, NotSatisfying}
Types:
   Pred = fun((Elem :: T) -> boolean())
   List = Satisfying = NotSatisfying = [T]
   T = 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:
See also `splitwith/2` for a different way to partition a list.

**prefix(List1, List2) -> boolean()**

Types:
- `List1 = List2 = [T]`
- `T = term()`

Returns `true` if `List1` is a prefix of `List2`, otherwise `false`.

**reverse(List1) -> List2**

Types:
- `List1 = List2 = [T]`
- `T = term()`

Returns a list with the elements in `List1` in reverse order.

**reverse(List1, Tail) -> List2**

Types:
- `List1 = [T]`
- `Tail = term()`
- `List2 = [T]`
- `T = term()`

Returns a list with the 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 = integer()`
- `Seq = [integer()]`

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:
length(lists:seq(From, To)) == To-From+1
length(lists:seq(From, To, Incr)) == (To-From+Incr) div Incr

Examples:

> lists:seq(1, 10).
[1,2,3,4,5,6,7,8,9,10]
> lists:seq(1, 20, 3).
[1,4,7,10,13,16,19]
> lists:seq(1, 0, 1).
[]
> lists:seq(10, 6, 4).
[]
> lists:seq(1, 1, 0).
[1]

sort(List1) -> List2

Types:

List1 = List2 = [T]
T = term()

Returns a list containing the sorted elements of List1.

sort(Fun, List1) -> List2

Types:

Fun = fun((A :: T, B :: T) -> boolean())
List1 = List2 = [T]
T = term()

Returns a list containing the sorted elements of List1, according to the ordering function Fun. Fun(A, B) should return true if A compares less than or equal to B in the ordering, false otherwise.

split(N, List1) -> {List2, List3}

Types:

N = integer() >= 0
0..length(List1)
List1 = List2 = List3 = [T]
T = term()

Splits List1 into List2 and List3. List2 contains the first N elements and List3 the rest of the elements (the Nth tail).

splitwith(Pred, List) -> {List1, List2}

Types:

Pred = fun((T) -> boolean())
List = List1 = List2 = [T]
T = term()

Partitions List into two lists according to Pred. splitwith/2 behaves as if it is defined as follows:
splitwith(Pred, List) ->
   {takewhile(Pred, List), dropwhile(Pred, List)}.

Examples:

> lists:splitwith(fun(A) -> A rem 2 == 1 end, [1,2,3,4,5,6,7]).
   {{[1]],[2,3,4,5,6,7]}
> lists:splitwith(fun(A) -> is_atom(A) end, [a,b,1,c,d,2,3,4,e]).
   {[a,b],[1,c,d,2,3,4,e]}

See also partition/2 for a different way to partition a list.

sublist(List1, Len) -> List2

Types:
   List1 = List2 = [T]
   Len = integer() >= 0
   T = term()

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 = [T]
   Start = integer() >= 1
   L..(length(List1)+1)
   Len = integer() >= 0
   T = term()

Returns the sub-list of List1 starting at Start and with (max) Len elements. It is not an error for Start+Len to exceed the length of the list.

> lists:sublist([1,2,3,4], 2, 2).
   [2,3]
> lists:sublist([1,2,3,4], 2, 5).
   [2,3,4]
> lists:sublist([1,2,3,4], 5, 2).
   []

subtract(List1, List2) -> List3

Types:
   List1 = List2 = List3 = [T]
   T = term()

Returns a new list List3 which is a copy of List1, subjected to the following procedure: for each element in List2, its first occurrence in List1 is deleted. For example:
lists:subtract("123212", "212").
"312".

lists:subtract(A, B) is equivalent to A -- B.

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 is a much better choice if both lists are long.)

suffix(List1, List2) -> boolean()
Types:
   List1 = List2 = [T]
   T = term()
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 :: T) -> boolean())
   List1 = List2 = [T]
   T = 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 = integer() >= 1
   1..tuple_size(Tuple)
   TupleList1 = [T1]
   TupleList2 = [T2]
   TupleList3 = [(T1 | T2)]
   T1 = T2 = 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 = integer() >= 1
    1..tuple_size(Tuple)
    TupleList1 = TupleList2 = [Tuple]
    Tuple = tuple()

Returns a list containing the sorted elements of the list TupleList1 where all but the first tuple of the tuples comparing equal have been deleted. Sorting is performed on the Nth element of the tuples.

umerge(ListOfLists) -> List1

Types:
    ListOfLists = [List]
    List = List1 = [T]
    T = 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 = [X]
    List2 = [Y]
    List3 = [(X | Y)]
    X = Y = 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) -> boolean())
    List1 = [A]
    List2 = [B]
    List3 = [(A | B)]
    A = B = term()

Returns the sorted list formed by merging List1 and List2. Both List1 and List2 must be sorted according to the ordering function Fun and contain no duplicates prior to evaluating this function. Fun (A, B) should return true if A compares less than or equal to 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 = [X]
List2 = [Y]
List3 = [Z]
List4 = [(X | Y | Z)]
X = Y = Z = 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 = [{A, B}]
- List2 = [A]
- List3 = [B]
- A = B = 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 = [{A, B, C}]
- List2 = [A]
- List3 = [B]
- List4 = [C]
- A = B = C = 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 = [T]
- T = 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((T, T) -> boolean())
- List1 = List2 = [T]
- T = term()

Returns a list which contains the sorted elements of List1 where all but the first element of the elements comparing equal according to the ordering function Fun have been deleted. Fun(A, B) should return true if A compares less than or equal to B in the ordering, false otherwise.
zip(List1, List2) -> List3
Types:

List1 = [A]
List2 = [B]
List3 = [{A, B}]
A = B = 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 = [A]
List2 = [B]
List3 = [C]
List4 = [{A, B, C}]
A = B = C = term()

"Zips" three lists of equal length into one list of three-tuples, where the first element of each tuple is taken from the first list, the second element is taken from corresponding element in the second list, and the third element is taken from the corresponding element in the third list.

zipwith(Combine, List1, List2) -> List3
Types:

Combine = fun((X, Y) -> T)
List1 = [X]
List2 = [Y]
List3 = [T]
X = Y = T = term()

Combine the elements of two lists of equal length into one list. For each pair X, Y of list elements from the two lists, the element in the result list will be Combine(X, Y).

zipwith(fun(X, Y) -> {X,Y} end, List1, List2) is equivalent to zip(List1, List2).

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 Combine(X, Y, Z).

zipwith3(fun(X, Y, Z) -> {X,Y,Z} end, List1, List2, List3) is equivalent to zip3(List1, List2, List3).

Examples:

```
> lists:zipwith3(fun(X, Y, Z) -> X+Y+Z end, [1,2,3], [4,5,6], [7,8,9]).
[12,15,18]
> lists:zipwith3(fun(X, Y, Z) -> [X,Y,Z] end, [a,b,c], [x,y,z], [1,2,3]).
[[a,x,1],[b,y,2],[c,z,3]]
```
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, ..... 

Data Types

args()

Term to be sent to gen_event:add_handler/3.

Exports

init(Dir, MaxBytes, MaxFiles) -> Args
init(Dir, MaxBytes, MaxFiles, Pred) -> Args

Types:

Dir = file:filename()
MaxBytes = integer() >= 0
MaxFiles = 1..255
Pred = fun((Event :: term()) -> boolean())
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), rb(3)
This module contains functions for maps processing.

**Exports**

find(\(Key, \text{Map}\)) \(\rightarrow\) \{ok, \(Value\)\} \(\mid\) error

*Types:*

\[
\begin{align*}
\text{Key} & \rightarrow \text{term}() \\
\text{Map} & \rightarrow \#\{\} \\
\text{Value} & \rightarrow \text{term}()
\end{align*}
\]

Returns a tuple \{ok, \(Value\)\} where \(Value\) is the value associated with \(Key\), or error if no value is associated with \(Key\) in \(Map\).

*Example:*

```erlang
> Map = #{"hi" => 42},
Key = "hi",
maps:find(Key,Map).
\{ok,42\}
```

fold(Fun, Init, Map) \(\rightarrow\) Acc

*Types:*

\[
\begin{align*}
\text{Fun} & \rightarrow \text{fun}(\langle K, V, \text{AccIn}\rangle \rightarrow \text{AccOut}) \\
\text{Init} & \rightarrow \text{Acc} = \text{AccIn} = \text{AccOut} \rightarrow \text{term}() \\
\text{Map} & \rightarrow \#\{\} \\
\text{K} & \rightarrow \text{V} \rightarrow \text{term}()
\end{align*}
\]

Calls \(F(K, V, \text{AccIn})\) for every \(K\) to value \(V\) association in \(Map\) in arbitrary order. The function \text{fun} \(F/3\) must return a new accumulator which is passed to the next successive call. \text{maps:fold/3} returns the final value of the accumulator. The initial accumulator value \(Init\) is returned if the map is empty.

*Example:*

```erlang
> Fun = fun(K,V,AccIn) when is_list(K) -> AccIn + V end,
Map = #{"k1" => 1, "k2" => 2, "k3" => 3},
maps:fold(Fun,0,Map).
6
```

from_list(List) \(\rightarrow\) Map

*Types:*
The function takes a list of key-value tuples elements and builds a map. The associations may be in any order and both keys and values in the association may be of any term. If the same key appears more than once, the latter (rightmost) value is used and the previous values are ignored.

Example:

```erlang
> List = [{“a”, ignored}, {1337, “value two”}, {42, value_three}, {“a”, 1}],
  maps:from_list(List).
#{42 => value_three, 1337 => “value two”, “a” => 1}
```

get(Key, Map) -> Value

Types:
- Key = term()
- Map = {}
- Value = term()

Returns the value Value associated with Key if Map contains Key. If no value is associated with Key then the call will fail with an exception.

Example:

```erlang
> Key = 1337,
  Map = #{42 => value_two, 1337 => “value one”, “a” => 1},
  maps:get(Key, Map).
“value one”
```

get(Key, Map, Default) -> Value | Default

Types:
- Key = term()
- Map = {}
- Value = Default = term()

Returns the value Value associated with Key if Map contains Key. If no value is associated with Key then returns Default.

Example:

```erlang
> Map = #{key1 => val1, key2 => val2}.
#{key1 => val1, key2 => val2}
> maps:get(key1, Map, “Default value”).
val1
> maps:get(key3, Map, “Default value”).
“Default value”
```
is_key(Key, Map) -> boolean()

Types:
    Key = term()
    Map = #{}

Returns true if map Map contains Key and returns false if it does not contain the Key. The function will fail with an exception if Map is not a Map.

Example:

   > Map = #{"42" => value}.
   #{"42" => value}.
   > maps:is_key("42",Map).
   true
   > maps:is_key(value,Map).
   false

keys(Map) -> Keys

Types:
    Map = #{}
    Keys = [Key]
    Key = term()

Returns a complete list of keys, in arbitrary order, which resides within Map.

Example:

   > Map = #{42 => value_three,1337 => "value two","a" => 1},
      maps:keys(Map).
   [42,1337,"a"]

map(Fun, Map1) -> Map2

Types:
    Fun = fun((K, V1) -> V2)
    Map1 = Map2 = #{}
    K = V1 = V2 = term()

The function produces a new map Map2 by calling the function fun F(K, V1) for every K to value V1 association in Map1 in arbitrary order. The function fun F/2 must return the value V2 to be associated with key K for the new map Map2.

Example:

   > Fun = fun(K,V1) when is_list(K) -> V1*2 end,
      Map = #{"k1" => 1, "k2" => 2, "k3" => 3},
      maps:map(Fun,Map).
      #{"k1" => 2,"k2" => 4,"k3" => 6}
merge(Map1, Map2) -> Map3
Types:
    Map1 = Map2 = Map3 = {}
Merges two maps into a single map Map3. If two keys exists in both maps the value in Map1 will be superseded by the value in Map2.
Example:

> Map1 = #{a => "value_one", b => "value_two"},
    Map2 = #{a => 1, c => 2},
    maps:merge(Map1,Map2).
# {a => 1, b => "value_two", c => 2}

new() -> Map
Types:
    Map = {}
Returns a new empty map.
Example:

> maps:new().
{}

put(Key, Value, Map1) -> Map2
Types:
    Key = Value = term()
    Map1 = Map2 = {}
Associates Key with value Value and inserts the association into map Map2. If key Key already exists in map Map1, the old associated value is replaced by value Value. The function returns a new map Map2 containing the new association and the old associations in Map1.
Example:

> Map = #{"a" => 1}.
#{"a" => 1}
> maps:put("a", 42, Map).
#{"a" => 42}
> maps:put("b", 1337, Map).
#{"a" => 1, "b" => 1337}

remove(Key, Map1) -> Map2
Types:
Key = term()
Map1 = Map2 = {}  
The function removes the Key, if it exists, and its associated value from Map1 and returns a new map Map2 without key Key.
Example:

> Map = #{"a" => 1}.
#{"a" => 1}
> maps:remove("a",Map).
#{
> maps:remove("b",Map).
#{"a" => 1}

size(Map) -> integer() >= 0
Types:
Map = {}
The function returns the number of key-value associations in the Map. This operation happens in constant time.
Example:

> Map = #{42 => value_two,1337 => "value one","a" => 1},
maps:size(Map).
3

to_list(Map) -> [{Key, Value}]
Types:
Map = {}
Key = Value = term()
The function returns a list of pairs representing the key-value associations of Map, where the pairs, [{K1,V1}, ..., (Kn,Vn)], are returned in arbitrary order.
Example:

> Map = #{42 => value_three,1337 => "value two","a" => 1},
maps:to_list(Map).
[{42,value_three},{1337,"value two"},{"a",1}]

update(Key, Value, Map1) -> Map2
Types:
Key = Value = term()
Map1 = Map2 = {}  
If Key exists in Map1 the old associated value is replaced by value Value. The function returns a new map Map2 containing the new associated value. If Key does not exist in Map1 an exception is generated.
Example:
values(Map) -> Values
Types:
    Map = {}
    Values = [Value]
    Value = term()

Returns a complete list of values, in arbitrary order, contained in map M.

Example:

```plaintext
> Map = #{42 => value_three, 1337 => "value two", "a" => 1},
maps:values(Map).
[value_three, "value two", 1]
```

with(Ks, Map1) -> Map2
Types:
    Ks = [K]
    Map1 = Map2 = {}
    K = term()

Returns a new map Map2 with the keys K1 through Kn and their associated values from map Map1. Any key in Ks that does not exist in Map1 are ignored.

Example:

```plaintext
> Map = #{42 => value_three, 1337 => "value two", "a" => 1},
Ks = ["a", 42, "other key"],
maps:with(Ks, Map).
#{42 => value_three, "a" => 1}
```

without(Ks, Map1) -> Map2
Types:
    Ks = [K]
    Map1 = Map2 = {}
    K = term()

Returns a new map Map2 without the keys K1 through Kn and their associated values from map Map1. Any key in Ks that does not exist in Map1 are ignored.

Example:

```plaintext
> Map = #{42 => value_three, 1337 => "value two", "a" => 1},
Ks = ["a", 42, "other key"],
maps:without(Ks, Map).
```
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)} \rightarrow \texttt{float()}
\texttt{cos(X)} \rightarrow \texttt{float()}
\texttt{tan(X)} \rightarrow \texttt{float()}
\texttt{asin(X)} \rightarrow \texttt{float()}
\texttt{acos(X)} \rightarrow \texttt{float()}
\texttt{atan(X)} \rightarrow \texttt{float()}
\texttt{atan2(Y, X)} \rightarrow \texttt{float()}
\texttt{sinh(X)} \rightarrow \texttt{float()}
\texttt{cosh(X)} \rightarrow \texttt{float()}
\texttt{tanh(X)} \rightarrow \texttt{float()}
\texttt{asinh(X)} \rightarrow \texttt{float()}
\texttt{acosh(X)} \rightarrow \texttt{float()}
\texttt{atanh(X)} \rightarrow \texttt{float()}
\texttt{exp(X)} \rightarrow \texttt{float()}
\texttt{log(X)} \rightarrow \texttt{float()}
\texttt{log10(X)} \rightarrow \texttt{float()}
\texttt{pow(X, Y)} \rightarrow \texttt{float()}
\texttt{sqrt(X)} \rightarrow \texttt{float()}

Types:
\begin{align*}
Y = X &= \texttt{number()}
\end{align*}
A collection of math functions which return floats. Arguments are numbers.

\texttt{erf(X)} \rightarrow \texttt{float()}

Types:
\begin{align*}
X &= \texttt{number()}
\end{align*}
Returns the error function of \texttt{X}, where
\( \text{erf}(X) = \frac{2}{\sqrt{\pi}} \int_0^X e^{-t^2} \, dt. \)

\( \text{erfc}(X) \rightarrow \text{float()} \)

Types:

\[ X = \text{number()} \]

\( \text{erfc}(X) \) returns \( 1.0 - \text{erf}(X) \), computed by methods that avoid cancellation for large \( X \).

**Bugs**

As these are the C library, the bugs are the same.
ms_transform

Erlang module

This module implements the parse_transform that makes calls to ets and dbg:fun2ms/1 translate into literal match specifications. It also implements the back end for the same functions when called from the Erlang shell.

The translations from fun's to match_specs is accessed through the two "pseudo functions" ets:fun2ms/1 and dbg:fun2ms/1.

Actually this introduction is more or less an introduction to the whole concept of match specifications. Since everyone trying to use ets:select or dbg seems to end up reading this page, it seems in good place to explain a little more than just what this module does.

There are some caveats one should be aware of, please read through the whole manual page if it's the first time you're using the transformations.

Match specifications are used more or less as filters. They resemble usual Erlang matching in a list comprehension or in a fun used in conjunction with lists:foldl etc. The syntax of pure match specifications is somewhat awkward though, as they are made up purely by Erlang terms and there is no syntax in the language to make the match specifications more readable.

As the match specifications execution and structure is quite like that of a fun, it would for most programmers be more straight forward to simply write it using the familiar fun syntax and having that translated into a match specification automatically. Of course a real fun is more powerful than the match specifications allow, but bearing the match specifications in mind, and what they can do, it's still more convenient to write it all as a fun. This module contains the code that simply translates the fun syntax into match_spec terms.

Let's start with an ets example. Using ets:select and a match specification, one can filter out rows of a table and construct a list of tuples containing relevant parts of the data in these rows. Of course one could use ets:foldl instead, but the select call is far more efficient. Without the translation, one has to struggle with writing match specifications terms to accommodate this, or one has to resort to the less powerful ets:match(_object) calls, or simply give up and use the more inefficient method of ets:foldl. Using the ets:fun2ms transformation, a ets:select call is at least as easy to write as any of the alternatives.

As an example, consider a simple table of employees:

```erlang
-record(emp, {empno,     %Employee number as a string, the key
    surname,   %Surname of the employee
    givenname, %Given name of employee
    dept,      %Department one of {dev,sales,prod,adm}
    empyear}). %Year the employee was employed
```

We create the table using:

```erlang
ets:new(emp_tab,[{keypos,#emp.empno},named_table,ordered_set]).
```

Let's also fill it with some randomly chosen data for the examples:

```erlang
[{emp,"011103","Black","Alfred",sales,2000},
 {emp,"041231","Doe","John",prod,2001},
 {emp,"052341","Smith","John",dev,1997},
 {emp,"076324","Smith","Ella",sales,1995},
```
Now, the amount of data in the table is of course too small to justify complicated ets searches, but on real tables, using \texttt{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 \texttt{ets:match} in such a situation:

\[
\texttt{l> ets:match(emp_tab, \{\'_', \'$1\', \'_', \'_', \text{sales}, \'_'\}).} \\
\texttt{[\"011103\"],[\"076324\"].}
\]

Even though \texttt{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 \texttt{ets:foldl} or \texttt{ets:foldr} instead:

\[
\texttt{ets:foldr(fun(#emp{empno = E, dept = sales},Acc) -> [E | Acc];} \\
\texttt{(_ ,Acc) -> Acc} \\
\texttt{end,} \\
\texttt{[],} \\
\texttt{emp_tab).}
\]

Running that would result in \{\"011103\",\"076324\"\}, which at least gets rid of the extra lists. The fun is also quite straightforward, so the only problem is that all the data from the table has to be transferred from the table to the calling process for filtering. That's inefficient compared to the \texttt{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 \texttt{ets:select} call that does what the \texttt{ets:foldr} does:

\[
\texttt{ets:select(emp_tab,\{#emp{empno = \'$1\', dept = sales, \_ = \'_\'},[[],[\'$1\']]}).}
\]

Even though the record syntax is used, it's still somewhat hard to read and even harder to write. The first element of the tuple, \#emp\{empno = \'$1\', dept = sales, \_ = \'_\'}\} tells what to match, elements not matching this will not be returned at all, as in the \texttt{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 \texttt{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 \texttt{ets:fun2ms}, one could have something that is as efficient as possible but still is written as a filter using the fun syntax:

\[
\texttt{-include_lib("stdlib/include/ms_transform.hrl").}
\]
This may not be the shortest of the expressions, but it requires no special knowledge of match specifications to read. The function's head should simply match what you want to filter out and the body returns what you want returned. As long as the function 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:foldl` example. In fact, it's even easier to read than the `ets:foldl` example, as the select call in itself discards anything that doesn't match, while the fun of the `foldl` 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:foldl` could do it (slowly, but correct):

```erlang
ext:foldl(fun(#emp{empno = E, empyear = Y},Acc) when Y < 2000 -> [E | Acc]; (_,Acc) -> Acc
dep_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:

```erlang
ets:select(emp_tab,[
  #{empno = '$1', empyear = '$2', _='_'},
  [{'<', '$2', 2000}],
  ['$1']])
```

This gives the same result, the `['<', '$2', 2000]` is in the guard part and therefore discards anything that does not have a `empyear` (bound to `$2` in the head) less than 2000, just as the guard in the `fold1` example. Let's jump on to writing it using `ets:fun2ms`:

```erlang
-include_lib("stdlib/include/ms_transform.hrl").
% ... 
ext:select(emp_tab, ets:fun2ms(
  fun(#emp{empno = E, empyear = Y}) when Y < 2000 -> E
  end)).
```

Obviously readability is gained by using the parse transformation.
I'll show some more examples without the tiresome comparing-to-alternatives stuff. Let's say we'd want the whole object matching instead of only one element. We could of course assign a variable to every part of the record and build it up once again in the body of the fun, but it's easier to do like this:

```erlang
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:

```erlang
ets:select(emp_tab, ets:fun2ms(
  fun(#emp{empno = [0 | Rest]}) ->
    [{0|Rest},[1|Rest}]
  end)).
```

As a matter of fact, this query hits the feature of partially bound keys in the table type ordered_set, so that not the whole table need be searched, only the part of the table containing keys beginning with 0 is in fact looked into.

The fun of course can have several clauses, so that if one could do the following: For each employee, if he or she is hired prior to 1997, return the tuple `{inventory, <employee number>}`, for each hired 1997 or later, but before 2001, return `{rookie, <employee number>}`, for all others return `{newbie, <employee number>}`, for 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>`:

```erlang
ets:select(emp_tab, ets:fun2ms(
  fun(#emp{empno = E, surname = "Smith"}) ->
    {guru,E};
    (#emp{empno = E, empyear = Y}) when Y < 1997 ->
      {inventory, E};
    (#emp{empno = E, empyear = Y}) when Y > 2001 ->
      {newbie, E};
    (#emp{empno = E, empyear = Y}) -> % 1997 -- 2001
      {rookie, E}
  end)).
```

The result will be:

```erlang
[{rookie,"011103"},
 {rookie,"041231"},
 {guru,"052341"},
 {guru,"076324"},
```
and so the Smith's will be happy...

So, what more can you do? Well, the simple answer would be; look in the documentation of match specifications in ERTS users guide. However let's briefly go through the most useful "built in functions" that you can use when the fun is to be translated into a match specification by ets:fun2ms (it's worth mentioning, although it might be obvious to some, that calling other functions than the one's allowed in match specifications cannot be done. No "usual" Erlang code can be executed by the fun being translated by fun2ms, the fun is after all limited exactly to the power of the match specifications, which is unfortunate, but the price one has to pay for the execution speed of an ets:select compared to ets:foldl/foldr).

The head of the fun is obviously a head matching (or mismatching) one parameter, one object of the table we select from. The object is always a single variable (can be _) or a tuple, as that's what's in ets, dets and mnesia tables (the match specification returned by ets:fun2ms can of course be used with dets:select and mnesia:select as well as with ets:select). The use of = in the head is allowed (and encouraged) on the top level.

The guard section can contain any guard expression of Erlang. Even the "old" type test are allowed on the toplevel of the guard (integer(X) instead of is_integer(X)). As the new type tests (the is_ tests) are in practice just guard bif's they can also be called from within the body of the fun, but so they can in ordinary Erlang code. Also arithmetics is allowed, as well as ordinary guard bif's. Here's a list of bif's and expressions:

- The type tests: is_atom, is_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, 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

```erlang
-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:

```erlang
(<0.86.0>) call ets:new(toy_table,[ordered_set])
```

Let's play we haven't spotted the problem yet, and want to see what ets:new returns. We do a slightly different trace pattern:

4> dbg:tp(ets,new,dbg:fun2ms(fun([toy_table,_]) -> return_trace() end)).
Resulting in the following trace output when we run the test:

```
(<0.86.0>) call ets:new(toy_table,[ordered_set])
(<0.86.0>) returned from ets:new/2 -> 24
```

The call to return_trace, makes a trace message appear when the function returns. It applies only to the specific function call triggering the match specification (and matching the head/guards of the match specification). This is the by far the most common call in the body of a dbg match specification.

As the test now fails with {badmatch,24}, it's obvious that the badmatch is because the atom toy_table does not match the number returned for an unnamed table. So we spotted the problem, the table should be named and the arguments supplied by our test program does not include named_table. We rewrite the start function to:

```
start(Args) ->
    toy_table = ets:new(toy_table,[named_table |Args]).
```

And with the same tracing turned on, we get the following trace output:

```
(<0.86.0>) call ets:new(toy_table,[named_table,ordered_set])
(<0.86.0>) returned from ets:new/2 -> toy_table
```

Very well. Let's say the module now passes all testing and goes into the system. After a while someone realizes that the table toy_table grows while the system is running and that for some reason there are a lot of elements with atom's as keys. You had expected only integer keys and so does the rest of the system. Well, obviously not all of the system. You turn on call tracing and try to see calls to your module with an atom as the key:

```
1> dbg:tracer().
{ok,<0.88.0>}
2> dbg:p(all,call).
{ok,[[matched,nonode@nohost,25]]}
3> dbg:tpl(toy,store,dbg:fun2ms(fun([A,_] when is_atom(A) -> true end)).
{ok,[[matched,nonode@nohost,1],[saved,1]]}
```

We use dbg:tpl here to make sure to catch local calls (let's say the module has grown since the smaller version and we're not sure this inserting of atoms is not done locally...). When in doubt always use local call tracing.

Let's say nothing happens when we trace in this way. Our function is never called with these parameters. We make the conclusion that someone else (some other module) is doing it and we realize that we must trace on ets:insert and want to see the calling function. The calling function may be retrieved using the match specification function caller and to get it into the trace message, one has to use the match spec function message. The filter call looks like this (looking for calls to ets:insert):

```
4> dbg:tpl(ets,insert,dbg:fun2ms(fun([[toy_table,{A,_}] when is_atom(A) ->
     message(caller())
     end])).
{ok,[[matched,nonode@nohost,1],[saved,2]]}
```

The caller will now appear in the "additional message" part of the trace output, and so after a while, the following output comes:
You have found out that the function evil_fun of the module evil_mod, with arity 2, is the one causing all this trouble.

This was just a toy example, but it illustrated the most used calls in match specifications for dbg. The other, more esoteric calls are listed and explained in the Users guide of the ERTS application, they really are beyond the scope of this document.

To end this chatty introduction with something more precise, here follows some parts about caveats and restrictions concerning the fun's used in conjunction with ets:fun2ms and dbg:fun2ms:

Warning:

To use the pseudo functions triggering the translation, one has to include the header file ms_transform.hrl in the source code. Failure to do so will possibly result in runtime errors rather than compile time, as the expression may be valid as a plain Erlang program without translation.

Warning:

The fun has to be literally constructed inside the parameter list to the pseudo functions. The fun cannot be bound to a variable first and then passed to ets:fun2ms or dbg:fun2ms, i.e this will work: ets:fun2ms (fun(A) -> A end) but not this: F = fun(A) -> A end, ets:fun2ms(F). The later will result in a compile time error if the header is included, otherwise a runtime error. Even if the later construction would ever appear to work, it really doesn't, so don't ever use it.

Several restrictions apply to the fun that is being translated into a match_spec. To put it simple you cannot use anything in the fun that you cannot use in a match_spec. This means that, among others, the following restrictions apply to the fun itself:

• Functions written in Erlang cannot be called, neither local functions, global functions or real fun's
• Everything that is written as a function call will be translated into a match_spec call to a builtin function, so that the call is_list(X) will be translated to {'is_list', '$1'} ('$1' is just an example, the numbering may vary). If one tries to call a function that is not a match_spec builtin, it will cause an error.
• Variables occurring in the head of the fun will be replaced by match_spec variables in the order of occurrence, so that the fragment fun({A,B,C}) will be translated to {'$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 [{is_atom, '$1'}],[{'$2'}]).
• Variables that are not appearing in the head are imported from the environment and made into match_spec const expressions. Example from the shell:

1> X = 25.
25
2> ets:fun2ms(fun({A,B}) when A > X -> B end).
[{{'$1','$2'}},{{'>','$1'},{'const',25}},{'$2'}]
• Matching with = cannot be used in the body. It can only be used on the top level in the head of the fun. Example from the shell again:

```
1> ets:fun2ms(fun({A,[B|C]} = D) when A > B -> D end).
\[\{\{\$1\},[\{\$2\}|\{\$3\}]\},[\{\$\}],[\{\$\}]]
2> ets:fun2ms(fun({A,[B|C]=D}) when A > B -> D end).
Error: fun with head matching ('=' in head) cannot be translated into match_spec
{error,transform_error}
3> ets:fun2ms(fun({A,[B|C]}) when A > B -> D = [B|C], D end).
Error: fun with body matching ('=' in body) is illegal as match_spec
{error,transform_error}
```

All variables are bound in the head of a match_spec, so the translator can not allow multiple bindings. The special case when matching is done on the top level makes the variable bind to '$_' in the resulting match_spec, it is to allow a more natural access to the whole matched object. The pseudo function object() could be used instead, see below. The following expressions are translated equally:

```
ets:fun2ms(fun({a,\_} = A) -> A end).
ets:fun2ms(fun({a,\_}) -> object() end).
```

• The special match_spec variables '$_' and '$*' can be accessed through the pseudo functions object() (for '$_') and bindings() (for '$*'). as an example, one could translate the following ets:match_object/2 call to a ets:select call:

```
ets:match_object(Table, {'$1',test,'$2'}).  
```

...is the same as...

```
ets:select(Table, ets:fun2ms(fun({A,test,B}) -> object() end)).
```

(This was just an example, in this simple case the former expression is probably preferable in terms of readability). The ets:select/2 call will conceptually look like this in the resulting code:

```
ets:select(Table, [[\{\$1\},test,\$2\}],[],[\$_])
```

Matching on the top level of the fun head might feel like a more natural way to access '$_', see above.

• Term constructions/literals are translated as much as is needed to get them into valid match_specs, so that tuples are made into match_spec tuple constructions (a one element tuple containing the tuple) and constant expressions are used when importing variables from the environment. Records are also translated into plain tuple constructions, calls to element etc. The guard test is_record/2 is translated into match_spec code using the three parameter version that's built into match Specs, so that is_record(A,t) is translated into {is_record, '$_',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 \texttt{ets:fun2ms/1}, the fun’s head must contain a single variable or a single tuple. If the pseudo function is \texttt{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 \textit{ERTS users guide}.

Exports

\texttt{parse_transform(Forms, Options) -> Forms}

Types:
\begin{itemize}
  \item Forms = [\texttt{erl_parse:abstract_form()}]
  \item Options = term()
\end{itemize}

Implements the actual transformation at compile time. This function is called by the compiler to do the source code transformation if and when the \texttt{ms_transform.hrl} header file is included in your source code. See the \texttt{ets} and \texttt{dbg:fun2ms/1} function manual pages for documentation on how to use this \texttt{parse_transform}, see the match_spec chapter in \textit{ERTS users guide} for a description of match specifications.

\texttt{transform_from_shell(Dialect, Clauses, BoundEnvironment) -> term()}

Types:
\begin{itemize}
  \item Dialect = ets | dbg
  \item Clauses = [\texttt{erl_parse:abstract_clause()}]
  \item BoundEnvironment = \texttt{erl_eval:binding_struct()}
\end{itemize}

Implements the actual transformation when the \texttt{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 \texttt{BoundEnvironment}. The result is a term, normalized, i.e. not in abstract format.

\texttt{format_error(Error) -> Chars}

Types:
\begin{itemize}
  \item Error = \{error, module(), term()\}
  \item Chars = \texttt{io_lib:chars()}
\end{itemize}

Takes an error code returned by one of the other functions in the module and creates a textual description of the error. Fairly uninteresting function actually.
Orddict implements a Key-Value dictionary. An orddict is a representation of a dictionary, where a list of pairs is used to store the keys and values. The list is ordered after the keys.

This module provides exactly the same interface as the module dict but with a defined representation. 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
orddict() = [{Key :: term(), Value :: term()}]
As returned by new/0.

Exports
append(Key, Value, Orddict1) -> Orddict2
Types:
  Key = Value = term()
  Orddict1 = Orddict2 = orddict()
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:
  Key = term()
  ValList = [Value :: term()]
  Orddict1 = Orddict2 = orddict()
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 = orddict()
This function erases all items with a given key from a dictionary.

fetch(Key, Orddict) -> Value
Types:
  Key = Value = term()
  Orddict = orddict()
This function returns the value associated with Key in the dictionary Orddict. fetch assumes that the Key is present in the dictionary and an exception is generated if Key is not in the dictionary.
fetch_keys(Orddict) -> Keys
Types:
    Orddict = orddict()
    Keys = [term()]
This function returns a list of all keys in the dictionary.

filter(Pred, Orddict1) -> Orddict2
Types:
    Pred = fun((Key :: term(), Value :: term()) -> boolean())
    Orddict1 = Orddict2 = orddict()
Orddict2 is a dictionary of all keys and values in Orddict1 for which Pred(Key, Value) is true.

find(Key, Orddict) -> {ok, Value} | error
Types:
    Key = term()
    Orddict = orddict()
    Value = term()
This function searches for a key in a dictionary. Returns {ok, Value} where Value is the value associated with Key, or error if the key is not present in the dictionary.

fold(Fun, Acc0, Orddict) -> Acc1
Types:
    Fun = fun((Key :: term(), Value :: term(), AccIn :: term()) ->
                 AccOut :: term())
    Acc0 = Acc1 = term()
    Orddict = orddict()
Calls Fun on successive keys and values of Orddict together with an extra argument Acc (short for accumulator). Fun must return a new accumulator which is passed to the next call. Acc0 is returned if the list is empty. The evaluation order is undefined.

from_list(List) -> Orddict
Types:
    List = [{Key :: term(), Value :: term()}]
    Orddict = orddict()
This function converts the Key - Value list List to a dictionary.

is_key(Key, Orddict) -> boolean()
Types:
    Key = term()
    Orddict = orddict()
This function tests if Key is contained in the dictionary Orddict.
map(Fun, Orddict1) -> Orddict2
Types:
   Fun = fun((Key :: term(), Value1 :: term()) -> Value2 :: term())
   Orddict1 = Orddict2 = orddict()

map calls Fun on successive keys and values of Orddict1 to return a new value for each key. The evaluation order is undefined.

merge(Fun, Orddict1, Orddict2) -> Orddict3
Types:
   Fun = fun((Key :: term(), Value1 :: term(), Value2 :: term()) -> Value :: term())
   Orddict1 = Orddict2 = Orddict3 = orddict()

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:

```erlang
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() -> orddict()
This function creates a new dictionary.

size(Orddict) -> integer() >= 0
Types:
   Orddict = orddict()
Returns the number of elements in an Orddict.

is_empty(Orddict) -> boolean()
Types:
   Orddict = orddict()
Returns true if Orddict has no elements, false otherwise.

store(Key, Value, Orddict1) -> Orddict2
Types:
   Key = Value = term()
   Orddict1 = Orddict2 = orddict()
This function stores a Key-Value pair in a dictionary. If the Key already exists in Orddict1, the associated value is replaced by Value.
to_list(Orddict) -> List
Types:
    Orddict = ordict()
    List = [{Key :: term(), Value :: term()}]
This function converts the dictionary to a list representation.

update(Key, Fun, Orddict1) -> Orddict2
Types:
    Key = term()
    Fun = fun((Value1 :: term()) -> Value2 :: term())
    Orddict1 = Orddict2 = ordict()
Update 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 :: term()) -> Value2 :: term())
    Orddict1 = Orddict2 = ordict()
Update 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 = ordict()
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:
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), gb_trees(3)
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 (\(\neq\)), this module considers two elements as different if and only if they do not compare equal (\(==\)).

Data Types

ordset(T) = [T]
As returned by new/0.

Exports

new() -> []
Returns a new empty ordered set.

is_set(Ordset) -> boolean()
Types:
   Ordset = term()
Returns true if Ordset is an ordered set of elements, otherwise false.

size(Ordset) -> integer() >= 0
Types:
   Ordset = ordset(term())
Returns the number of elements in Ordset.

to_list(Ordset) -> List
Types:
   Ordset = ordset(T)
   List = [T]
Returns the elements of Ordset as a list.

from_list(List) -> Ordset
Types:
   List = [T]
   Ordset = ordset(T)
Returns an ordered set of the elements in List.

is_element(Element, Ordset) -> boolean()
Types:
ordsets

Element = term()
Ordset = ordset(term())
Returns true if Element is an element of Ordset, otherwise false.

add_element(Element, Ordset1) -> Ordset2
Types:
   Element = E
   Ordset1 = ordset(T)
   Ordset2 = ordset(T | E)
Returns a new ordered set formed from Ordset1 with Element inserted.

del_element(Element, Ordset1) -> Ordset2
Types:
   Element = term()
   Ordset1 = Ordset2 = ordset(T)
Returns Ordset1, but with Element removed.

union(Ordset1, Ordset2) -> Ordset3
Types:
   Ordset1 = ordset(T1)
   Ordset2 = ordset(T2)
   Ordset3 = ordset(T1 | T2)
Returns the merged (union) set of Ordset1 and Ordset2.

union(OrdsetList) -> Ordset
Types:
   OrdsetList = [ordset(T)]
   Ordset = ordset(T)
Returns the merged (union) set of the list of sets.

intersection(Ordset1, Ordset2) -> Ordset3
Types:
   Ordset1 = Ordset2 = Ordset3 = ordset(term())
Returns the intersection of Ordset1 and Ordset2.

intersection(OrdsetList) -> Ordset
Types:
   OrdsetList = [ordset(term()), ...]
   Ordset = ordset(term())
Returns the intersection of the non-empty list of sets.
is_disjoint(Ordset1, Ordset2) -> boolean()
Types:
  Ordset1 = Ordset2 = ordset(term())
Returns true if Ordset1 and Ordset2 are disjoint (have no elements in common), and false otherwise.

subtract(Ordset1, Ordset2) -> Ordset3
Types:
  Ordset1 = Ordset2 = Ordset3 = ordset(term())
Returns only the elements of Ordset1 which are not also elements of Ordset2.

is_subset(Ordset1, Ordset2) -> boolean()
Types:
  Ordset1 = Ordset2 = ordset(term())
Returns true when every element of Ordset1 is also a member of Ordset2, otherwise false.

fold(Function, Acc0, Ordset) -> Acc1
Types:
  Function = fun((Element :: T, AccIn :: term()) -> AccOut :: term())
  Ordset = ordset(T)
  Acc0 = Acc1 = term()
Fold Function over every element in Ordset returning the final value of the accumulator.

filter(Pred, Ordset1) -> Ordset2
Types:
  Pred = fun((Element :: T) -> boolean())
  Ordset1 = Ordset2 = ordset(T)
Filter elements in Ordset1 with boolean function Pred.

See Also
  gb_sets(3), sets(3)
Warning:
This module is deprecated and will be removed in Erlang/OTP 18.

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 \( P_1 \) sends the message \( M_1 \) to the group, and process \( P_2 \) simultaneously sends message \( M_2 \), 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.

```erlang
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`.

```erlang
join(PgName, Pid) -> Members
```

Types:

```
PgName = term()
Pid = pid()
Members = [pid()]
```

Joins the pid `Pid` to the process group `PgName`. Returns a list of all old members of the group.

```erlang
send(PgName, Msg) -> ok
```

Types:
PgName = Msg = term()

Sends the tuple \{pg_message, From, PgName, Msg\} to all members of the process group PgName.

Failure: \{badarg, \{PgName, Msg\}\} if PgName is not a process group (a globally registered name).

esend(PgName, Msg) -> ok

Types:

  PgName = Msg = term()

Sends the tuple \{pg_message, From, PgName, Msg\} to all members of the process group PgName, except ourselves.

Failure: \{badarg, \{PgName, Msg\}\} if PgName is not a process group (a globally registered name).

members(PgName) -> Members

Types:

  PgName = term()

  Members = [pid()]

Returns a list of all members of the process group PgName.
pool

Erlang module

pool can be used to run a set of Erlang nodes as a pool of computational processors. It is organized as a master and a set of slave nodes and includes the following features:

- The slave nodes send regular reports to the master about their current load.
- Queries can be sent to the master to determine which node will have the least load.

The BIF `statistics(run_queue)` is used for estimating future loads. It returns the length of the queue of ready to run processes in the Erlang runtime system.

The slave nodes are started with the `slave` module. This effects, tty IO, file IO, and code loading.

If the master node fails, the entire pool will exit.

Exports

```
start(Name) -> Nodes
start(Name, Args) -> Nodes
Types:
  Name = atom()
  Args = string()
  Nodes = [node()]
Starts a new pool. The file `.hosts.erlang` is read to find host names where the pool nodes can be started. See section Files 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)`.

Access rights must be set so that all nodes in the pool have the authority to access each other.

The function is synchronous and all the nodes, as well as all the system servers, are running when it returns a value.
```

```
attach(Node) -> already_attached | attached
Types:
  Node = node()
This function ensures that a pool master is running and includes Node in the pool master's pool of nodes.
```

```
stop() -> stopped
Stops the pool and kills all the slave nodes.
```

```
get_nodes() -> [node()]
Returns a list of the current member nodes of the pool.
```

```
pspawn(Mod, Fun, Args) -> pid()
Types:
```

Mod = module()
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 = module()
    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.

**Data Types**

```
spawn_option() = link |
| monitor |
| {priority, priority_level()} |
| {min_heap_size, integer() >= 0} |
| {min_bin_vheap_size, integer() >= 0} |
| {fullsweep_after, integer() >= 0} |
```

See `erlang:spawn_opt/2,3,4,5`.

```
priority_level() = high | low | max | normal
```

```
dict_or_pid() = pid() |
| (ProcInfo :: [term()]) |
| {X :: integer(), |
| Y :: integer(), |
| Z :: integer()}
```

**Exports**

```
spawn(Fun) -> pid() |
spawn(Node, Fun) -> pid() |
spawn(Module, Function, Args) -> pid() |
spawn(Node, Module, Function, Args) -> pid() |
```

Types:
Node = node()
Fun = function()
Module = module()
Function = atom()
Args = [term()]

Spawns a new process and initializes it as described above. The process is spawned using the `spawn` BIFs.

\[
\text{spawn} \text{_{link}} \text{(Fun)} -> \text{pid}()
\text{spawn} \text{_{link}} \text{(Node, Fun)} -> \text{pid}()
\text{spawn} \text{_{link}} \text{(Module, Function, Args)} -> \text{pid}()
\text{spawn} \text{_{link}} \text{(Node, Module, Function, Args)} -> \text{pid}()
\]

Types:
Node = node()
Fun = function()
Module = module()
Function = atom()
Args = [term()]

Spawns a new process and initializes it as described above. The process is spawned using the `spawn_link` BIFs.

\[
\text{spawn} \text{_{opt}} \text{(Fun, SpawnOpts)} -> \text{pid}()
\text{spawn} \text{_{opt}} \text{(Node, Function, SpawnOpts)} -> \text{pid}()
\text{spawn} \text{_{opt}} \text{(Module, Function, Args, SpawnOpts)} -> \text{pid}()
\text{spawn} \text{_{opt}} \text{(Node, Module, Function, Args, SpawnOpts)} -> \text{pid}()
\]

Types:
Node = node()
Fun = function()
Module = module()
Function = atom()
Args = [term()]
SpawnOpts = [\text{spawn\_option()}]

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`.  

Ericsson AB. All Rights Reserved.: STDLIB | 295
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 = module()
Function = atom()
Args = [term()]
Time = timeout()
SpawnOpts = [spawn_option()]
Ret = term() | {error, Reason :: term()}

Starts a new process synchronously. Spawns the process and waits for it to start. When the process has started, it must call init_ack(Parent,Ret) or init_ack(Ret), where Parent is the process that evaluates this function. At this time, Ret is returned.

If the start_link/3,4,5 function is used and the process crashes before it has called init_ack/1,2, {error, Reason} is returned if the calling process traps exits.

If Time is specified as an integer, this function waits for Time milliseconds for the new process to call init_ack, or {error, timeout} is returned, and the process is killed.

The SpawnOpts argument, if given, will be passed as the last argument to the spawn_opt/2,3,4,5 BIF.

Note:
Using the spawn option monitor is currently not allowed, but will cause the function to fail with reason badarg.

init_ack(Ret) -> ok
init_ack(Parent, Ret) -> ok
Types:
Parent = pid()
Ret = term()

This function must used by a process that has been started by a start_link/3,4,5 function. It tells Parent that the process has initialized itself, has started, or has failed to initialize itself.

The init_ack/1 function uses the parent value previously stored by the start function used.

If this function is not called, the start function will return an error tuple (if a link and/or a timeout is used) or hang otherwise.

The following example illustrates how this function and proc_lib:start_link/3 are used.

-module(my_proc).
-export([start_link/0]).
-export([init/1]).
**start_link()**

```
start_link() ->
    proc_lib:start_link(my_proc, init, [self()]).
```

**init(Parent)**

```
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()]`

Equivalent to `format(CrashReport, latin1)`.

**format(CrashReport, Encoding) -> string()**

Types:

- `CrashReport = [term()]`
- `Encoding = latin1 | unicode | utf8`

This function can be used by a user defined event handler to format a crash report. The crash report is sent using `error_logger:error_report(crash_report, CrashReport)`. That is, the event to be handled is of the format `{error_report, GL, {Pid, crash_report, CrashReport}}` where `GL` is the group leader pid of the process `Pid` which sent the crash report.

**initial_call(Process) -> {Module, Function, Args} | false**

Types:

- `Process = dict_or_pid()`
- `Module = 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}
Types:
   Process = dict_or_pid()
   Module = module()
   Function = atom()
   Arity = byte()
```

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) -> no_return()
Types:
   Module = 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)
Property lists are ordinary lists containing entries in the form of either tuples, whose first elements are keys used for lookup and insertion, or atoms, which work as shorthand for tuples \{Atom, true\}. (Other terms are allowed in the lists, but are ignored by this module.) If there is more than one entry in a list for a certain key, the first occurrence normally overrides any later (irrespective of the arity of the tuples).

Property lists are useful for representing inherited properties, such as options passed to a function where a user may specify options overriding the default settings, object properties, annotations, etc.

Two keys are considered equal if they match (=:=). In other words, numbers are compared literally rather than by value, so that, for instance, 1 and 1.0 are different keys.

**Data Types**

\[
\text{property}() = \text{atom()} | \text{tuple()}
\]

**Exports**

\[
\text{append_values}(\text{Key}, \text{ListIn}) \rightarrow \text{ListOut}
\]

Types:

\[
\begin{align*}
\text{Key} &= \text{term()} \\
\text{ListIn} &= \text{ListOut} = [\text{term()}]
\end{align*}
\]

Similar to \text{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., \text{append_values}(a, \{(a, [1,2]), \{b, 0\}, \{a, 3\}, \{c, -1\}, \{a, [4]\}\}) will return the list [1,2,3,4].

\[
\text{compact}([\text{property()}]) \rightarrow \text{ListOut}
\]

Types:

\[
\begin{align*}
\text{ListIn} &= \text{ListOut} = [\text{property()}]
\end{align*}
\]

Minimizes the representation of all entries in the list. This is equivalent to \[\text{property}(P) | P \leftarrow \text{ListIn}\]. See also: \text{property}/1, \text{unfold}/1.

\[
\text{delete}(\text{Key}, \text{List}) \rightarrow \text{List}
\]

Types:

\[
\begin{align*}
\text{Key} &= \text{term()} \\
\text{List} &= [\text{term()}]
\end{align*}
\]

Deletes all entries associated with Key from List.

\[
\text{expand}(\text{Expansions}, \text{ListIn}) \rightarrow \text{ListOut}
\]

Types:
Expansions = [{Property :: property(), Expansion :: [term()]}]
ListIn = ListOut = [term()]

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 ListIn 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 ListIn.

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) -> boolean()
Types:
  Key = term()
  List = [term()]

Returns the value of a boolean key/value option. If lookup(Key, List) would yield {Key, true}, this function returns true; otherwise false is returned.
See also: get_value/2, lookup/2.

get_keys(List) -> [term()]
Types:
List = [term()]
Returns an unordered list of the keys used in List, not containing duplicates.

get_value(Key, List) -> term()
Types:
  Key = term()
  List = [term()]
Equivalent to get_value(Key, List, undefined).

get_value(Key, List, Default) -> term()
Types:
  Key = term()
  List = [term()]
  Default = term()
Returns the value of a simple key/value property in List. If lookup(Key, List) would yield {Key, Value},
this function returns the corresponding Value, otherwise Default is returned.
See also: get_all_values/2, get_bool/2, get_value/2, lookup/2.

is_defined(Key, List) -> boolean()
Types:
  Key = term()
  List = [term()]
Returns true if List contains at least one entry associated with Key, otherwise false is returned.

lookup(Key, List) -> none | tuple()
Types:
  Key = term()
  List = [term()]
Returns the first entry associated with Key in List, if one exists, otherwise returns none. For an atom A in the list,
the tuple {A, true} is the entry associated with A.
See also: get_bool/2, get_value/2, lookup_all/2.

lookup_all(Key, List) -> [tuple()]
Types:
  Key = term()
  List = [term()]
Returns the list of all entries associated with Key in List. If no such entry exists, the result is the empty list.
See also: lookup/2.

normalize(ListIn, Stages) -> ListOut
Types:
ListIn = [term()]
Stages = [Operation]
Operation = \{aliases, Aliases\}
    | \{negations, Negations\}
    | \{expand, Expansions\}
Aliases = Negations = \{{Key, Key}\}
Expansions = \{{Property :: property(), Expansion :: [term()]}\}
ListOut = [term()]

Passes ListIn through a sequence of substitution/expansion stages. For an aliases operation, the function substitute_aliases/2 is applied using the given list of aliases; for a negations operation, substitute_negations/2 is applied using the given negation list; for an expand operation, the function expand/2 is applied using the given list of expansions. The final result is automatically compacted (cf. compact/1).

Typically you want to substitute negations first, then aliases, then perform one or more expansions (sometimes you want to pre-expand particular entries before doing the main expansion). You might want to substitute negations and/or aliases repeatedly, to allow such forms in the right-hand side of aliases and expansion lists.

See also: compact/1, expand/2, substitute_aliases/2, substitute_negations/2.

property(PropertyIn) -> PropertyOut
Types:
    PropertyIn = PropertyOut = property()

Creates a normal form (minimal) representation of a property. If PropertyIn is \{Key, true\} where Key is an atom, this returns Key, otherwise the whole term PropertyIn is returned.

See also: property/2.

property(Key, Value) -> Property
Types:
    Key = Value = term()
    Property = atom() | \{term(), term()\}

Creates a normal form (minimal) representation of a simple key/value property. Returns Key if Value is true and Key is an atom, otherwise a tuple \{Key, Value\} is returned.

See also: property/1.

split(List, Keys) -> \{Lists, Rest\}
Types:
    List = Keys = [term()]
    Lists = [[term()]]
    Rest = [term()]

Partitions List into a list of sublists and a remainder. Lists contains one sublist for each key in Keys, in the corresponding order. The relative order of the elements in each sublist is preserved from the original List. Rest contains the elements in List that are not associated with any of the given keys, also with their original relative order preserved.

Example: split([\{c, 2\}, \{c, 1\}, a, \{c, 3, 4\}, d, \{b, 5\}, b, \{a, b, c\}])
returns
substitute_aliases(Aliases, ListIn) -> ListOut
Types:
  Aliases = [{Key, Key}]
  Key = term()
  ListIn = ListOut = [term()]
Substitutes keys of properties. For each entry in ListIn, if it is associated with some key K1 such that {K1, K2} occurs in Aliases, the key of the entry is changed to K2. If the same K1 occurs more than once in Aliases, only the first occurrence is used.
Example: substitute_aliases([[color, colour]], L) will replace all tuples {color, ...} in L with {colour, ...}, and all atoms color with colour.
See also: normalize/2, substitute_negations/2.

substitute_negations(Negations, ListIn) -> ListOut
Types:
  Negations = [{Key, Key}]
  Key = term()
  ListIn = ListOut = [term()]
Substitutes keys of boolean-valued properties and simultaneously negates their values. For each entry in ListIn, if it is associated with some key K1 such that {K1, K2} occurs in Negations, then if the entry was {K1, true} it will be replaced with {K2, false}, otherwise it will be replaced with {K2, true}, thus changing the name of the option and simultaneously negating the value given by get_bool(ListIn). If the same K1 occurs more than once in Negations, only the first occurrence is used.
Example: substitute_negations([[no_foo, foo]], L) will replace any atom no_foo or tuple {no_foo, true} in L with {foo, false}, and any other tuple {no_foo, ...} with {foo, true}.
See also: get_bool/2, normalize/2, substitute_aliases/2.

unfold(ListIn) -> ListOut
Types:
  ListIn = ListOut = [term()]
Unfolds all occurrences of atoms in ListIn to tuples {Atom, true}.
The qlc module provides a query interface to Mnesia, ETS, Dets and other data structures that implement an iterator style traversal of objects.

Overview

The qlc module implements a query interface to QLC tables. Typical QLC tables are ETS, Dets, and Mnesia tables. There is also support for user defined tables, see the Implementing a QLC table section. A query is stated using Query List Comprehensions (QLCs). The answers to a query are determined by data in QLC tables that fulfill the constraints expressed by the QLCs of the query. QLCs are similar to ordinary list comprehensions as described in the Erlang Reference Manual and Programming Examples except that variables introduced in patterns cannot be used in list expressions. In fact, in the absence of optimizations and options such as cache and unique (see below), every QLC free of QLC tables evaluates to the same list of answers as the identical ordinary list comprehension.

While ordinary list comprehensions evaluate to lists, calling qlc:q/1,2 returns a Query Handle. To obtain all the answers to a query, qlc:eval/1,2 should be called with the query handle as first argument. Query handles are essentially functional objects (“funs”) created in the module calling q/1,2. As the funs refer to the module's code, one should be careful not to keep query handles too long if the module's code is to be replaced. Code replacement is described in the Erlang Reference Manual. The list of answers can also be traversed in chunks by use of a Query Cursor. Query cursors are created by calling qlc:cursor/1,2 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:

\[
[\text{Expression} \mid \text{Qualifier}_1, \text{Qualifier}_2, \ldots]
\]

Expression (the template) is an arbitrary Erlang expression. Qualifiers are either filters or generators. Filters are Erlang expressions returning bool(). Generators have the form \text{Pattern} \leftarrow \text{ListExpression}, where ListExpression is an expression evaluating to a query handle or a list. Query handles are returned from qlc:table/2, qlc:append/1,2, qlc:sort/1,2, qlc:keysort/2,3, qlc:q/1,2, and qlc:string_to_handle/1,2,3.

Evaluation

The evaluation of a query handle begins by the inspection of options and the collection of information about tables. As a result qualifiers are modified during the optimization phase. Next all list expressions are evaluated. If a cursor has been created evaluation takes place in the cursor process. For those list expressions that are QLCs, the list expressions of the QLCs’ generators are evaluated as well. One has to be careful if list expressions have side effects since the order in which list expressions are evaluated is unspecified. Finally the answers are found by evaluating the qualifiers from left to right, backtracking when some filter returns false, or collecting the template when all filters return true.

Filters that do not return bool() but fail are handled differently depending on their syntax: if the filter is a guard it returns false, otherwise the query evaluation fails. This behavior makes it possible for the qlc module to do some optimizations without affecting the meaning of a query. For example, when 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 $P_1$ of one query handler and some position $P_2$ 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 $P_1$ and $P_2$ 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 $P_1$ and $P_2$ do not compare equal. If there are many objects with the same value of $P_2$ a temporary file will be used for the equivalence classes.

The qlc module warns at compile time if a QLC combines query handles in such a way that more than one join is possible. In other words, there is no query planner that can choose a good order between possible join operations. It is up to the user to order the joins by introducing query handles.

The join is to be expressed as a guard filter. The filter must be placed immediately after the two joined generators, possibly after guard filters that use variables from no other generators but the two joined generators. The qlc module inspects the operands of $=:=/2$, $==/2$, is_record/2, element/2, and logical operators (and/2, or/2, andalso/2, orelse/2, xor/2) when determining which joins to consider.

**Common options**

The following options are accepted by cursor/2, eval/2, fold/4, and info/2:

- `{cache_all, Cache}` where `Cache` is equal to ets or list adds a `{cache, Cache}` option to every list expression of the query except tables and lists. Default is `{cache_all, no}`. The option `cache_all` is equivalent to `{cache_all, ets}`.
- `{max_list_size, MaxListSize}` where `MaxListSize` is the size in bytes of terms on the external format. If the accumulated size of collected objects exceeds `MaxListSize` the objects are written onto a temporary file. This option is used by the `{cache, list}` option as well as by the merge join method. Default is 512*1024 bytes.
- `{tmpdir_usage, TmpFileUsage}` determines the action taken when qlc is about to create temporary files on the directory set by the `tmpdir` option. If the value is `not_allowed` an error tuple is returned, otherwise temporary files are created as needed. Default is `allowed` which means that no further action is taken. The values `info_msg`, `warning_msg`, and `error_msg` mean that the function with the corresponding name in the module `error_logger` is called for printing some information (currently the stacktrace).
- `{tmpdir, TempDirectory}` sets the directory used by merge join for temporary files and by the `{cache, list}` option. The option also overrides the `tmpdir` option of `keysort/3` and `sort/2`. The default value is "" which means that the directory returned by `file:get_cwd()` is used.
- `{unique_all, true}` adds a `unique, true` option to every list expression of the query. Default is `{unique_all, false}`. The option `unique_all` is equivalent to `{unique_all, true}`.

**Getting started**

As already mentioned queries are stated in the list comprehension syntax as described in the *Erlang Reference Manual*. In the following some familiarity with list comprehensions is assumed. There are examples in *Programming Examples* that can get you started. It should be stressed that list comprehensions do not add any computational power to the language; anything that can be done with list comprehensions can also be done without them. But they add a syntax for expressing simple search problems which is compact and clear once you get used to it.
Many list comprehension expressions can be evaluated by the qlc module. Exceptions are expressions such that variables introduced in patterns (or filters) are used in some generator later in the list comprehension. As an example consider an implementation of lists:append(L): [X || Y <- L, X <- Y]. Y is introduced in the first generator and used in the second. The ordinary list comprehension is normally to be preferred when there is a choice as to which to use. One difference is that qlc:eval/1,2 collects answers in a list which is finally reversed, while list comprehensions collect answers on the stack which is finally unwound.

What the qlc module primarily adds to list comprehensions is that data can be read from QLC tables in small chunks. A QLC table is created by calling qlc:table/2. Usually qlc:table/2 is not called directly from the query but via an interface function of some data structure. There are a few examples of such functions in Erlang/OTP: mnesia:table/1,2, ets:table/1,2, and dets:table/1,2. For a given data structure there can be several functions that create QLC tables, but common for all these functions is that they return a query handle created by qlc:table/2. Using the QLC tables provided by OTP is probably sufficient in most cases, but for the more advanced user the section Implementing a QLC table describes the implementation of a function calling qlc:table/2.

Besides qlc:table/2 there are other functions that return query handles. They might not be used as often as tables, but are useful from time to time. qlc:append traverses objects from several tables or lists after each other. If, for instance, you want to traverse all answers to a query QH and then finish off by a term {finished}, you can do that by calling qlc:append(QH, [{finished}]). append first returns all objects of QH, then {finished}. If there is one tuple {finished} among the answers to QH it will be returned twice from append.

As another example, consider concatenating the answers to two queries QH1 and QH2 while removing all duplicates. The means to accomplish this is to use the unique option:

```
qlc:q([X || X <- qlc:append(QH1, QH2)], {unique, true})
```

The cost is substantial: every returned answer will be stored in an ETS table. Before returning an answer it is looked up in the ETS table to check if it has already been returned. Without the unique options all answers to QH1 would be returned followed by all answers to QH2. The unique options keeps the order between the remaining answers.

If the order of the answers is not important there is the alternative to sort the answers uniquely:

```
qlc:sort(qlc:q([X || X <- qlc:append(QH1, QH2)], {unique, true})).
```

This query also removes duplicates but the answers will be sorted. If there are many answers temporary files will be used. Note that in order to get the first unique answer all answers have to be found and sorted. 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:

```
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:
The qlc module will evaluate this differently depending on the query handles QH1 and QH2. If, for example, X2 is matched against the key of a QLC table the lookup join method will traverse the objects of QH2 while looking up key values in the table. On the other hand, if neither X2 nor Y2 is matched against the key or an indexed position of a QLC table, the merge join method will make sure that QH1 and QH2 are both sorted on position 2 and next do the join by traversing the objects one by one.

The join option can be used to force the qlc module to use a certain join method. For the rest of this section it is assumed that the excessively slow join method called "nested loop" has been chosen:

```
qlc:q([X1,X2,X3,Y1] ||
       [X1,X2,X3] <- QH1,
       [Y1,Y2]   <- QH2,
       X2 =:= Y2)
```

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* section, `gb_table:table/1`, the iterator for the gb-tree will be initiated for each answer to QH1 after which the objects of the gb-tree will be returned one by one. This is probably the most efficient way of traversing the table in that case since it takes minimal computational power to get the following object. But if QH2 is not a table but a more complicated QLC, it can be more efficient use some RAM memory for collecting the answers in a cache, particularly if there are only a few answers. It must then be assumed that evaluating QH2 has no side effects so that the meaning of the query does not change if QH2 is evaluated only once. One way of caching the answers is to evaluate QH2 first of all and substitute the list of answers for QH2 in the query. Another way is to use the cache option. It is stated like this:

```
QH2' = qlc:q([X | X <- QH2], {cache, ets})
```

or just

```
QH2' = qlc:q([X | X <- QH2], cache)
```

The effect of the cache option is that when the generator QH2' is run the first time every answer is stored in an ETS table. When next answer of QH1 is tried, answers to QH2' are copied from the ETS table which is very fast. As for the unique option the cost is a possibly substantial amount of RAM memory. The (cache, list) option offers the possibility to store the answers in a list on the process heap. While this has the potential of being faster than ETS tables since there is no need to copy answers from the table it can often result in slower evaluation due to more garbage collections of the process’ heap as well as increased RAM memory consumption due to larger heaps. Another drawback with cache lists is that if the size of the list exceeds a limit a temporary file will be used. Reading the answers from a file is very much slower than copying them from an ETS table. But if the available RAM memory is scarce setting the limit 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` function the implementation of a QLC table for the `gb_trees` 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}];
                none -> []
            end
        end, Ks)
    end,
    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,'=='}).

gb_nodes(T, infinity, ElementFun) ->
    gb_nodes(T, -1, ElementFun);

gb_nodes(T, NElements, ElementFun) ->
    gb_iter(gb_trees:iterator(T), NElements, ElementFun).

gb_iter(I, 0, _EFun) ->
    '...';

gb_iter(I0, N, EFun) ->
    case gb_trees:next(I0) of
        {X, V, I} ->
            EFun(X, V) | gb_iter(I, N-1, EFun));
```

308 | Ericsson AB. All Rights Reserved.: STDLIB
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

\[
\text{qlc:q}([T \mid P \leftarrow \text{LE}, \text{F}])
\]

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:

```erlang
1> E1 = ets:new(t, [set]), % uses =:=/2 for key equality
2> Q1 = qlc:q([{K} || {K} <- ets:table(E1),
3> K == 2.71 orelse K == a]),
4> io:format("-s-n", [qlc:info(Q1)]).
5> ets:match_spec_run(lists:flatmap(fun(V) ->
6> ets:lookup(20493, V)
7> end, [a,2.71]),
8> ets:match_spec_compile([{{'$1'},[],['$1']}]))
```
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), 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:

```erlang
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,[[{'2','2'},[{'==','2'},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:

```erlang
4> E3 = ets:new(t, [ordered_set]), % uses ==/2 for key equality
   true = 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.

**Data Types**

- `abstract_expr() = erl_parse:abstract_expr()`
  Parse trees for Erlang expression, see the abstract format documentation in the ERTS User’s Guide.

- `answer() = term()`
- `answers() = [answer()]`
- `cache() = ets | list | no`
- `match_expression() = ets:match_spec()`

Match specification, see the match specification documentation in the ERTS User’s Guide and ms_transform(3).

- `no_files() = integer() >= 1`  
  Actually an integer > 1.
key_pos() = integer() >= 1 | [integer() >= 1]
max_list_size() = integer() >= 0
order() = ascending | descending | order_fun()
order_fun() = fun((term(), term()) -> boolean())

A query cursor.

query_cursor()

A query handle.

query_handle()

query_handle_or_list() = query_handle() | list()
query_list_comprehension() = term()

A literal query list comprehension.

spawn_options() = default | [proc_lib:spawn_option()]
sort_options() = [sort_option()] | sort_option()
sort_option() = {compressed, boolean()}
  | {no_files, no_files()}
  | {order, order()}
  | {size, integer() >= 1}
  | {tmpdir, tmp_directory()}</
  | {unique, boolean()}

See file_sorter(3).

tmp_directory() = [] | file:name()
tmp_file_usage() = allowed
  | not_allowed
  | info_msg
  | warning_msg
  | error_msg

Exports

append(QHL) -> QH
Types:
  QHL = [query_handle_or_list()]
  QH = query_handle()

Returns a query handle. When evaluating the query handle QH all answers to the first query handle in QHL are returned followed by all answers to the rest of the query handles in QHL.

append(QH1, QH2) -> QH3
Types:
  QH1 = QH2 = query_handle_or_list()
  QH3 = query_handle()

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(QH) -> Cursor
cursor(QH, Options) -> Cursor

Types:
  QH = query_handle_or_list()
  Options = [Option] | Option
  Option = {cache_all, cache()}
  | cache_all
  | {max_list_size, max_list_size()}
  | {spawn_options, spawn_options()}
  | {tmpdir_usage, tmp_file_usage()}
  | {tmpdir, tmp_directory()}
  | {unique_all, boolean()}
  | unique_all

Cursor = query_cursor()

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

cursor(QH) is equivalent to cursor(QH, []).

delete_cursor(QueryCursor) -> ok

Types:
  QueryCursor = query_cursor()

Deletes a query cursor. Only the owner of the cursor can delete the cursor.

eval(QH) -> Answers | Error
eval(QH, Options) -> Answers | Error
e(QH) -> Answers | Error
e(QH, Options) -> Answers | Error

Types:
  QH = query_handle_or_list()
  Options = [Option] | Option
  Option = {cache_all, cache()}
  | cache_all
  | {max_list_size, max_list_size()}
  | {tmpdir_usage, tmp_file_usage()}

132 | Ericsson AB. All Rights Reserved.: STDLIB
Evaluates a query handle in the calling process and collects all answers in a list.

```erl
1> QH = qlc:q([{X,Y} || X <- [a,b], Y <- [1,2]]),
   qlc:eval(QH).
[{a,1},{a,2},{b,1},{b,2}]
```

eval(QH) is equivalent to eval(QH, []).

```
fold(Function, Acc0, QH) -> Acc1 | Error
fold(Function, Acc0, QH, Options) -> Acc1 | Error
```

Types:
- `QH = query_handle_or_list()`
- `Function = fun((answer(), AccIn) -> AccOut)`
- `Acc0 = Acc1 = AccIn = AccOut = term()`
- `Options = [Option] | Option`
- `Option = {cache_all, cache()}`
- `| cache_all`
- `| {max_list_size, max_list_size()}`
- `| {tmpdir_usage, tmp_file_usage()}`
- `| {tmpdir, tmp_directory()}`
- `| {unique_all, boolean()}`
- `| unique_all`
- `Error = {error, module(), Reason}`
- `Reason = file_sorter:reason()`

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.

```erl
1> QH = [1,2,3,4,5,6],
   qlc:fold(fun(X, Sum) -> X + Sum end, 0, QH).
21
```

fold(Function, Acc0, QH) is equivalent to fold(Function, Acc0, QH, []).

```
format_error(Error) -> Chars
```

Types:
qlc

Error = {error, module(), term()}
Chars = io_lib: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(QH) -> Info
info(QH, Options) -> Info
Types:
QH = query_handle_or_list()
Options = [Option] | Option
Option = EvalOption | ReturnOption
EvalOption = {cache_all, cache()}
| cache_all
| {max_list_size, max_list_size()}
| {tmpdir_usage, tmp_file_usage()}
| {tmpdir, tmp_directory()}
| {unique_all, boolean()}
| unique_all
ReturnOption = {depth, Depth}
| {flat, boolean()}
| {format, Format}
| {n_elements, NElements}
Depth = infinity | integer() >= 0
Format = abstract_code | string
NElements = infinity | integer() >= 1
Info = abstract_expr() | 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 ||

314 | Ericsson AB. All Rights Reserved.: STDLIB


In this example two simple QLCs have been inserted just to hold the \{unique, true\} option.

```
\begin{verbatim}
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)]).
\end{verbatim}
```

In this example the query list comprehension \texttt{V2} has been inserted to show the joined generators and the join method chosen. A convention is used for lookup join: the first generator (\texttt{G2}) is the one traversed, the second one (\texttt{G1}) is the table where constants are looked up.

\texttt{info(Q)} is equivalent to \texttt{info(QH, [])}.

**keysort(KeyPos, QH1) -> QH2**

**keysort(KeyPos, QH1, SortOptions) -> QH2**

Types:
KeyPos = \textit{key\_pos}()
SortOptions = \textit{sort\_options}()
QH1 = \textit{query\_handle\_or\_list}()
QH2 = \textit{query\_handle}()

Returns a query handle. When evaluating the query handle QH2 the answers to the query handle QH1 are sorted by \textit{file\_sorter:keysort/4} 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 \textit{Size} bytes, where \textit{Size} is the value of the \textit{size} option. \textit{keysort} (KeyPos, QH1) is equivalent to \textit{keysort} (KeyPos, QH1, []).

next\_answers(QueryCursor) -> Answers | Error
next\_answers(QueryCursor, NumberOfAnswers) -> Answers | Error

Types:
\begin{itemize}
  \item QueryCursor = \textit{query\_cursor}()
  \item Answers = \textit{answers}()
  \item NumberOfAnswers = all\_remaining | integer() >= 1
  \item Error = \{error, module(), Reason\}
  \item Reason = \textit{file\_sorter:reason}()
\end{itemize}

Returns some or all of the remaining answers to a query cursor. Only the owner of \textit{QueryCursor} can retrieve answers.

The optional argument \textit{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 \textit{next\_answers} will return [].

q(QLC) -> QH
q(QLC, Options) -> QH

Types:
\begin{itemize}
  \item QH = \textit{query\_handle}()
  \item Options = [Option] | Option
  \item Option = \{max\_lookup, MaxLookup\}
    \begin{itemize}
      \item cache, \textit{cache()}
      \item cache
      \item \{join, Join\}
      \item \{lookup, Lookup\}
      \item \{unique, boolean()\}
      \item unique
    \end{itemize}
  \item MaxLookup = integer() >= 0 | infinity
  \item Join = any | lookup | merge | nested\_loop
  \item Lookup = boolean() | any
  \item QLC = \textit{query\_list\_comprehension}()
\end{itemize}

Returns a query handle for a query list comprehension. The query list comprehension must be the first argument to \textit{qlc:q/1,2} or it will be evaluated as an ordinary list comprehension. It is also necessary to add the line

\texttt{-include\_lib("stdlib\_include\_qlc\_hrl").}
to the source file. This causes a parse transform to substitute a fun for the query list comprehension. The (compiled) fun will be called when the query handle is evaluated.

When calling qlc:q/1,2 from the Erlang shell the parse transform is automatically called. When this happens the fun substituted for the query list comprehension is not compiled but will be evaluated by erl_eval(3). This is also true when expressions are evaluated by means of file:eval/1,2 or in the debugger.

To be very explicit, this will not work:

```erlang
...  
A = [X || {X} <- [{1},{2}]],  
QH = qlc:q(A),  
...  
```

The variable A will be bound to the evaluated value of the list comprehension ([1,2]). The compiler complains with an error message ("argument is not a query list comprehension"); the shell process stops with a badarg reason.

q(QLC) is equivalent to q(QLC, []).

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.

```erlang
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}),  
ioutil:format("\~{s}-n", [qlc:info(Q)]).  
begin  
  V1 =  
  qlc:q([  
    P0 ||  
    P0 = {X,Z} <-  

```
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 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.

In this example using the gb_table module from the Implementing a QLC table 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.

\[
\begin{align*}
\text{sort}(QH1) & \rightarrow QH2 \\
\text{sort}(QH1, \text{SortOptions}) & \rightarrow QH2
\end{align*}
\]

Types:
- SortOptions = \text{sort\_options()}
- QH1 = \text{query\_handle\_or\_list()}
- QH2 = \text{query\_handle()}

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} according to the options.

The sorter will use temporary files only if QH1 does not evaluate to a list and the size of the binary representation of the answers exceeds Size bytes, where Size is the value of the size option.

\text{sort}(QH1) is equivalent to \text{sort}(QH1, []).

\[
\begin{align*}
\text{string\_to\_handle}(\text{QueryString}) & \rightarrow QH | \text{Error} \\
\text{string\_to\_handle}(\text{QueryString}, \text{Options}) & \rightarrow QH | \text{Error} \\
\text{string\_to\_handle}(\text{QueryString}, \text{Options}, \text{Bindings}) & \rightarrow QH | \text{Error}
\end{align*}
\]

Types:
- QueryString = \text{string()}
- Options = [Option] | Option
  - Option = \{\text{max\_lookup, MaxLookup}\} \\
  - \{\text{cache, \text{cache}}()\} \\
  - \text{cache} \\
  - \{\text{join, Join}\} \\
  - \{\text{lookup, Lookup}\} \\
  - \{\text{unique, boolean()}\} \\
  - \text{unique}
- MaxLookup = \text{integer()} \geq 0 | \text{infinity}
- Join = \text{any} | \text{lookup} | \text{merge} | \text{nested\_loop}
- Lookup = \text{boolean()} | \text{any}
- Bindings = \text{erl\_eval:\text{binding\_struct}}() \\
- QH = \text{query\_handle()}
- Error = \{\text{error, module(), Reason}\}
  - Reason = \text{erl\_parse:error\_info()} | \text{erl\_scan:error\_info()}

A string version of \text{qlc:q/1,2}. When the query handle is evaluated the fun created by the parse transform is interpreted by \text{erl\_eval}. The query string is to be one single query list comprehension terminated by a period.
L = [1,2,3],
Bs = erl_eval:add_binding('L', L, erl_eval:new_bindings()),
QH = qlc:string_to_handle("[X+1 || X <- L].", [], Bs),
qlc:eval(QH).
[2,3,4]

string_to_handle(QueryString) is equivalent to string_to_handle(QueryString, []).
string_to_handle(QueryString, Options) is equivalent to string_to_handle(QueryString, Options, erl_eval:new_bindings()).

This function is probably useful mostly when called from outside of Erlang, for instance from a driver written in C.

table(TraverseFun, Options) -> QH
Types:
  TraverseFun = TraverseFun0 | TraverseFun1
  TraverseFun0 = fun(() -> TraverseResult)
  TraverseFun1 = fun((match_expression()) -> TraverseResult)
  TraverseResult = Objects | term()
  Objects = [] | [term() | ObjectList]
  ObjectList = TraverseFun0 | Objects
  Options = [Option] | Option
  Option = {format_fun, FormatFun}
          | {info_fun, InfoFun}
          | {lookup_fun, LookupFun}
          | {parent_fun, ParentFun}
          | {post_fun, PostFun}
          | {pre_fun, PreFun}
          | {key_equality, KeyComparison}
  FormatFun = undefined | fun((SelectedObjects) -> FormatedTable)
  SelectedObjects = all
                       | {all, NElements, DepthFun}
                       | {match_spec, match_expression()}
                       | {lookup, Position, Keys}
| lookup, Position, Keys, NElements, DepthFun |
NElements = infinity | integer() >= 1
DepthFun = fun(term()) -> term() |
FormattedTable = Mod, Fun, Args | abstract_expr() | string()
InfoFun = undefined | fun(InfoTag) -> InfoValue |
InfoTag = indices | is_unique_objects | keypos | num_of_objects
InfoValue = undefined | term() |
LookupFun = undefined | fun(Position, Keys) -> LookupResult |
LookupResult = term() |
ParentFun = undefined | fun(PreArgs) -> ParentFunValue |
PostFun = undefined | fun() -> term() |
PreFun = undefined | fun(PreArgs) -> term() |
PreArgs = {parent_value, ParentFunValue} |
PreArg = stop_fun, StopFun |
ParentFunValue = undefined | term() |
StopFun = undefined | fun() -> term() |
KeyComparison = '==' | '==' |
Position = integer() >= 1 |
Keys = term() |
Mod = Fun = atom() |
Args = term() |
QH = query_handle()

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 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` 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` for displaying the call that created the table's query handle. The default value, `undefined`, means that `info/1,2` displays a call to `'$MOD':'$FUN'/0`. It is up to `FormatFun` to present the selected objects of the table in a suitable way. However, if a character list is chosen for presentation it must be an Erlang expression that can be scanned and parsed (a trailing dot will be added by `qlc:info` though). `FormatFun` is called with an argument that describes the selected objects based on optimizations done as a result of analyzing the filters of the QLC where the call to `qlc:table/2` occurs. The possible values of the argument are:

- `{lookup, Position, Keys, NElements, DepthFun}`. `LookupFun` is used for looking up objects in the table.
- `{match_spec, MatchExpression}`. No way of finding all possible answers by looking up keys was found, but the filters could be transformed into a match specification. All answers are found by calling `TraverseFun(MatchExpression)`.
- `{all, NElements, DepthFun}`. No optimization was found. A match specification matching all objects will be used if `TraverseFun` is unary.

`NElements` is the value of the `info/1,2` option `n_elements`, and `DepthFun` is a function that can be used for limiting the size of terms; calling `DepthFun(Term)` substitutes `...'` for parts of `Term` below the depth specified by the `info/1,2` option `depth`. If calling `FormatFun` with an argument including `NElements` and `DepthFun` fails, `FormatFun` is called once again with an argument excluding `NElements` and `DepthFun` (`{lookup, Position, Keys}` or all).

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)`, `dets(3)` and `mnesia(3)` for the various options recognized by `table/1,2` in respective module.

See Also

`dets(3)`, `Erlang Reference Manual`, `erl_eval(3)`, `erlang(3)`, `ets(3)`, `file(3)`, `error_logger(3)`, `file_sorter(3)`, `mnesia(3)`, `Programming Examples`, `shell(3)`
This module implements (double ended) FIFO queues in an efficient manner.

All functions fail with reason badarg if arguments are of wrong type, for example queue arguments are not queues, indexes are not integers, list arguments are not lists. Improper lists cause internal crashes. An index out of range for a queue also causes a failure with reason badarg.

Some functions, where noted, fail with reason empty for an empty queue.

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 len/1, join/2, split/2, filter/2 and 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 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" 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

Data Types

queue(Item)

As returned by new/0.

queue()

queue() is equivalent to queue(term()).

Exports

new() -&gt; queue()

Returns an empty queue.
is_queue(Term :: term()) -> boolean()
Tests if Term is a queue and returns true if so and false otherwise.

is_empty(Q :: queue()) -> boolean()
Tests if Q is empty and returns true if so and false otherwise.

len(Q :: queue()) -> integer() >= 0
Calculates and returns the length of queue Q.

in(Item, Q1 :: queue(Item)) -> Q2 :: queue(Item)
Inserts Item at the rear of queue Q1. Returns the resulting queue Q2.

in_r(Item, Q1 :: queue(Item)) -> Q2 :: queue(Item)
Inserts Item at the front of queue Q1. Returns the resulting queue Q2.

out(Q1 :: queue(Item)) ->
    {{value, Item}, Q2 :: queue(Item)} | {empty, Q1 :: queue(Item)}
Removes the item at the front of queue Q1. Returns the tuple {{value, Item}, Q2}, where Item is the item removed and Q2 is the resulting queue. If Q1 is empty, the tuple {empty, Q1} is returned.

out_r(Q1 :: queue(Item)) ->
    {{value, Item}, Q2 :: queue(Item)} | {empty, Q1 :: queue(Item)}
Removes the item at the rear of the queue Q1. Returns the tuple {{value, Item}, Q2}, where Item is the item removed and Q2 is the new queue. If Q1 is empty, the tuple {empty, Q1} is returned.

from_list(L :: [Item]) -> queue(Item)
Returns a queue containing the items in L in the same order; the head item of the list will become the front item of the queue.

to_list(Q :: queue(Item)) -> [Item]
Returns a list of the items in the queue in the same order; the front item of the queue will become the head of the list.

reverse(Q1 :: queue(Item)) -> Q2 :: queue(Item)
Returns a queue Q2 that contains the items of Q1 in the reverse order.

split(N :: integer() >= 0, Q1 :: queue(Item)) ->
    {Q2 :: queue(Item), Q3 :: queue(Item)}
Splits Q1 in two. The N front items are put in Q2 and the rest in Q3.

join(Q1 :: queue(Item), Q2 :: queue(Item)) -> Q3 :: queue(Item)
Returns a queue Q3 that is the result of joining Q1 and Q2 with Q1 in front of Q2.
filter(Fun, Q1 :: queue(Item)) -> Q2 :: queue(Item)
Types:
   Fun = fun((Item) -> boolean() | [Item])
Returns a queue Q2 that is the result of calling Fun(Item) on all items in Q1, in order from front to rear.
If Fun(Item) returns true, Item is copied to the result queue. If it returns false, Item is not copied. If it returns a list the list elements are inserted instead of Item in the result queue.
So, Fun(Item) returning [Item] is thereby semantically equivalent to returning true, just as returning [] is semantically equivalent to returning false. But returning a list builds more garbage than returning an atom.

member(Item, Q :: queue(Item)) -> boolean()
Returns true if Item matches some element in Q, otherwise false.

Extended API

Exports

get(Q :: queue(Item)) -> Item
Returns Item at the front of queue Q.
Fails with reason empty if Q is empty.

get_r(Q :: queue(Item)) -> Item
Returns Item at the rear of queue Q.
Fails with reason empty if Q is empty.

drop(Q1 :: queue(Item)) -> Q2 :: queue(Item)
Returns a queue Q2 that is the result of removing the front item from Q1.
Fails with reason empty if Q1 is empty.

drop_r(Q1 :: queue(Item)) -> Q2 :: queue(Item)
Returns a queue Q2 that is the result of removing the rear item from Q1.
Fails with reason empty if Q1 is empty.

peek(Q :: queue(Item)) -> empty | {value, Item}
Returns the tuple (value, Item) where Item is the front item of Q, or empty if Q is empty.

peek_r(Q :: queue(Item)) -> empty | {value, Item}
Returns the tuple (value, Item) where Item is the rear item of Q, or empty if Q is empty.
Okasaki API

Exports

cons(Item, Q1 :: queue(Item)) -> Q2 :: queue(Item)
Inserts Item at the head of queue Q1. Returns the new queue Q2.

head(Q :: queue(Item)) -> Item
Returns Item from the head of queue Q.
Fails with reason empty if Q is empty.

tail(Q1 :: queue(Item)) -> Q2 :: queue(Item)
Returns a queue Q2 that is the result of removing the head item from Q1.
Fails with reason empty if Q1 is empty.

snoc(Q1 :: queue(Item), Item) -> Q2 :: queue(Item)
Inserts Item as the tail item of queue Q1. Returns the new queue Q2.

daeh(Q :: queue(Item)) -> Item
last(Q :: queue(Item)) -> Item
Returns the tail item of queue Q.
Fails with reason empty if Q is empty.

liat(Q1 :: queue(Item)) -> Q2 :: queue(Item)
init(Q1 :: queue(Item)) -> Q2 :: queue(Item)
lait(Q1 :: queue(Item)) -> Q2 :: queue(Item)
Returns a queue Q2 that is the result of removing the tail item from Q1.
Fails with reason empty if Q1 is empty.

The name lait/1 is a misspelling - do not use it anymore.
random
Erlang module

Random number generator. The method is attributed to B.A. Wichmann and I.D.Hill, in 'An efficient and portable

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.

It should be noted that this random number generator is not cryptographically strong. If a strong cryptographic random
number generator is needed for example crypto:rand_bytes/1 could be used instead.

Data Types
ran() = {integer(), integer(), integer()}
The state.

Exports

seed() -> ran()
Seeds random number generation with default (fixed) values in the process dictionary, and returns the old state.

seed(A1, A2, A3) -> undefined | ran()
Types:
    A1 = A2 = A3 = integer()
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),
    ...
```

seed(X1 :: {A1, A2, A3}) -> undefined | ran()
Types:
    A1 = A2 = A3 = integer()

seed({A1, A2, A3}) is equivalent to seed(A1, A2, A3).

seed0() -> ran()
Returns the default state.
random

uniform() -> float()
Returns a random float uniformly distributed between 0.0 and 1.0, updating the state in the process dictionary.

uniform(N) -> integer() >= 1
Types:
    N = integer() >= 1
Given an integer N >= 1, uniform/1 returns a random integer uniformly distributed between 1 and N, updating the state in the process dictionary.

uniform_s(State0) -> {float(), State1}
Types:
    State0 = State1 = ran()
Given a state, uniform_s/1 returns a random float uniformly distributed between 0.0 and 1.0, and a new state.

uniform_s(N, State0) -> {integer(), State1}
Types:
    N = integer() >= 1
    State0 = State1 = ran()
Given an integer N >= 1 and a state, uniform_s/2 returns a random integer uniformly distributed between 1 and N, and a new state.

Note
Some of the functions use the process dictionary variable random_seed to remember the current seed.
If a process calls uniform/0 or uniform/1 without setting a seed first, seed/0 is called automatically.
The implementation changed in R15. Upgrading to R15 will break applications that expect a specific output for a given seed. The output is still deterministic number series, but different compared to releases older than R15. The seed {0,0,0} will for example no longer produce a flawed series of only zeros.
re
Erlang module

This module contains regular expression matching functions for strings and binaries.
The regular expression syntax and semantics resemble that of Perl.
The library’s matching algorithms are currently based on the PCRE library, but not all of the PCRE library is interfaced
and some parts of the library go beyond what PCRE offers. The sections of the PCRE documentation which are relevant
to this module are included here.

Note:
The Erlang literal syntax for strings uses the "\" (backslash) character as an escape code. You need to escape
backslashes in literal strings, both in your code and in the shell, with an additional backslash, i.e.: "\\".

Data Types
mp() = {re_pattern, term(), term(), term(), term()}
Opaque datatype containing a compiled regular expression. The mp() is guaranteed to be a tuple() having the atom
re_pattern as its first element, to allow for matching in guards. The arity of the tuple() or the content of the other
fields may change in future releases.
nl_spec() = cr | crlf | lf | anycrlf | any
compile_option() = unicode
| anchored
| caseless
| dollar_endonly
| dotall
| extended
| firstline
| multiline
| no_auto_capture
| dupnames
| ungreedy
| {newline, nl_spec()}
| bsr_anycrlf
| bsr_unicode
| no_start_optimize
| ucp
| never_utf

Exports

compile(Regexp) -> {ok, MP} | {error, ErrSpec}
Types:
Regexp = iodata()
MP = mp()
ErrSpec =
    {ErrString :: string(), Position :: integer() >= 0}
The same as compile(Regexp, [])

compile(Regexp, Options) -> {ok, MP} | {error, ErrSpec}
Types:
    Regexp = iodata() | unicode:charlist()
    Options = [Option]
    Option = compile_option()
    MP = mp()
    ErrSpec =
    {ErrString :: string(), Position :: integer() >= 0}
This function compiles a regular expression with the syntax described below into an internal format to be used later as a parameter to the run/2,3 functions.
Compiling the regular expression before matching is useful if the same expression is to be used in matching against multiple subjects during the program's lifetime. Compiling once and executing many times is far more efficient than compiling each time one wants to match.
When the unicode option is given, the regular expression should be given as a valid Unicode charlist(), otherwise as any valid iodata().
The options have the following meanings:
unicode
    The regular expression is given as a Unicode charlist() and the resulting regular expression code is to be run against a valid Unicode charlist() subject. Also consider the ucp option when using Unicode characters.
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 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 this option's setting.
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.
- any
  - Any of the newline sequences above, plus the Unicode sequences VT (vertical tab, U+000B), 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).

**no_start_optimize**

This option disables optimization that may malfunction if "Special start-of-pattern items" are present in the regular expression. A typical example would be when matching "DEFABC" against "(*COMMIT)ABC", where the start optimization of PCRE would skip the subject up to the "A" and would never realize that the
(*COMMIT) instruction should have made the matching fail. This option is only relevant if you use "start-of-pattern items", as discussed in the section "PCRE regular expression details" below.

ucp
Specifies that Unicode Character Properties should be used when resolving \B, \b, \d, \D, \s, \S, \W and \w. Without this flag, only ISO-Latin-1 properties are used. Using Unicode properties hurts performance, but is semantically correct when working with Unicode characters beyond the ISO-Latin-1 range.

never_utf
Specifies that the (*UTF) and/or (*UTF8) "start-of-pattern items" are forbidden. This flag can not be combined with unicode. Useful if ISO-Latin-1 patterns from an external source are to be compiled.

inspect(MP, Item) -> {namelist, [binary()]} Types: MP = mp() Item = namelist
This function takes a compiled regular expression and an item, returning the relevant data from the regular expression. Currently the only supported item is namelist, which returns the tuple {namelist, [binary()]} containing the names of all (unique) named subpatterns in the regular expression.

Example:

```
1> {ok,MP} = re:compile("(?<A>A)|(?<B>B)|(?<C>C)").
{ok,{re_pattern,3,0,0,
    <<69,82,67,80,119,0,0,0,0,0,0,0,0,1,0,0,0,255,255,255,255,
    255,255,...>>}}
2> re:inspect(MP,namelist).
{namelist,[['A'>>,['B'>>,['C']>>]]}
3> {ok,MPD} = re:compile("(?<C>A)|(?<B>B)|(?<C>C)",[dupnames]).
{ok,{re_pattern,3,0,0,
    <<69,82,67,80,119,0,0,0,0,0,0,0,0,1,0,0,0,255,255,255,255,
    255,255,...>>}}
4> re:inspect(MPD,namelist).
{namelist,[['B'>>,['C']>>]]}
```

Note specifically in the second example that the duplicate name only occurs once in the returned list, and that the list is in alphabetical order regardless of where the names are positioned in the regular expression. The order of the names is the same as the order of captured subexpressions if {capture, all_names} is given as an option to re:run/3. You can therefore create a name-to-value mapping from the result of re:run/3 like this:

```
1> {ok,MP} = re:compile("(?<A>A)|(?<B>B)|(?<C>C)").
{ok,{re_pattern,3,0,0,
    <<69,82,67,80,119,0,0,0,0,0,0,0,0,1,0,0,0,255,255,255,255,
    255,255,...>>}}
2> {namelist, N} = re:inspect(MP,namelist).
{namelist,[['A'>>,['B'>>,['C']>>]]}
3> {match,L} = re:run("AA",MP,[{capture,all_names,binary}]).
{match,[['A'>>,<<>>,<<>>]}]
4> NameMap = lists:zip(N,L).
{{[['A'>>,<<>>,<<>>]},[['B'>>,<<>>,<<>>]},[['C'>>,<<>>]}}
```

More items are expected to be added in the future.
run(Subject, RE) -> {match, Captured} | nomatch

Types:
  Subject = iodata() | unicode:charlist()
  RE = mp() | iodata()
  Captured = [CaptureData]
  CaptureData = {integer(), integer()}

The same as run(Subject,RE,[]).

run(Subject, RE, Options) ->
  {match, Captured} | match | nomatch | {error, ErrType}

Types:
  Subject = iodata() | unicode:charlist()
  RE = mp() | iodata() | unicode:charlist()
  Options = [Option]
  Option = anchored
    | global
    | notbol
    | noteol
    | notempty
    | notempty_atstart
    | report_errors
    | {offset, integer() >= 0}
    | {match_limit, integer() >= 0}
    | {match_limit_recursion, integer() >= 0}
    | {newline, NLSpec :: nl_spec()}
    | bsr_anycrlf
    | bsr_unicode
    | {capture, ValueSpec}
    | {capture, ValueSpec, Type}
    | CompileOpt
  Type = index | list | binary
  ValueSpec = all
    | all_but_first
    | all_names
    | first
    | none
    | ValueList
  ValueList = [ValueID]
  ValueID = integer() | string() | atom()
  CompileOpt = compile_option()

See compile/2 above.

Captured = [CaptureData] | [[CaptureData]]
CaptureData = {integer(), integer()}
  | ListConversionData
  | binary()
ListConversionData = string()
  | {error, string(), binary()}
Executes a regexp matching, returning \texttt{match/\{match, Captured\}} or \texttt{nomatch}. The regular expression can be given either as \texttt{iodata()} in which case it is automatically compiled (as by \texttt{re:compile/2}) and executed, or as a pre-compiled \texttt{mp()} in which case it is executed against the subject directly.

When compilation is involved, the exception \texttt{badarg} is thrown if a compilation error occurs. Call \texttt{re:compile/2} to get information about the location of the error in the regular expression.

If the regular expression is previously compiled, the option list can only contain the options \texttt{anchored}, \texttt{global}, \texttt{notbol}, \texttt{noteol}, \texttt{report_errors}, \texttt{notempty}, \texttt{notempty_atstart}, \{\texttt{offset}, integer() \geq 0\}, \{\texttt{match_limit}, integer() \geq 0\}, \{\texttt{match_limit_recursion}, integer() \geq 0\}, \{\texttt{newline}, NLSpec\} and \{\texttt{capture}, ValueSpec\}/\{\texttt{capture}, ValueSpec, Type\}. Otherwise all options valid for the \texttt{re:compile/2} function are allowed as well. Options allowed both for compilation and execution of a match, namely \texttt{anchored} and \{\texttt{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 \texttt{unicode}, the \texttt{Subject} should be provided as a valid Unicode charlist(), otherwise any iodata() will do. If compilation is involved and the option \texttt{unicode} is given, both the Subject and the regular expression should be given as valid Unicode charlists().

The \{\texttt{capture}, ValueSpec\}/\{\texttt{capture}, ValueSpec, Type\} defines what to return from the function upon successful matching. The \texttt{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 \texttt{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 (\{\texttt{capture, none}\}), the function will return the single atom \texttt{match} upon successful matching, otherwise the tuple \{\texttt{match, ValueList}\} is returned.

Disabling capturing can be done either by specifying none or an empty list as ValueSpec.

The \texttt{report_errors} option adds the possibility that an error tuple is returned. The tuple will either indicate a matching error (\texttt{match_limit} or \texttt{match_limit_recursion}) or a compilation error, where the error tuple has the format \{\texttt{error, \{compile, CompileErr\}}\}. Note that if the option \texttt{report_errors} is not given, the function never returns error tuples, but will report compilation errors as a \texttt{badarg} exception and failed matches due to exceeded match limits simply as \texttt{nomatch}.

The options relevant for execution are:

\begin{itemize}
  \item \texttt{anchored} Limits \texttt{re:run/3} to matching at the first matching position. If a pattern was compiled with \texttt{anchored}, or turned out to be anchored by virtue of its contents, it cannot be made unanchored at matching time, hence there is no \texttt{unanchored} option.
  \item \texttt{global} Implements global (repetitive) search (the g flag in Perl). Each match is returned as a separate list() containing the specific match as well as any matching subexpressions (or as specified by the \texttt{capture} option). The Captured part of the return value will hence be a list() of list()s when this option is given.
\end{itemize}

The interaction of the global option with a regular expression which matches an empty string surprises some users. When the global option is given, \texttt{re:run/3} handles empty matches in the same way as Perl: a zero-length match at any point will be retried with the options \{\texttt{anchored, notempty_atstart}\} as well. If that search gives a result of length \geq 0, the result is included. For example:
The following matching will be performed:

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 \[anchored, notempty_atstart\]

The search is retried with the options \[anchored, notempty_atstart\] at the same position, which does not give any interesting result of longer length, so the search position is now advanced to the next character (a).

At offset 1

This time, the search results in \(\{1,0\}, \{1,0\}\). so this search will also be repeated with the extra options.

At offset 1 with \[anchored, notempty_atstart\]

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 \[anchored, notempty_atstart\]

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:

\[
\{\text{match, }\{\{0,0\}, \{0,0\}\}, \{\{1,0\}, \{1,0\}\}, \{\{1,2\}, \{1,2\}\}, \{\{3,0\}, \{3,0\}\}\}\}
\]

**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 would normally match the empty string at the start of the subject. With the notempty option, this match is not valid, so re:run/3 searches further into the string for occurrences of "a" or "b".

**notempty_atstart**

This is like notempty, except that an empty string match that is not at the start of the subject is permitted. If the pattern is anchored, such a match can occur only if the pattern contains \K.

Perl has no direct equivalent of notempty or notempty_atstart, 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_atstart 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 only affects the behavior of the circumflex metacharacter. It does not affect \A.
note

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.

report_errors

This option gives better control of the error handling in re:run/3. When it is given, compilation errors (if the regular expression isn't already compiled) as well as run-time errors are explicitly returned as an error tuple.

The possible run-time errors are:

match_limit

The PCRE library sets a limit on how many times the internal match function can be called. The default value for this is 10000000 in the library compiled for Erlang. If \{error, match_limit\} is returned, it means that the execution of the regular expression has reached this limit. Normally this is to be regarded as a nomatch, which is the default return value when this happens, but by specifying report_errors, you will get informed when the match fails due to too many internal calls.

match_limit_recursion

This error is very similar to match_limit, but occurs when the internal match function of PCRE is "recursively" called more times than the "match_limit_recursion" limit, which is by default 10000000 as well. Note that as long as the match_limit and match_limit_default values are kept at the default values, the match_limit_recursion error can not occur, as the match_limit_error will occur before that (each recursive call is also a call, but not vice versa). Both limits can however be changed, either by setting limits directly in the regular expression string (see reference section below) or by giving options to re:run/3.

It is important to understand that what is referred to as "recursion" when limiting matches is not actually recursion on the C stack of the Erlang machine, neither is it recursion on the Erlang process stack. The version of PCRE compiled into the Erlang VM uses machine "heap" memory to store values that needs to be kept over recursion in regular expression matches.

\{match_limit, integer() >= 0\}

This option limits the execution time of a match in an implementation-specific way. It is described in the following way by the PCRE documentation:

The match_limit field provides a means of preventing PCRE from using up a vast amount of resources when running patterns that are not going to match, but which have a very large number of possibilities in their search trees. The classic example is a pattern that uses nested unlimited repeats.

Internally, pcre_exec() uses a function called match(), which it calls repeatedly (sometimes recursively). The limit set by match_limit is imposed on the number of times this function is called during a match, which has the effect of limiting the amount of backtracking that can take place. For patterns that are not anchored, the count restarts from zero for each position in the subject string.

This means that runaway regular expression matches can fail faster if the limit is lowered using this option. The default value compiled into the Erlang virtual machine is 10000000.
Note:
This option does in no way affect the execution of the Erlang virtual machine in terms of "long running BIF's": re:run always give control back to the scheduler of Erlang processes at intervals that ensures the real time properties of the Erlang system.

{match_limit_recursion, integer() >= 0}
This option limits the execution time and memory consumption of a match in an implementation-specific way, very similar to match_limit. It is described in the following way by the PCRE documentation:

The match_limit_recursion field is similar to match_limit, but instead of limiting the total number of times that match() is called, it limits the depth of recursion. The recursion depth is smaller than the total number of calls, because not all calls to match() are recursive. This limit is of use only if it is set smaller than match_limit.

Limiting the recursion depth limits the amount of machine stack that can be used, or, when PCRE has been compiled to use memory on the heap instead of the stack, the amount of heap memory that can be used.

The Erlang virtual machine uses a PCRE library where heap memory is used when regular expression match recursion happens, why this limits the usage of machine heap, not C stack.

Specifying a lower value may result in matches with deep recursion failing, when they should actually have matched:

1> re:run("aaaaaaaaaaaaaaaaz","(a+)*z").
{match,[[0,14],[0,13]]}
2> re:run("aaaaaaaaaaaaaaaaz","(a+)*z",[{match_limit_recursion,5}]).
nomatch
3> re:run("aaaaaaaaaaaaaaaaz","(a+)*z",[{match_limit_recursion,5},report_errors]).
{error,match_limit_recursion}

This option, as well as the match_limit option should only be used in very rare cases. Understanding of the PCRE library internals is recommended before tampering with these limits.

{offset, integer() => 0}
Start matching at the offset (position) given in the subject string. The offset is zero-based, so that the default is {offset,0} (all of the subject string).

{newline, NLSpec}
Override the default definition of a newline in the subject string, which is LF (ASCII 10) in Erlang.

cr
Newline is indicated by a single character CR (ASCII 13)

lf
Newline is indicated by a single character LF (ASCII 10), the default

crlf
Newline is indicated by the two-character CRLF (ASCII 13 followed by ASCII 10) sequence.

anycrlf
Any of the three preceding sequences should be recognized.
any
Any of the newline sequences above, plus the Unicode sequences VT (vertical tab, U+000B), 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. (overrides compilation option)

bsr_unicode
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.

all_names

All named subpatterns in the regular expression, as if a list() of all the names in alphabetical order was given. The list of all names can also be retrieved with the inspect/2 function.

first

Only the first captured subpattern, which is always the complete matching part of the subject. All explicitly captured subpatterns are discarded.

all_except_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. For example, consider the regular expression:

```
.*(abcd).*
```

matched against the string "ABCabcdABC", capturing only the "abcd" part (the first explicit subpattern):

```
re:run("ABCabcdABC",".*(abcd).*",[\{capture,[1]\}]).
```

The call will yield the following result:

```
{match,[\{3,4\}]}
```

as the first explicitly captured subpattern is ")abcd)", matching "abcd" in the subject, at (zero-based) position 3, of length 4.

Now consider the same regular expression, but with the subpattern explicitly named 'FOO':

```
.*(\(<FOO>abcd)\).*
```

With this expression, we could still give the index of the subpattern with the following call:

```
re:run("ABCabcdABC",".*(\(<FOO>abcd)\).*",[\{capture,[1]\}]).
```
giving the same result as before. But, since the subpattern is named, we can also specify 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 virtual) *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 counter-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). If the `unicode` option is used in combination with the `\C` sequence in the regular expression, a captured subpattern can contain bytes that are not valid UTF-8 (`\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 `\C` sequence when capturing lists.

- **binary**
  - Return matching substrings as binaries. If the `unicode` option is used, these binaries are in UTF-8. If the `\C` sequence is used together with `unicode` the binaries may be invalid UTF-8.

In general, subpatterns that were not assigned a value in the match are returned as the tuple `{-1,0}` when `type` is `index`. Unassigned subpatterns are returned as the empty binary or list, respectively, for other return types. Consider the regular expression:

```
".*((?<FOO>abdd)\[a..d]\).*"
```

There are three explicitly capturing subpatterns, where the opening parenthesis position determines the order in the result, hence `((?<FOO>abdd)\[a..d])` is subpattern index 1, `(?<FOO>abdd)` is subpattern index 2 and `\[a..d]` is subpattern index 3. When matched against the following string:

```
"ABCababcdABC"
```
the subpattern at index 2 won’t match, as “abdd” is not present in the string, but the complete pattern matches (due to the alternative a (...)d). The subpattern at index 2 is therefore unassigned and the default return value will be:

{match,[[0,10],[3,4],[-1,0],[4,3]]}

Setting the capture Type to binary would give the following:

{match,[<<"ABCabcdABC">>,<<"abcd">>,<<>>,<<"bcd">>,]}

where the empty binary (<<>>) represents the unassigned subpattern. In the binary case, some information about the matching is therefore lost, the <<>> might just as well be an empty string captured.

If differentiation between empty matches and non existing subpatterns is necessary, use the type index and do the conversion to the final type in Erlang code.

When the option global is given, the capture specification affects each match separately, so that:

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 re:compile/2 function.

replace(Subject, RE, Replacement) -> iodata() | unicode:charlist()

Types:
  Subject = iodata() | unicode:charlist()
  RE = mp() | iodata()
  Replacement = iodata() | unicode:charlist()

The same as replace(Subject,RE,Replacement,[]).

replace(Subject, RE, Replacement, Options) ->
  iodata() | unicode:charlist()

Types:
  Subject = iodata() | unicode:charlist()
  RE = mp() | iodata() | unicode:charlist()
  Replacement = iodata() | unicode:charlist()
  Options = [Option]
  Option = anchored
    | global
    | notbol
    | noteol
    | notempty
    | notempty_atstart
    | {offset, integer() >= 0}
    | {newline, NLSpec}
    | bsr_anycrlf
Replaces the matched part of the Subject string with the contents of Replacement.

The permissible options are the same as for re:run/3, except that the capture option 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), \g{N} 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, so a single \ has to be written as "\\" and therefore a double \ as "\\\\". Example:

```erlang
re:replace("abcd","c",[&],[return,list]).
```
gives

"ab[c]d"

while

```erlang
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() | unicode:charlist()
RE = mp() | iodata()
SplitList = [iodata() | unicode:charlist()]

The same as split(Subject,RE,[]).
split(Subject, RE, Options) -> SplitList

Types:

Subject = iodata() | unicode:charlist()
RE = mp() | iodata() | unicode:charlist()
Options = [Option]
Option = anchored
  | notbol
  | noteol
  | notempty
  | notempty_atstart
  | {offset, integer() >= 0}
  | {newline, nl_spec()}
  | {match_limit, integer() >= 0}
  | {match_limit_recursion, integer() >= 0}
  | bsr_anycrlf
  | bsr_unicode
  | {return, ReturnType}
  | {parts, NumParts}
  | group
  | trim
  | CompileOpt

NumParts = integer() >= 0 | infinity
ReturnType = iodata | list | binary
CompileOpt = compile_option()

See compile/2 above.

SplitList = [RetData] | [GroupedRetData]
GroupedRetData = [RetData]
RetData = iodata() | unicode:charlist() | binary() | list()

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 parentheses 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"]

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", so 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. Since 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. For example:

re:split("Erlang","[lg]",{{return,list}}).

will return:

["Er","an",[]]

since 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 trim as an option:

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}:

re:split("Erlang","[lg]",{{return,list},{parts,2}}).

344 | Ericsson AB. All Rights Reserved.: STDLIB
This will give:

```erlang
["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, which is why the result differs from that of `trim`.

More than three parts are not possible with this indata, so

```erlang
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 `trim`. If subexpressions are captured, empty subexpression matches at the end are also stripped from the result if `trim` or `{parts,0}` is specified.

If you are familiar with Perl, the `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 `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 `re:run/3` function:

{return,ReturnType}

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 based on the PCRE documentation, with changes in cases where the re module behaves differently to 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:

- Special start-of-pattern items
- Characters and metacharacters
- Backslash
- Circumflex and dollar
- Full stop (period, dot) and \N
- Matching a single data unit
- Square brackets and character classes
- POSIX character classes
- Vertical bar
- Internal option setting
- Subpatterns
- Duplicate subpattern numbers
- Named subpatterns
- Repetition
- Atomic grouping and possessive quantifiers
- Back references
- Assertions
- Conditional subpatterns
- Comments
- Recursive patterns
- Subpatterns as subroutines
- Oniguruma subroutine syntax
- Backtracking control

Special start-of-pattern items

A number of options that can be passed to \texttt{re:compile/2} can also be set by special items at the start of a pattern. These are not Perl-compatible, but are provided to make these options accessible to pattern writers who are not able to change the program that processes the pattern. Any number of these items may appear, but they must all be together right at the start of the pattern string, and the letters must be in upper case.

\textit{UTF support}

Unicode support is basically UTF-8 based. To use Unicode characters, you either call \texttt{re:compile/2/re:run/3} with the \texttt{unicode} option, or the pattern must start with one of these special sequences:

\texttt{(*UTF8)}

\texttt{(*UTF)}
Both options give the same effect, the input string is interpreted as UTF-8. Note that with these instructions, the automatic conversion of lists to UTF-8 is not performed by the \re functions, why using these options is not recommended. Add the `unicode` option when running `re:compile/2` instead.

Some applications that allow their users to supply patterns may wish to restrict them to non-UTF data for security reasons. If the `never_utf` option is set at compile time, (*UTF) etc. are not allowed, and their appearance causes an error.

**Unicode property support**

Another special sequence that may appear at the start of a pattern is

(*UCP)

This has the same effect as setting the `ucp` option: it causes sequences such as \d and \w to use Unicode properties to determine character types, instead of recognizing only characters with codes less than 128 via a lookup table.

**Disabling start-up optimizations**

If a pattern starts with (*NO_START_OPT), it has the same effect as setting the `no_Start_optimize` option at compile time.

**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. If more than one of them is present, the last one is used.

The newline convention affects where the circumflex and dollar assertions are true. It also affects the interpretation of the dot metacharacter when `dotall` is not set, and the behaviour of \N. However, it 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.

**Setting match and recursion limits**

The caller of `re:run/3` can set a limit on the number of times the internal match() function is called and on the maximum depth of recursive calls. These facilities are provided to catch runaway matches that are provoked by patterns with huge matching trees (a typical example is a pattern with nested unlimited repeats) and to avoid running out of system stack by too much recursion. When one of these limits is reached, `pcre_exec()` gives an error return. The limits can also be set by items at the start of the pattern of the form
where \( d \) is any number of decimal digits. However, the value of the setting must be less than the value set by the caller of \texttt{re:run/3} for it to have any effect. In other words, the pattern writer can lower the limit set by the programmer, but not raise it. If there is more than one setting of one of these limits, the lower value is used.

The current default value for both the limits are 10000000 in the Erlang VM. Note that the recursion limit does not actually affect the stack depth of the VM, as PCRE for Erlang is compiled in such a way that the match function never does recursion on the "C-stack".

**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 \texttt{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 \texttt{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:

\[
\begin{align*}
\backslash & \quad \text{general escape character with several uses} \\
\wedge & \quad \text{assert start of string (or line, in multiline mode)} \\
\$ & \quad \text{assert end of string (or line, in multiline mode)} \\
. & \quad \text{match any character except newline (by default)} \\
[ & \quad \text{start character class definition} \\
| & \quad \text{start of alternative branch} \\
( & \quad \text{start subpattern} \\
) & \quad \text{end subpattern} \\
? & \quad \text{extends the meaning of \texttt{.}, also 0 or 1 quantifier, also quantifier minimizer} \\
* & \quad \text{0 or more quantifier} \\
+ & \quad \text{1 or more quantifier, also "possessive quantifier"} \\
\{ & \quad \text{start min/max quantifier}
\end{align*}
\]

Part of a pattern that is in square brackets is called a "character class". In a character class the only metacharacters are:
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 character that is not a number or a letter, 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 \\.

In unicode mode, only ASCII numbers and letters have any special meaning after a backslash. All other characters (in particular, those whose codepoints are greater than 127) are treated as literals.

If a pattern is compiled with the extended option, white space 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 white space 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. An isolated \E that is not preceded by \Q is ignored. If \Q is not followed by \E later in the pattern, the literal interpretation continues to the end of the pattern (that is, \E is assumed at the end). If the isolated \Q is inside a character class, this causes an error, because the character class is not terminated.

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 often 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 ASCII character
\e
   escape (hex 1B)
\f
   form feed (hex 0C)
\n
   linefeed (hex 0A)
\r
   carriage return (hex 0D)
\t
   tab (hex 09)
\ddd
   character with octal code ddd, or back reference
\xhh
   character with hex code hh
\x{hhh..}
   character with hex code hhh..

The precise effect of \cx on ASCII characters 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 \cA to \cZ become hex 01 to hex 1A (A is 41, Z is 5A), but \c{ becomes hex 3B (\ is 7B), and \c; becomes hex 7B (; is 3B). If the data item (byte or 16-bit value) following \c has a value greater than 127, a compile-time error occurs. This locks out non-ASCII characters in all modes.

The \c facility was designed for use with ASCII characters, but with the extension to Unicode it is even less useful than it once was.

By default, after \x, from zero to two hexadecimal digits are read (letters can be in upper or lower case). Any number of hexadecimal digits may appear between \x{ and }, but the character code is constrained as follows:

8-bit non-Unicode mode
   less than 0x100
8-bit UTF-8 mode
   less than 0x10ffff and a valid codepoint

Invalid Unicode codepoints are the range 0xd800 to 0xdfff (the so-called "surrogate" codepoints), and 0xffef.

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, \xd is exactly the same as \x{dc}.

After \0 up to two further octal digits are read. If there are fewer than two digits, just those that are present are used. Thus the sequence \0x\07 specifies two binary zeros followed by a BEL character (code value 7). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a back reference. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and uses them to generate a data character. Any subsequent digits stand for themselves. The value of the character is constrained in the same way as characters specified in hexadecimal. For example:

\040
   is another way of writing a ASCII 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 value 255 (decimal)
\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, \b is interpreted as the backspace character (hex 08).

\N is not allowed in a character class. \B, \R, and \X are not special inside a character class. Like other unrecognized escape sequences, they are treated as the literal characters "B", "R", and "X". Outside a character class, these sequences have different meanings.

Unsupported escape sequences
In Perl, the sequences \l, \L, \u, and \U are recognized by its string handler and used to modify the case of following characters. PCRE does not support these escape sequences.

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.

Absolute and relative subroutine calls
For compatibility with Oniguruma, the non-Perl syntax \g followed by a name or a number enclosed either in angle brackets or single quotes, is an alternative syntax for referencing a subpattern as a "subroutine". Details are discussed later. Note that \g{...} (Perl syntax) and \g<...> (Oniguruma syntax) are not synonymous. The former is a back reference; the latter is a subroutine call.

Generic character types
Another use of backslash is for specifying generic character types:
\d any decimal digit
\D any character that is not a decimal digit
\h any horizontal white space character
\H any character that is not a horizontal white space character
\s any white space character
\S  
  any character that is not a white space character
\w  
  any vertical white space character
\V  
  any character that is not a vertical white space character
\w  
  any "word" character
\W  
  any "non-word" character

There is also the single sequence \N, which matches a non-newline character. This is the same as the "." metacharacter when dotall is not set. Perl also uses \N to match characters by name; PCRE does not support this.

Each pair of lower and upper case escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair. The 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, because 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 POSIX "space" class. The \s characters are HT (9), LF (10), FF (12), CR (13), and space (32). If "use locale;" is included in a Perl script, \s may match the VT character. In PCRE, it never does.

A "word" character is an underscore or any character that is a letter or digit. By default, the definition of letters and digits is controlled by PCRE's low-valued character tables, in Erlang's case (and without the unicode option), the ISO-Latin-1 character set.

By default, in unicode mode, characters with values greater than 128 never match \d, \s, or \w, and always match \D, \S, and \W. These sequences retain their original meanings from before UTF support was available, mainly for efficiency reasons. However, if the ucp option is set, the behaviour is changed so that Unicode properties are used to determine character types, as follows:

\d  
  any character that \p{Nd} matches (decimal digit)
\s  
  any character that \p{Z} matches, plus HT, LF, FF, CR)
\w  
  any character that \p{L} or \p{N} matches, plus underscore)

The upper case escapes match the inverse sets of characters. Note that \d matches only decimal digits, whereas \w matches any Unicode digit, as well as any Unicode letter, and underscore. Note also that ucp affects \b, and \B because they are defined in terms of \w and \W. Matching these sequences is noticeably slower when \ucp is set.

The sequences \h, \H, \v, and \V are features that were added to Perl at release 5.10. In contrast to the other sequences, which match only ASCII characters by default, these always match certain high-valued codepoints, whether or not \ucp is set. The horizontal space characters are:

U+0009  
  Horizontal tab (HT)
U+0020  
  Space
U+00A0  
  Non-break space
U+1680  
  Ogham space mark
U+180E  
  Mongolian vowel separator

352 | Ericsson AB. All Rights Reserved.: STDLIB
U+2000  
   En quad  
U+2001  
   Em quad  
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 (LF)  
U+000B  
   Vertical tab (VT)  
U+000C  
   Form feed (FF)  
U+000D  
   Carriage return (CR)  
U+0085  
   Next line (NEL)  
U+2028  
   Line separator  
U+2029  
   Paragraph separator  

In 8-bit, non-UTF-8 mode, only the characters with codepoints less than 256 are relevant.

Newline sequences
Outside a character class, by default, the escape sequence \R matches any Unicode newline sequence. In non-UTF-8 mode \R is equivalent to the following:
(?>\n|\r|\f|\u000a|\u000d|\u0085)  

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 (form feed, 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 Unicode 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_unicode option. It is also possible to specify these settings by starting a pattern string with one of the following sequences:

(*BSR_ANYCRRLF) CR, LF, or CRLF only (*BSR_UNICODE) any Unicode newline sequence

These override the default and the options given to the compiling function, but they can themselves be overridden by options given to a matching function. 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_ANYCRRLF)

They can also be combined with the (*UTF8), (*UTF) or (*UCP) special sequences. Inside a character class, \R is treated as an unrecognized escape sequence, and so matches the letter "R" by default.

Unicode character properties

Three additional escape sequences that match characters with specific properties are available. When in 8-bit non-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:

\p{xx}
  a character with the xx property
\P{xx}
  a character without the xx property
\X
  a Unicode extended grapheme cluster

The property names represented by xx above are limited to the Unicode script names, the general category properties, "Any", which matches any character (including newline), and some special PCRE properties (described in the next section). Other Perl properties such as "InMusicalSymbols" are not currently supported by PCRE. Note that \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:

\p{Greek} \P{Han}

Those that are not part of an identified script are lumped together as "Common". The current list of scripts is:

- Arabic
- Armenian
- Avestan
- Balinese
- Bamum
- Batak
- Bengali
- Bopomofo
• Braille
• Buginese
• Buhid
• Canadian_Aboriginal
• Carian
• Chakma
• Cham
• Cherokee
• Common
• Coptic
• Cuneiform
• Cypriot
• Cyrillic
• Deseret
• Devanagari
• Egyptian_Hieroglyphs
• Ethiopic
• Georgian
• Glagolitic
• Gothic
• Greek
• Gujarati
• Gurmukhi
• Han
• Hangul
• Hanunoo
• Hebrew
• Hiragana
• Imperial_Aramaic
• Inherited
• Inscriptional_Pahlavi
• Inscriptional_Parthian
• Javanese
• Kaithi
• Kannada
• Katakana
• Kayah_Li
• Kharoshthi
• Khmer
• Lao
• Latin
• Lepcha
• Limbu
• Linear_B
• Lisu
• Lycian
• Lydian
• Malayalam
• Mandaic
• Meetei_Mayek
• Meroitic_Cursive
• Meroitic_Hieroglyphs
• Miao
• Mongolian
• Myanmar
• New_Tai_Lue
• Nko
• Ogham
• Old_Italic
• Old_Persian
• Oriya
• Old_South_Arabian
• Old_Turkic
• Ol_Chiki
• Osmany
• Phags_Pa
• Phoenician
• Rejang
• Runic
• Samaritan
• Saurashtra
• Sharada
• Shavian
• Sinhala
• Sora_Sompeng
• Sundanese
• Syloti_Nagri
• Syriac
• Tagalog
• Tagbanwa
• Tai_Le
• Tai_Tham
• Tai_Viet
• Takri
• Tamil
• Telugu
• Thaana
• Thai
• Tibetan
• Tifinagh
• Ugaritic
• Vai
• Yi

Each character has exactly one Unicode 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, \p{^Lu} is the same as \P{Lu}.

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:

• \p{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
Ll
  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
Nl  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 Unicode strings and so cannot be tested by PCRE. Perl does not support the Cs property.

The long synonyms for property names 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. This is different from the behaviour of current versions of Perl.

Matching characters by Unicode property is not fast, because PCRE has to do a multistage table lookup in order to find a character's property. That is why the traditional escape sequences such as \d and \w do not use Unicode properties in PCRE by default, though you can make them do so by setting the ucp option or by starting the pattern with (*UCP).

**Extended grapheme clusters**

The \X escape matches any number of Unicode characters that form an "extended grapheme cluster", and treats the sequence as an atomic group (see below). Up to and including release 8.31, PCRE matched an earlier, simpler definition that was equivalent to

```
(?>(?<!\P{M})\p{M}*)
```

That is, it matched a character without the "mark" property, followed by zero or more characters with the "mark" property. Characters with the "mark" property are typically non-spacing accents that affect the preceding character.

This simple definition was extended in Unicode to include more complicated kinds of composite character by giving each character a grapheme breaking property, and creating rules that use these properties to define the boundaries of extended grapheme clusters. In releases of PCRE later than 8.31, \X matches one of these clusters.

\X always matches at least one character. Then it decides whether to add additional characters according to the following rules for ending a cluster:

1. End at the end of the subject string.
2. Do not end between CR and LF; otherwise end after any control character.
3. Do not break Hangul (a Korean script) syllable sequences. Hangul characters are of five types: L, V, T, LV, and LVT. An L character may be followed by an L, V, LV, or LVT character; an LV or V character may be followed by a V or T character; an LVT or T character may be followed only by a T character.
4. Do not end before extending characters or spacing marks. Characters with the "mark" property always have the "extend" grapheme breaking property.
5. Do not end after prepend characters.
6. Otherwise, end the cluster.

**PCRE's additional properties**

As well as the standard Unicode properties described above, PCRE supports four more that make it possible to convert traditional escape sequences such as \w and \s and POSIX character classes to use Unicode properties. PCRE uses these non-standard, non-Perl properties internally when PCRE_UCP is set. However, they may also be used explicitly. These properties are:

- Xan
  - Any alphanumeric character
- Xps
  - Any POSIX space character
- Xsp
  - Any Perl space character
- Xwd
  - Any Perl "word" character
Xan matches characters that have either the L (letter) or the N (number) property. Xps matches the characters tab, linefeed, vertical tab, form feed, or carriage return, and any other character that has the Z (separator) property. Xsp is the same as Xps, except that vertical tab is excluded. Xwd matches the same characters as Xan, plus underscore.

There is another non-standard property, Xuc, which matches any character that can be represented by a Universal Character Name in C++ and other programming languages. These are the characters $, @, ` (grave accent), and all characters with Unicode code points greater than or equal to U+00A0, except for the surrogates U+D800 to U+DFFF. Note that most base (ASCII) characters are excluded. (Universal Character Names are of the form \uHHHH or \UHHHHHHHHH where H is a hexadecimal digit. Note that the Xuc property does not match these sequences but the characters that they represent.)

Resetting the match start

The escape sequence \K 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".

Perl documents that the use of \K within assertions is "not well defined". In PCRE, \K is acted upon when it occurs inside positive assertions, but is ignored in negative assertions.

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

Inside a character class, `\b` has a different meaning; it matches the backspace character. If any other of these assertions appears in a character class, by default it matches the corresponding literal character (for example, `\B` matches the letter B).

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. In a UTF mode, the meanings of \w and \W can be changed by setting the `UCP` option. When this is done, it also affects `\b` and `\B`. Neither PCRE nor Perl has a separate "start of word" or "end of word" metasequence. However, whatever follows `\b` normally determines which it is. For example, the fragment `\ba` matches "a" at the start of a word.
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

The circumflex and dollar metacharacters are zero-width assertions. That is, they test for a particular condition being true without consuming any characters from the subject string.

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.)

The 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). Note, however, that it does not actually match the newline. 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
abc" (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) and \N**

Outside a character class, a dot in the pattern matches any one character in the subject string except (by default) a character that signifies the end of a line.

When a line ending is defined as a single character, dot never matches that character: when the two-character sequence CRLF is used, dot does not match CR if it is immediately followed by LF, but otherwise it matches all characters (including isolated CRs and LFs). 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.

The escape sequence `\N` behaves like a dot, except that it is not affected by the PCRE_DOTALL option. In other words, it matches any character except one that signifies the end of a line. Perl also uses `\N` to match characters by name; PCRE does not support this.

**Matching a single data unit**

Outside a character class, the escape sequence `\C` matches any one data unit, whether or not a UTF mode is set. One data unit is one byte. Unlike a dot, `\C` always matches line-ending characters. The feature is provided in Perl in order to match individual bytes in UTF-8 mode, but it is unclear how it can usefully be used. Because `\C` breaks up characters into individual data units, matching one unit with `\C` in a UTF mode means that the rest of the string may start with a malformed UTF character. This has undefined results, because PCRE assumes that it is dealing with valid UTF strings. PCRE does not allow `\C` to appear in lookbehind assertions (described below) in a UTF mode, because this would make it impossible to calculate the length of the lookbehind.

In general, the `\C` escape sequence is best avoided. However, one way of using it that avoids the problem of malformed UTF characters is to use a lookahead to check the length of the next character, as in this pattern, which could be used with a UTF-8 string (ignore white space and line breaks):

```regex
(?| (?=[\x00-\x7f])(\C) | (?=[\x80-\xff])(\C)(\C) | (?=[\x{800}-\xff{fff}])\C\C\C | (?=[\x{10000}-\x{fffffff}])\C\C\C\C)
```

A group that starts with `(?|` resets the capturing parentheses numbers in each alternative (see "Duplicate Subpattern Numbers" below). The assertions at the start of each branch check the next UTF-8 character for values whose encoding uses 1, 2, 3, or 4 bytes, respectively. The character's individual bytes are then captured by the appropriate number of groups.

**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 by default. However, if the `PCRE_JAVASCRIPT_COMPAT` option is set, a lone closing square bracket causes a compile-time error. 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 a UTF mode, the character may be more than one data unit long. 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 (0xffff) can be included in a class as a literal string of data units, 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 a UTF 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 in a UTF mode for characters 128 and above, you must ensure that PCRE is compiled with Unicode property support as well as with UTF 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 PCRE_DOTALL and PCRE_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.

Ranges operate in the collating sequence of character values. They can also be used for characters specified numerically, for example \[000-037\]. Ranges can include any characters that are valid for the current mode.

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 \[[W\-\w\x{wyzyzabc}\] matched caselessly, and in a non-UTF mode, if character tables for a French locale are in use, \[xc8-\xcb\] matches accented E characters in both cases. In UTF modes, PCRE supports the concept of case for characters with values greater than 128 only when it is compiled with Unicode property support.

The character escape sequences \d, \D, \h, \H, \p, \P, \s, \S, \w, \W, and \W may appear in a character class, and add the characters that they match to the class. For example, \[dABCDEF\] matches any hexadecimal digit. In UTF modes, the \ucp option affects the meanings of \d, \s, \w and their upper case partners, just as it does when they appear outside a character class, as described in the section entitled "Generic character types" above. The escape sequence \b has a different meaning inside a character class; it matches the backspace character. The sequences \B, \N, \R, and \X are not special inside a character class. Like any other unrecognized escape sequences, they are treated as the literal characters "B", "N", "R", and "X".

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, whereas \[w\] includes underscore. A positive character class should be read as "something OR something OR ..." and a negative class as "NOT something AND NOT something AND NOT ...".

The only metacharacters that are recognized in character classes are backslash, hyphen (only where it can be interpreted as specifying a range), circumflex (only at the start), opening square bracket (only when it can be interpreted as
introducing a POSIX class name - see the next section), and the terminating closing square bracket. However, escaping other non-alphanumeric characters does no harm.

**POSIX character classes**

Perl supports the POSIX notation for character classes. This uses names enclosed by [: and :] within the enclosing square brackets. PCRE also supports this notation. For example,

```
[01[:alpha:]:%]
```

matches "0", "1", any alphabetic character, or "%". The supported class names are:

- **alnum**: letters and digits
- **alpha**: letters
- **ascii**: character codes 0 - 127
- **blank**: space or tab only
- **cntrl**: control characters
- **digit**: decimal digits (same as \d)
- **graph**: printing characters, excluding space
- **lower**: lower case letters
- **print**: printing characters, including space
- **punct**: printing characters, excluding letters and digits and space
- **space**: whitespace (not quite the same as \s)
- **upper**: upper case letters
- **word**: "word" characters (same as \w)
- **xdigit**: hexadecimal digits

The "space" characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). Notice that this list includes the VT character (code 11). This makes "space" different to \s, which does not include VT (for Perl compatibility).

The name "word" is a Perl extension, and "blank" is a GNU extension from Perl 5.8. Another Perl extension is negation, which is indicated by a ^ character after the colon. For example,

```
[12[^:digit:]]
```

matches "1", "2", or any non-digit. PCRE (and Perl) also recognize the POSIX syntax [.ch.] and [=ch=] where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

By default, in UTF modes, characters with values greater than 128 do not match any of the POSIX character classes. However, if the PCRE_UCP option is passed to `pcre_compile()`, some of the classes are changed so that Unicode character properties are used. This is achieved by replacing the POSIX classes by other sequences, as follows:

```
[:alnum:]
```

becomes `\p{Xan}`
Negated versions, such as [:^alpha:] use \P instead of \p. The other POSIX classes are unchanged, and match only characters with code points less than 128.

**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 one of these option changes 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 subpattern 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 compiling or matching functions are called. In some cases the pattern can contain special leading sequences such as (*CRLF) to override what the application has set or what has been defaulted. Details are given in the section entitled "Newline sequences" above. There are also the (*UTF8) and (*UCP) leading sequences that can be used to set UTF and Unicode property modes; they are equivalent to setting the unicode and the ucp options, respectively. The (*UTF) sequence is a generic version that can be used with any of the libraries. However, the application can set the never_utf option, which locks out the use of the (*UTF) sequences.

Subpatterns

Subpatterns are delimited by parentheses (round brackets), which can be nested. Turning part of a pattern into a subpattern does two things:

1. It localizes a set of alternatives. For example, the pattern
   cat(aract|erpillar)

   matches "cataact", "caterpillar", or "cat". Without the parentheses, it would match "cataact", "erpillar" or an empty string.

2. It sets up the subpattern as a capturing subpattern. This means that, when the complete pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller via the return value of re:run/3.

Opening parentheses are counted from left to right (starting from 1) to obtain numbers for the capturing subpatterns. For example, if the string "the red king" is matched against the pattern

the ((red|white) (king|queen))

the captured substrings are "red king", "red", and "king", and are numbered 1, 2, and 3, respectively.

The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by a question mark and a colon, the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string "the white queen" is matched against the pattern

the ((?:red|white) (king|queen))

the captured substrings are "white queen" and "queen", and are numbered 1 and 2. The maximum number of capturing subpatterns is 65535.

As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the "(?" and the ":"). Thus the two patterns

- (?i:saturday|sunday)
- (?:(?i)saturday|sunday)
match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

**Duplicate subpattern numbers**

Perl 5.10 introduced a feature whereby each alternative in a subpattern uses the same numbers for its capturing parentheses. Such a subpattern starts with (\?) and is itself a non-capturing subpattern. For example, consider this pattern:

(\?)(Sat)ur((Sun))day

Because the two alternatives are inside a (\?) group, both sets of capturing parentheses are numbered one. Thus, when the pattern matches, you can look at captured substring number one, whichever alternative matched. This construct is useful when you want to capture part, but not all, of one of a number of alternatives. Inside a (\?) group, parentheses are numbered as usual, but the number is reset at the start of each branch. The numbers of any capturing parentheses 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.

<table>
<thead>
<tr>
<th># before</th>
<th>------------------</th>
<th>branch reset</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ ( a )</td>
<td>(?</td>
<td>x ( y ) z )</td>
<td>( p ( q ) r )</td>
</tr>
<tr>
<td># 1</td>
<td>2</td>
<td>3</td>
<td>2 3 4</td>
</tr>
</tbody>
</table>

A back reference to a numbered subpattern uses the most recent value that is set for that number by any subpattern. The following pattern matches "abcabc" or "defdef":

/\(?|(abc)|(def))\1/\1/

In contrast, a subroutine call to a numbered subpattern always refers to the first one in the pattern with the given number. The following pattern matches "abcabc" or "defabc":

/\(?|(abc)|(def))(\?1)/

If a condition test for a subpattern's having matched refers to a non-unique number, the test is true if any of the subpatterns of that number have matched.

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. Perl allows identically numbered subpatterns to have different names, but PCRE does not.

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 back references, 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. (Duplicate names are also always permitted for subpatterns with the same number, set up as described in the previous section.) Duplicate names can be useful for patterns where only one instance of the named
parentheses can match. Suppose you want to match the name of a weekday, either as a 3-letter abbreviation or as the full name, and in both cases you want to extract the abbreviation. This pattern (ignoring the line breaks) does the job:

```
(?><DN>Mon|Fri|Sun)(?:day)?|
(?><DN>Tue)(?:sday)?|
(?><DN>Wed)(?:nesday)?|
(?><DN>Thu)(?:rsday)?|
(?><DN>Sat)(?:urday)?
```

There are five capturing substrings, but only one is ever set after a match. (An alternative way of solving this problem is to use a "branch reset" subpattern, as described in the previous section.)

In case of capturing named subpatterns which names are not unique, the first matching occurrence (counted from left to right in the subject) is returned from `re:exec/3`, if the name is specified in the `values` part of the `capture` statement. The `all_names` capturing value will match all of the names in the same way.

**Warning:** You cannot use different names to distinguish between two subpatterns with the same number because PCRE uses only the numbers when matching. For this reason, an error is given at compile time if different names are given to subpatterns with the same number. However, you can give the same name to subpatterns with the same number, even when `dupnames` is not set.

**Repetition**

Repetition is specified by quantifiers, which can follow any of the following items:

- a literal data character
- the dot metacharacter
- the \C escape sequence
- the \X escape sequence
- the \R escape sequence
- an escape such as \d or \pL that matches a single character
- a character class
- a back reference (see next section)
- a parenthesized subpattern (including assertions)
- a subroutine call to a subpattern (recursive or otherwise)

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 Unicode mode, quantifiers apply to characters rather than to individual data units. Thus, for example, \x{100}{2} matches two characters, each of which is represented by a two-byte sequence in a UTF-8 string. Similarly, \X{3} matches three Unicode extended grapheme clusters, each of which may be several data units 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. This may be useful for subpatterns that are referenced as subroutines from elsewhere in the pattern (but see also the section entitled "Defining subpatterns for use by reference only" below). Items other than subpatterns that have a \{0\} quantifier are omitted from the compiled pattern.

For convenience, the three most common quantifiers have single-character abbreviations:

*  
is equivalent to \{0,\}
+

is equivalent to \{1,\}
?

is equivalent to \{0,1\}

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

(a?)*

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between /* and */ and within the comment, individual * and / characters may appear. An attempt to match C comments by applying the pattern

/\*.*\*/

to the string

/* first comment */ not comment /* second comment */

fails, because it matches the entire string owing to the greediness of the .* item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

/\*.*?\*/

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

\d??\d

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the ungreedy option is set (an option that is not available in Perl), the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more memory is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with .* or \{0,\} and the dotall option (equivalent to Perl's \s) is set, thus allowing the dot to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position.
in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE normally
treats such a pattern as though it were preceded by \A.

In cases where it is known that the subject string contains no newlines, it is worth setting `dotall` in order to obtain
this optimization, or alternatively using `^` to indicate anchoring explicitly.

However, there are some cases where the optimization cannot be used. When `^` is inside capturing parentheses that
are the subject of a back reference elsewhere in the pattern, a match at the start may fail where a later one succeeds.
Consider, for example:

```perl
(.*)abc\1
```

If the subject is "xyz123abc123" the match point is the fourth character. For this reason, such a pattern is not implicitly
anchored.

Another case where implicit anchoring is not applied is when the leading `.*` is inside an atomic group. Once again, a
match at the start may fail where a later one succeeds. Consider this pattern:

```perl
(?>.*?a)b
```

It matches "ab" in the subject "aab". The use of the backtracking control verbs (`*PRUNE`) and (`*SKIP`) also disable
this optimization.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For
example, after

```perl
(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

```perl
/(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:

```perl
(?>\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

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

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. 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, it is equivalent to \2 in this example. Similarly, \g{-2} would be equivalent to \1. The use of relative references can be helpful in long patterns, and also in patterns that are created by joining together fragments that contain references within themselves.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (see "Subpatterns as subroutines" below for a way of doing that). So the pattern

(sens|respons)e and \1ibility

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If caseful matching is in force at the time of the back reference, the case of letters is relevant. For example,

((?i)rah)s+\1

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

There are several different ways of writing back references to named subpatterns. The .NET syntax \k{name} and the Perl syntax \k<name> or \k{name} are supported, as is the Python syntax (?P=name). Perl 5.10’s unified back reference syntax, in which \g can be used for both numeric and named references, is also supported. We could rewrite the above example in any of the following ways:

- (?<p1>(?i)rah)s+\k<p1>
- (?p1(?i)rah)s+\k{p1}
- (?P<p1>(?i)rah)s+(?P=p1)
- (?<p1>(?i)rah)s+\g{p1}

A subpattern that is referenced by name may appear in the pattern before or after the reference.

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern

(a(bc)de)\2

always fails if it starts to match "a" rather than "bc". Because there may be many capturing parentheses in a pattern, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the extended option is set, this can be whitespace. Otherwise an empty comment (see "Comments" below) can be used.

Recursive back references

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern

(a|b\1)+

matches any number of "a"s and also "aba", "ababbaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such
that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

Back references of this type cause the group that they reference to be treated as an atomic group. Once the whole group has been matched, a subsequent matching failure cannot cause backtracking into the middle of the group.

**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. If such an 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. (Perl sometimes, but not always, does do capturing in negative assertions.)

For compatibility with Perl, assertion subpatterns may be repeated; though it makes no sense to assert the same thing several times, the side effect of capturing parentheses may occasionally be useful. In practice, there only three cases:

1. If the quantifier is \{0\}, the assertion is never obeyed during matching. However, it may contain internal capturing parenthesized groups that are called from elsewhere via the subroutine mechanism.

2. If quantifier is \{0,n\} where n is greater than zero, it is treated as if it were \{0,1\}. At run time, the rest of the pattern match is tried with and without the assertion, the order depending on the greediness of the quantifier.

3. If the minimum repetition is greater than zero, the quantifier is ignored. The assertion is obeyed just once when encountered during matching.

**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. The backtracking control verb (*FAIL) or (*F) is a synonym for (?!).

**Lookbehind assertions**

Lookbehind assertions start with (?<= for positive assertions and (?! for negative assertions. For example,

(?<foo)bar

does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several top-level alternatives, they do not all have to have the same fixed length. Thus
(?<=bullock|donkey)
is permitted, but
(?<!dogs?|cats?)
causes an error at compile time. Branches that match different length strings are permitted only at the top level of a
lookbehind assertion. This is an extension compared with Perl, which requires all branches to match the same length
of string. An assertion such as
(?<=ab(c|de))
is not permitted, because its single top-level branch can match two different lengths, but it is acceptable to PCRE if
rewritten to use two top-level branches:
(?<=abc|abde)
In some cases, the escape sequence \K (see above) can be used instead of a lookbehind assertion to get round the
fixed-length restriction.
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.
In a UTF mode, PCRE does not allow the \C escape (which matches a single data unit even in a UTF 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 data units, are also not permitted.
"Subroutine" calls (see below) such as (?2) or (?&X) are permitted in lookbehinds, as long as the subpattern matches
a fixed-length string. Recursion, however, is not supported.
Possessive quantifiers can be used in conjunction with lookbehind assertions to specify efficient matching of fixed-
length strings at the end of subject strings. Consider a simple pattern such as
abcd$
when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for
each "a" in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as
^.*abcd$
the initial .* matches the entire string at first, but when this fails (because there is no following "a"), it backtracks to
match all but the last character, then all but the last two characters, and so on. Once again the search for "a" covers the
entire string, from right to left, so we are no better off. However, if the pattern is written as
^.*+(?=abcd)
there can be no backtracking for the .*+ item; it can match only the entire string. The subsequent lookbehind assertion
does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach
makes a significant difference to the processing time.

Using multiple assertions
Several assertions (of any sort) may occur in succession. For example,
(?<=[d\d]{3})(?<!999)foo
matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at
the same point in the subject string. First there is a check that the previous three characters are all digits, and then there
is a check that the same three characters are not "999". This pattern does not match "foo" preceded by six characters,
the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abcfoo". A
pattern to do that is
(?<=[d\d]{3}...)(?<!999)foo
This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example,

(?!<=(?<!foo)bar)baz

matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while

(?!<=\d[3](?!999)...)foo

is another pattern that matches "foo" preceded by three digits and any three characters that are not "999".

**Conditional subpatterns**

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a specific capturing subpattern has already been matched. 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. Each of the two alternatives may itself contain nested subpatterns of any form, including conditional subpatterns; the restriction to two alternatives applies only at the level of the condition. This pattern fragment is an example where the alternatives are complex:

(?(1) (A|B|C) | (D | (?(2)E|F) | E) )

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 a capturing subpattern of that number has previously matched. If there is more than one capturing subpattern with the same number (see the earlier section about duplicate subpattern numbers), the condition is true if any of them have 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. Inside loops it can also make sense to refer to subsequent groups. The next parentheses to be opened can be referenced as (?(+1), and so on. (The value zero in any of these forms is not used; it provokes a compile-time error.)

Consider the following pattern, which contains non-significant whitespace to make it more readable (assume the extended option) and to divide it into three parts for ease of discussion:

( \( )? \[^()]+ (?(1) \) )

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether or not 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:

```regex
(<!OPEN> \( )? \[^()]+ (\?(<OPEN>) )
```

If the name used in a condition of this kind is a duplicate, the test is applied to all subpatterns of the same name, and is true if any one of them has matched.

**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:

```regex
(?(R3)...) or (?(R&name)...
```

the condition is true if the most recent recursion is into a subpattern whose number or name is given. This condition does not check the entire recursion stack. If the name used in a condition of this kind is a duplicate, the test is applied to all subpatterns of the same name, and is true if any one of them is the most recent recursion.

At "top level", all these recursion test conditions are false. The syntax for recursive patterns is 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 such as "192.168.23.245" could be written like this (ignore whitespace and line breaks):

```regex
(?<DEFINE> (?<byte> 2\[0-4]\d | 25\[0-5] | 1\d\d | \[1-9\]?\d) ) \b (?&byte) \b (?&byte)(\.|(?&byte))\{3} \b
```

The first part of the pattern is a DEFINE group inside which a another group named "byte" is defined. This matches an individual component of an IPv4 address (a number less than 256). When matching takes place, this part of the pattern is skipped because DEFINE acts like a false condition. The rest of the pattern uses references to the named group to match the four dot-separated components of an IPv4 address, insisting on a word boundary at each end.

**Assertion conditions**

If the condition is not in any of the above formats, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant whitespace, and with the two alternatives on the second line:

```regex
(?(?=\[^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

There are two ways of including comments in patterns that are processed by PCRE. In both cases, the start of the comment must not be in a character class, nor in the middle of any other sequence of related characters such as (?: or a subpattern name or number. The characters that make up a comment play no part in the pattern matching.

The sequence (?# marks the start of a comment that continues up to the next closing parenthesis. Nested parentheses are not permitted. If the PCRE_EXTENDED option is set, an unescaped # character also introduces a comment, which in this case continues to immediately after the next newline character or character sequence in the pattern. Which characters are interpreted as newlines is controlled by the options passed to a compiling function or by a special sequence at the start of the pattern, as described in the section entitled "Newline conventions" above. Note that the end of this type of comment is a literal newline sequence in the pattern; escape sequences that happen to represent a newline do not count. For example, consider this pattern when extended is set, and the default newline convention is in force:

```
abc #comment \n still comment
```

On encountering the # character, `pcre_compile()` skips along, looking for a newline in the pattern. The sequence `\n` is still literal at this stage, so it does not terminate the comment. Only an actual character with the code value 0x0a (the default newline) does so.

Recursive patterns

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth.

For some time, Perl has provided a facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at run time, and the code can refer to the expression itself. A Perl pattern using code interpolation to solve the parentheses problem can be created like this:

```
$re = qr\{ (?: (?>[^()]+) | (?p{$re}) )* \}x;
```

The (?p{...}) item interpolates Perl code at run time, and in this case refers recursively to the pattern in which it appears. Obviously, PCRE cannot support the interpolation of Perl code. Instead, it supports special syntax for recursion of the entire pattern, and also for individual subpattern recursion. After its introduction in PCRE and Python, this kind of recursion was subsequently 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 subroutine call of the subpattern of the given number, provided that it occurs inside that subpattern. (If not, it is a non-recursive subroutine call, which is described in the next section.) The special item (?R) or (?0) is a recursive call of the entire regular expression.

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. Note the use of a possessive quantifier to avoid backtracking into sequences of non-parentheses. If this were part of a larger pattern, you would not want to recurse the entire pattern, so instead you could use this:

```
\( ( [^0]+ | (?1) )* \) )
```

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. Instead of (?1) in the pattern above you can write (?-2) to refer to the second most recently opened parentheses.
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 non-
recursive 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 a
possessive quantifier 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

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa())
```

it yields "no match" quickly. However, if a possessive quantifier 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 of capturing parentheses are those from the outermost level. If the pattern above
is matched against

```
(ab(cd)ef)
```

the value for the inner capturing parentheses (numbered 2) is "ef", which is the last value taken on at the top level.

If a capturing subpattern is not matched at the top level, its final captured value is unset, even if it was (temporarily)
set at a deeper level during the matching process.

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.

**Differences in recursion processing between PCRE and Perl**

Recursion processing in PCRE differs from Perl in two important ways. 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 untired alternatives and there is a subsequent matching failure. This can be
illustrated by the following pattern, which purports to match a palindromic string that contains an odd number of
characters (for example, "a", "aba", "abcba", "abcdcba"):

```
^\.(\.(?!1)2)$
```

The idea is that it either matches a single character, or two identical characters surrounding a sub-palindrome. In Perl,
this pattern works; in PCRE it does not if the pattern is longer than three characters. Consider the subject string "abcb":

At the top level, the first character is matched, but as it is not at the end of the string, the first alternative fails; the
second alternative is taken and the recursion kicks in. The recursive call to subpattern 1 successfully matches the next
character ("b"). (Note that the beginning and end of line tests are not part of the recursion).

Back at the top level, the next character ("c") is compared with what subpattern 2 matched, which was "a". This fails.
Because the recursion is treated as an atomic group, there are now no backtracking points, and so the entire match
fails. (Perl is able, at this point, to re-enter the recursion and try the second alternative.) However, if the pattern is
written with the alternatives in the other order, things are different:
This time, the recursing alternative is tried first, and continues to recurse until it runs out of characters, at which point
the recursion fails. But this time we do have another alternative to try at the higher level. That is the big difference: in
the previous case the remaining alternative is at a deeper recursion level, which PCRE cannot use.
To change the pattern so that it matches all palindromic strings, not just those with an odd number of characters, it
is tempting to change the pattern to this:

^((.(?1)|).)$

Again, this works in Perl, but not in PCRE, and for the same reason. When a deeper recursion has matched a single
character, it cannot be entered again in order to match an empty string. The solution is to separate the two cases, and
write out the odd and even cases as alternatives at the higher level:

^\?((.(?1)|).)((.?)(?3)|.))$

If you want to match typical palindromic phrases, the pattern has to ignore all non-word characters, which can be
done like this:

^\W*+(\?((.)(?1)$\W*+(?1)$\W*+(?3)$\W*+(?4)$\W*+$)

If run with the caseless option, this pattern matches phrases such as "A man, a plan, a canal: Panama!" and it works
well in both PCRE and Perl. Note the use of the possessive quantifier *+ to avoid backtracking into sequences of non-
word characters. Without this, PCRE takes a great deal longer (ten times or more) to match typical phrases, and Perl
takes so long that you think it has gone into a loop.

**WARNING**: The palindrome-matching patterns above work only if the subject string does not start with a palindrome
that is shorter than the entire string. For example, although "abcba" is correctly matched, if the subject is "ababa",
PCRE finds the palindrome "aba" at the start, then fails at top level because the end of the string does not follow. Once
again, it cannot jump back into the recursion to try other alternatives, so the entire match fails.

The second way in which PCRE and Perl differ in their recursion processing is in the handling of captured values. In
Perl, when a subpattern is called recursively or as a subpattern (see the next section), it has no access to any values
that were captured outside the recursion, whereas in PCRE these values can be referenced. Consider this pattern:

^(.)(\1|a(?2))$

In PCRE, this pattern matches "bab". The first capturing parentheses match "b", then in the second group, when the
back reference \1 fails to match "b", the second alternative matches "a" and then recurses. In the recursion, \1 does
now match "b" and so the whole match succeeds. In Perl, the pattern fails to match because inside the recursive call
\1 cannot access the externally set value.

### Subpatterns as subroutines

If the syntax for a recursive subpattern call (either by number or by name) is used outside the parentheses to which it
refers, it operates like a subroutine in a programming language. The called subpattern may be defined before or after
the reference. A numbered reference can be absolute or relative, as in these examples:

- (...(absolute)...(?2)...)
- (...(relative)...(?-1)...)
- (...(?+1)...(relative)...

An earlier example pointed out that the pattern

(sens|respons)e and \1ibility

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If instead the pattern

(sens|respons)e and (?1)ibility
is used, it does match "sense and responsibility" as well as the other two strings. Another example is given in the discussion of DEFINE above.

All subroutine calls, whether recursive or not, are always treated as atomic groups. That is, once a subroutine 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. Any capturing parentheses that are set during the subroutine call revert to their previous values afterwards.

Processing options such as case-independence are fixed when a subpattern is defined, so if it is used as a subroutine, such options cannot be changed for different calls. For example, consider this pattern:

(abc)(?i:(?-.1))

It matches "abcabc". It does not match "abcABC" because the change of processing option does not affect the called subpattern.

Oniguruma subroutine syntax

For compatibility with Oniguruma, the non-Perl syntax \g followed by a name or a number enclosed either in angle brackets or single quotes, is an alternative syntax for referencing a subpattern as a subroutine, possibly recursively. Here are two of the examples used above, rewritten using this syntax:

(?<pn> \((?=.*[^()]+)|\g<pn> \)* \))

(sens|respons)e and \g'1'ibility

PCRE supports an extension to Oniguruma: if a number is preceded by a plus or a minus sign it is taken as a relative reference. For example:

(abc)(?i:\g<-1>)

Note that \g{...} (Perl syntax) and \g<...> (Oniguruma syntax) are not synonymous. The former is a back reference; the latter is a subroutine call.

Backtracking control

Perl 5.10 introduced a number of "Special Backtracking Control Verbs", which are still 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. They are generally of the form (*VERB) or (*VERB:NAME). Some may take either form, possibly behaving differently depending on whether or not a name is present. A name is any sequence of characters that does not include a closing parenthesis. The maximum length of name is 255 in the 8-bit library and 65535 in the 16-bit and 32-bit libraries. If the name is empty, that is, if the closing parenthesis immediately follows the colon, the effect is as if the colon were not there. Any number of these verbs may occur in a pattern.

The behaviour of these verbs in repeated groups, assertions, and in subpatterns called as subroutines (whether or not recursively) is documented below.

Optimizations that affect backtracking verbs

PCRE contains some optimizations that are used to speed up matching by running some checks at the start of each match attempt. For example, it may know the minimum length of matching subject, or that a particular character must be present. When one of these optimizations bypasses the running of a match, any included backtracking verbs will not, of course, be processed. You can suppress the start-of-match optimizations by setting the no_start_optimize option when calling re:compile/2 or re:run/3, or by starting the pattern with (*NO_START_OPT).

Experiments with Perl suggest that it too has similar optimizations, sometimes leading to anomalous results.
**Verbs that act immediately**

The following verbs act as soon as they are encountered. They may not be followed by a name.

(*ACCEPT*)

This verb causes the match to end successfully, skipping the remainder of the pattern. However, when it is inside a subpattern that is called as a subroutine, only that subpattern is ended successfully. Matching then continues at the outer level. If (*ACCEPT*) in triggered in a positive assertion, the assertion succeeds; in a negative assertion, the assertion fails.

If (*ACCEPT*) is inside capturing parentheses, the data so far is captured. For example:

\`\`A((?:A|B(*ACCEPT)|C)D)\`

This matches "AB", "AAD", or "ACD"; when it matches "AB", "B" is captured by the outer parentheses.

(*FAIL*) or (*F*)

This verb causes a matching failure, forcing backtracking to occur. It is equivalent to (?!1) but easier to read. The Perl documentation notes that it is probably useful only when combined with (?{1}) or (?{1}). Those are, of course, Perl features that are not present in PCRE. The nearest equivalent is the callout feature, as for example in this pattern:

\`\`a+(?C)(*FAIL)\`

A match with the string "aaaa" always fails, but the callout is taken before each backtrack happens (in this example, 10 times).

**Recording which path was taken**

There is one verb whose main purpose is to track how a match was arrived at, though it also has a secondary use in conjunction with advancing the match starting point (see (*SKIP*) below).

---

**Warning:**

In Erlang, there is no interface to retrieve a mark with `re:run([2, 3])`, so only the secondary purpose is relevant to the Erlang programmer!

The rest of this section is therefore deliberately not adapted for reading by the Erlang programmer, however the examples might help in understanding NAMES as they can be used by (*SKIP*).

(*MARK:NAME) or (*:NAME)

A name is always required with this verb. There may be as many instances of (*MARK*) as you like in a pattern, and their names do not have to be unique.

When a match succeeds, the name of the last-encountered (*MARK:NAME), (*PRUNE:NAME), or (*THEN:NAME) on the matching path is passed back to the caller as described in the section entitled "Extra data for `pcre_exec()`" in the `pcreapi` documentation. Here is an example of `pcrtest` output, where the /K modifier requests the retrieval and outputting of (*MARK*) data:

```
re> /X(*MARK:A)Y|X(*MARK:B)Z/K
data> XY
  0: XY
  MK: A
  XZ
  0: XZ
  MK: B
```
The (*MARK) name is tagged with "MK:" in this output, and in this example it indicates which of the two alternatives matched. This is a more efficient way of obtaining this information than putting each alternative in its own capturing parentheses.

If a verb with a name is encountered in a positive assertion that is true, the name is recorded and passed back if it is the last-encountered. This does not happen for negative assertions or failing positive assertions.

After a partial match or a failed match, the last encountered name in the entire match process is returned. For example:

```re
re> /X(*MARK:A)Y|X(*MARK:B)Z/K
data> XP
No match, mark = B
```

Note that in this unanchored example the mark is retained from the match attempt that started at the letter "X" in the subject. Subsequent match attempts starting at "P" and then with an empty string do not get as far as the (*MARK) item, but nevertheless do not reset it.

**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, causing a backtrack to the verb, a failure is forced. That is, backtracking cannot pass to the left of the verb. However, when one of these verbs appears inside an atomic group or an assertion that is true, its effect is confined to that group, because once the group has been matched, there is never any backtracking into it. In this situation, backtracking can "jump back" to the left of the entire atomic group or assertion. (Remember also, as stated above, that this localization also applies in subroutine calls.)

These verbs differ in exactly what kind of failure occurs when backtracking reaches them. The behaviour described below is what happens when the verb is not in a subroutine or an assertion. Subsequent sections cover these special cases.

(*COMMIT)

This verb, which may not be followed by a name, causes the whole match to fail outright if there is a later matching failure that causes backtracking to reach it. Even if the pattern is unanchored, no further attempts to find a match by advancing the starting point take place. If (*COMMIT) is the only backtracking verb that is encountered, once it has been passed `re:run/({2,3})` is committed to finding a match at the current starting point, or not at all. For example:

```re
a+(*COMMIT)b
```

This matches "xxaab" but not "aacaab". It can be thought of as a kind of dynamic anchor, or "I've started, so I must finish." The name of the most recently passed (*MARK) in the path is passed back when (*COMMIT) forces a match failure.

If there is more than one backtracking verb in a pattern, a different one that follows (*COMMIT) may be triggered first, so merely passing (*COMMIT) during a match does not always guarantee that a match must be at this starting point. Note that (*COMMIT) at the start of a pattern is not the same as an anchor, unless PCRE's start-of-match optimizations are turned off, as shown in this example:

```re
1> re:run("xyzabc","(*COMMIT)abc",{{capture,all,list}}).
{match,["abc"]}
2> re:run("xyzabc","(*COMMIT)abc",{{capture,all,list},no_start_optimize}).
nomatch
```

PCRE knows that any match must start with "a", so the optimization skips along the subject to "a" before running the first match attempt, which succeeds. When the optimization is disabled by the no_start_optimize option, the match starts at "x" and so the (*COMMIT) causes it to fail without trying any other starting points.
(*PRUNE) or (*PRUNE:NAME)

This verb causes the match to fail at the current starting position in the subject if there is a later matching failure that causes backtracking to reach it. 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), before it is reached, 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. In an anchored pattern (*PRUNE) has the same effect as (*COMMIT).

The behaviour of (*PRUNE:NAME) is not the same as (*MARK:NAME)(*PRUNE). It is like (*MARK:NAME) in that the name is remembered for passing back to the caller. However, (*SKIP:NAME) searches only for names set with (*MARK).

### Warning:

The fact that (*PRUNE:NAME) remembers the name is useless to the Erlang programmer, as names can not be retrieved.

(*SKIP)

This verb, when given without a name, is like (*PRUNE), except that if the pattern is unanchored, the "bumpalong" advance is not to the next character, but to the position in the subject where (*SKIP) was encountered. (*SKIP) signifies that whatever text was matched leading up to it cannot be part of a successful match. Consider:

```
a++(*SKIP)b
```

If the subject is "aaaac...", after the first match attempt fails (starting at the first character in the string), the starting point skips on to start the next attempt at "c". Note that a possessive quantifier does not have the same effect as 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".

(*SKIP:NAME)

When (*SKIP) has an associated name, its behaviour is modified. When it is triggered, the previous path through the pattern is searched for the most recent (*MARK) that has the same name. If one is found, the "bumpalong" advance is to the subject position that corresponds to that (*MARK) instead of to where (*SKIP) was encountered. If no (*MARK) with a matching name is found, the (*SKIP) is ignored.

Note that (*SKIP:NAME) searches only for names set by (*MARK:NAME). It ignores names that are set by (*PRUNE:NAME) or (*THEN:NAME).

(*THEN) or (*THEN:NAME)

This verb causes a skip to the next innermost alternative when backtracking reaches it. That is, it cancels any further backtracking within the current alternative. Its name comes from the observation that it can be used for a pattern-based if-then-else block:

```
( COND1 (*THEN) FOO | COND2 (*THEN) BAR | COND3 (*THEN) BAZ ) ...
```

If the COND1 pattern matches, FOO is tried (and possibly further items after the end of the group if FOO succeeds); on failure, the matcher skips to the second alternative and tries COND2, without backtracking into COND1. If that succeeds and BAR fails, COND3 is tried. If subsequently BAZ fails, there are no more alternatives, so there is a backtrack to whatever came before the entire group. If (*THEN) is not inside an alternation, it acts like (*PRUNE).

The behaviour of (*THEN:NAME) is the not the same as (*MARK:NAME)(*THEN). It is like (*MARK:NAME) in that the name is remembered for passing back to the caller. However, (*SKIP:NAME) searches only for names set with (*MARK).
Warning:
The fact that (*THEN:NAME) remembers the name is useless to the Erlang programmer, as names can not be retrieved.

A subpattern that does not contain a | character is just a part of the enclosing alternative; it is not a nested alternation with only one alternative. The effect of (*THEN) extends beyond such a subpattern to the enclosing alternative. Consider this pattern, where A, B, etc. are complex pattern fragments that do not contain any | characters at this level:

A (B(*THEN)C) | D

If A and B are matched, but there is a failure in C, matching does not backtrack into A; instead it moves to the next alternative, that is, D. However, if the subpattern containing (*THEN) is given an alternative, it behaves differently:

A (B(*THEN)C | (*FAIL)) | D

The effect of (*THEN) is now confined to the inner subpattern. After a failure in C, matching moves to (*FAIL), which causes the whole subpattern to fail because there are no more alternatives to try. In this case, matching does now backtrack into A.

Note that a conditional subpattern is not considered as having two alternatives, because only one is ever used. In other words, the | character in a conditional subpattern has a different meaning. Ignoring white space, consider:

^.*? (?(?=a) a | b(*THEN)c )

If the subject is "ba", this pattern does not match. Because .*? is ungreedy, it initially matches zero characters. The condition (?=a) then fails, the character "b" is matched, but "c" is not. At this point, matching does not backtrack to .*? as might perhaps be expected from the presence of the | character. The conditional subpattern is part of the single alternative that comprises the whole pattern, and so the match fails. (If there was a backtrack into .*?, allowing it to match "b", the match would succeed.)

The verbs just described provide four different "strengths" of control when subsequent matching fails. (*THEN) is the weakest, carrying on the match at the next alternative. (*PRUNE) comes next, failing the match at the current starting position, but allowing an advance to the next character (for an unanchored pattern). (*SKIP) is similar, except that the advance may be more than one character. (*COMMIT) is the strongest, causing the entire match to fail.

More than one backtracking verb

If more than one backtracking verb is present in a pattern, the one that is backtracked onto first acts. For example, consider this pattern, where A, B, etc. are complex pattern fragments:

(A(*COMMIT)B(*THEN)C|ABD)

If A matches but B fails, the backtrack to (*COMMIT) causes the entire match to fail. However, if A and B match, but C fails, the backtrack to (*THEN) causes the next alternative (ABD) to be tried. This behaviour is consistent, but is not always the same as Perl's. It means that if two or more backtracking verbs appear in succession, all the the last of them has no effect. Consider this example:

...((*COMMIT)(*PRUNE)... 

If there is a matching failure to the right, backtracking onto (*PRUNE) cases it to be triggered, and its action is taken. There can never be a backtrack onto (*COMMIT).

Backtracking verbs in repeated groups

PCRE differs from Perl in its handling of backtracking verbs in repeated groups. For example, consider:

/l(at(*COMMIT)b)+ac/

If the subject is "abac", Perl matches, but PCRE fails because the (*COMMIT) in the second repeat of the group acts.

Backtracking verbs in assertions
(*FAIL) in an assertion has its normal effect: it forces an immediate backtrack.

(*ACCEPT) in a positive assertion causes the assertion to succeed without any further processing. In a negative assertion, (*ACCEPT) causes the assertion to fail without any further processing.

The other backtracking verbs are not treated specially if they appear in a positive assertion. In particular, (*THEN) skips to the next alternative in the innermost enclosing group that has alternations, whether or not this is within the assertion.

Negative assertions are, however, different, in order to ensure that changing a positive assertion into a negative assertion changes its result. Backtracking into (*COMMIT), (*SKIP), or (*PRUNE) causes a negative assertion to be true, without considering any further alternative branches in the assertion. Backtracking into (*THEN) causes it to skip to the next enclosing alternative within the assertion (the normal behaviour), but if the assertion does not have such an alternative, (*THEN) behaves like (*PRUNE).

**Backtracking verbs in subroutines**

These behaviours occur whether or not the subpattern is called recursively. Perl's treatment of subroutines is different in some cases.

(*FAIL) in a subpattern called as a subroutine has its normal effect: it forces an immediate backtrack.

(*ACCEPT) in a subpattern called as a subroutine causes the subroutine match to succeed without any further processing. Matching then continues after the subroutine call.

(*COMMIT), (*SKIP), and (*PRUNE) in a subpattern called as a subroutine cause the subroutine match to fail.

(*THEN) skips to the next alternative in the innermost enclosing group within the subpattern that has alternatives. If there is no such group within the subpattern, (*THEN) causes the subroutine match to fail.
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(Element)`

As returned by `new/0`.

`set()`

`set()` is equivalent to `set(term())`.

**Exports**

`new() -> set()`

Returns a new empty set.

`is_set(Set) -> boolean()`

Types:

\[ Set = \text{term()} \]

Returns `true` if `Set` is a set of elements, otherwise `false`.

`size(Set) -> integer() \geq 0`

Types:

\[ Set = \text{set()} \]

Returns the number of elements in `Set`.

`to_list(Set) -> List`

Types:

\[ Set = \text{set}(\text{Element}) \]
\[ List = [\text{Element}] \]

Returns the elements of `Set` as a list.

`from_list(List) -> Set`

Types:

\[ List = [\text{Element}] \]
\[ Set = \text{set}(\text{Element}) \]

Returns an set of the elements in `List`.
is_element(Element, Set) -> boolean()
Types:
  Set = set(Element)
Returns true if Element is an element of Set, otherwise false.

add_element(Element, Set1) -> Set2
Types:
  Set1 = Set2 = set(Element)
Returns a new set formed from Set1 with Element inserted.

del_element(Element, Set1) -> Set2
Types:
  Set1 = Set2 = set(Element)
Returns Set1, but with Element removed.

union(Set1, Set2) -> Set3
Types:
  Set1 = Set2 = Set3 = set(Element)
Returns the merged (union) set of Set1 and Set2.

union(SetList) -> Set
Types:
  SetList = [set(Element)]
  Set = set(Element)
Returns the merged (union) set of the list of sets.

intersection(Set1, Set2) -> Set3
Types:
  Set1 = Set2 = Set3 = set(Element)
Returns the intersection of Set1 and Set2.

intersection(SetList) -> Set
Types:
  SetList = [set(Element), ...]
  Set = set(Element)
Returns the intersection of the non-empty list of sets.

is_disjoint(Set1, Set2) -> boolean()
Types:
  Set1 = Set2 = set(Element)
Returns true if Set1 and Set2 are disjoint (have no elements in common), and false otherwise.
subtract(Set1, Set2) -> Set3
Types:
    Set1 = Set2 = Set3 = set(Element)
Returns only the elements of Set1 which are not also elements of Set2.

is_subset(Set1, Set2) -> boolean()
Types:
    Set1 = Set2 = set(Element)
Returns true when every element of Set1 is also a member of Set2, otherwise false.

fold(Function, Acc0, Set) -> Acc1
Types:
    Function = fun((Element, AccIn) -> AccOut)
    Set = set(Element)
    Acc0 = Acc1 = AccIn = AccOut = Acc
Fold Function over every element in Set returning the final value of the accumulator.

filter(Pred, Set1) -> Set2
Types:
    Pred = fun((Element) -> boolean())
    Set1 = Set2 = set(Element)
Filter elements in Set1 with boolean function Pred.

See Also
ordsets(3), gb_sets(3)
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 \texttt{shell:history/1} and \texttt{shell:results/1}, or by setting the application configuration parameters \texttt{shell_history_length} and \texttt{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 \texttt{shell:catch_exception/1} or by setting the application configuration parameter \texttt{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 \texttt{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 \texttt{shell_default}. The module \texttt{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 \texttt{shell_default} and \texttt{user_default} (if loaded) are read each time a new job is started. For instance, adding the line

\begin{verbatim}
-module(user_default).
-include_lib("kernel/include/file.hrl").
\end{verbatim}

to \texttt{user_default} makes the definition of \texttt{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

\texttt{b()}

Prints the current variable bindings.
f()  
Removes all variable bindings.

f(X)  
Removes the binding of variable X.

h()  
Prints the history list.

history(N)  
Sets the number of previous commands to keep in the history list to N. The previous number is returned. The default number is 20.

results(N)  
Sets the number of results from previous commands to keep in the history list to N. The previous number is returned. The default number is 20.

e(N)  
Repeats the command N, if N is positive. If it is negative, the Nth previous command is repeated (i.e., e(-1) repeats the previous command).

v(N)  
Uses the return value of the command N in the current command, if N is positive. If it is negative, the return value of the Nth previous command is used (i.e., v(-1) uses the value of the previous command).

help()  
Evaluates shell_default:help().

c(File)  
Evaluates shell_default:c(File). This compiles and loads code in File and purges old versions of code, if necessary. Assumes that the file and module names are the same.

catch_exception(Bool)  
Sets the exception handling of the evaluator process. The previous exception handling is returned. The default (false) is to kill the evaluator process when an exception occurs, which causes the shell to create a new evaluator process. When the exception handling is set to true the evaluator process lives on which means that for instance ports and ETS tables as well as processes linked to the evaluator process survive the exception.

rd(RecordName, RecordDefinition)  
Defines a record in the shell. RecordName is an atom and RecordDefinition lists the field names and the default values. Usually record definitions are made known to the shell by use of the rr commands described below, but sometimes it is handy to define records on the fly.

rf()  
Removes all record definitions, then reads record definitions from the modules shell_default and user_default (if loaded). Returns the names of the records defined.

rf(RecordNames)  
Removes selected record definitions. RecordNames is a record name or a list of record names. Use '_' to remove all record definitions.

rl()  
Prints all record definitions.
**rl(RecordNames)**

Prints selected record definitions. *RecordNames* is a record name or a list of record names.

**rp(Term)**

Prints a term using the record definitions known to the shell. All of *Term* is printed; the depth is not limited as is the case when a return value is printed.

**rr(Module)**

Reads record definitions from a module's BEAM file. If there are no record definitions in the BEAM file, the source file is located and read instead. Returns the names of the record definitions read. *Module* is an atom.

**rr(Wildcard)**

Reads record definitions from files. Existing definitions of any of the record names read are replaced. *Wildcard* is a wildcard string as defined in `filelib(3)` but not an atom.

**rr(WildcardOrModule, RecordNames)**

Reads record definitions from files but discards record names not mentioned in *RecordNames* (a record name or a list of record names).

**rr(WildcardOrModule, RecordNames, Options)**

Reads record definitions from files. The compiler options `{i, Dir}`, `{d, Macro}`, and `{d, Macro, Value}` are recognized and used for setting up the include path and macro definitions. Use `'_` as value of *RecordNames* to read all record definitions.

### Example

The following example is a long dialogue with the shell. Commands starting with `>` are inputs to the shell. All other lines are output from the shell. All commands in this example are explained at the end of the dialogue.

```erlang
strider 1> erl
Erlang (BEAM) emulator version 5.3 [hipe] [threads:0]
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> {_, _} = Descriptor.
{4,abcd}
10> L.
```

---

*Ericsson AB. All Rights Reserved.: STDLIB | 391*
4
11> {P, Q, R} = Descriptor.
   ** exception error: no match of right hand side value {4,abcd}
12> P.
   * 1: variable 'P' is unbound **
13> Descriptor.
   {4,abcd}
14>{P, Q} = Descriptor.
   {4,abcd}
15> P.
   4
16> f().
   ok
17> put(aa, hello).
   undefined
18> get(aa).
   hello
19> Y = test1:demo(1).
   11
20> get().
   [{aa,worked}]
21> put(aa, hello).
   worked
22> Z = test1:demo(2).
   ** exception error: no match of right hand side value 1
      in function  test1:demo/1
23> Z.
   * 1: variable 'Z' is unbound **
24> get(aa).
   hello
25> erase(), put(aa, hello).
   undefined
26> spawn(test1, demo, [1]).
   <0.57.0>
27> get(aa).
   hello
28> io:format("hello hello\n").
   hello hello
29> e(28).
   hello hello
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
37> rd(rec, {f = orddict:new()}).
   rec
38> #rec{}.
   #rec(f = [])
   ok
39> rd(rec, {c}), A.
   * 1: variable 'A' is unbound **
40> #rec{}.  
#rec{c = undefined}  
ok  
41> test1:loop(0).  
Hello Number: 0  
Hello Number: 1  
Hello Number: 2  
Hello Number: 3  

User switch command  
  --> i  
  --> c  
  .  
  .  
Hello Number: 3374  
Hello Number: 3375  
Hello Number: 3376  
Hello Number: 3377  
Hello Number: 3378  
** exception exit: killed  
42> E = ets:new(t, []).  
17  
43> ets:insert({d,1,2}).  
** exception error: undefined function ets:insert/1  
44> ets:insert(E, {d,1,2}).  
** exception error: argument is of wrong type  
  in function ets:insert/2  
  called as ets:insert(16,{d,1,2})  
45> f(E).  
ok  
46> catch_exception(true).  
false  
47> E = ets:new(t, []).  
18  
48> ets:insert({d,1,2}).  
* exception error: undefined function ets:insert/1  
49> ets:insert(E, {d,1,2}).  
true  
50> halt().  
strider 2>

Comments
Command 1 sets the variable Str to the string "abcd".  
Command 2 sets L to the length of the string evaluating the BIF atom_to_list.  
Command 3 builds the tuple Descriptor.  
Command 4 prints the value of the variable L.  
Command 5 evaluates the internal shell command b (), which is an abbreviation of "bindings". This prints the current shell variables and their bindings. The ok at the end is the return value of the b () function.  
Command 6 f (L) evaluates the internal shell command f (L) (abbreviation of "forget"). The value of the variable L is removed.  
Command 7 prints the new bindings.  
Command 8 has no effect since L has no value.  
Command 9 performs a pattern matching operation on Descriptor, binding a new value to L.  
Command 10 prints the current value of L.
Command 11 tries to match \((P, Q, R)\) against Descriptor which is \(\{4, \text{abc}\}\). 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:

```erlang
demo(X) ->
    put(aa, worked),
    X = 1,
    X + 10.
```

Commands 17 and 18 set and inspect the value of the item aa in the process dictionary.

Command 19 evaluates test1:demo(1). The evaluation succeeds and the changes made in the process dictionary become visible to the shell. The new value of the dictionary item aa can be seen in command 20.

Commands 21 and 22 change the value of the dictionary item aa to hello and call test1:demo(2). Evaluation fails and the changes made to the dictionary in test1:demo(2), before the error occurred, are discarded.

Commands 23 and 24 show that Z was not bound and that the dictionary item aa has retained its original value.

Commands 25, 26 and 27 show the effect of evaluating test1:demo(1) in the background. In this case, the expression is evaluated in a newly spawned process. Any changes made in the process dictionary are local to the newly spawned process and therefore not visible to the shell.

Commands 28, 29 and 30 use the history facilities of the shell.

Command 29 is \(e(28)\). This re-evaluates command 28. Command 30 is \(v(28)\). This uses the value (result) of command 28. In the cases of a pure function (a function with no side effects), the result is the same. For a function with side effects, the result can be different.

The next few commands show some record manipulation. It is assumed that ex.erl defines a record like this:

```erlang
-record(rec, {a, b = val()}).
val() ->
    3.
```

Commands 31 and 32 compiles the file ex.erl and reads the record definitions in ex.beam. If the compiler did not output any record definitions on the BEAM file, rr(ex) tries to read record definitions from the source file instead.

Command 33 prints the definition of the record named rec.

Command 34 tries to create a rec record, but fails since the function val/0 is undefined. Command 35 shows the workaround: explicitly assign values to record fields that cannot otherwise be initialized.

Command 36 prints the newly created record using record definitions maintained by the shell.

Command 37 defines a record directly in the shell. The definition replaces the one read from the file ex.beam.

Command 38 creates a record using the new definition, and prints the result.
Command 39 and 40 show that record definitions are updated as side effects. The evaluation of the command fails but the definition of rec has been carried out.

For the next command, it is assumed that test1:loop(N) is defined in the following way:

```
loop(N) ->
    io:format("Hello Number: ~w~n", [N]),
    loop(N+1).
```

Command 41 evaluates test1:loop(0), which puts the system into an infinite loop. At this point the user types Control G, which suspends output from the current process, which is stuck in a loop, and activates JCL mode. In JCL mode the user can start and stop jobs.

In this particular case, the i command ("interrupt") is used to terminate the looping program, and the c command is used to connect to the shell again. Since the process was running in the background before we killed it, there will be more printouts before the "** exception exit: killed" message is shown.

Command 42 creates an ETS table.

Command 43 tries to insert a tuple into the ETS table but the first argument (the table) is missing. The exception kills the evaluator process.

Command 44 corrects the mistake, but the ETS table has been destroyed since it was owned by the killed evaluator process.

Command 46 sets the exception handling of the evaluator process to true. The exception handling can also be set when starting Erlang, like this: erl -stdlib shell_catch_exception true.

Command 48 makes the same mistake as in command 43, but this time the evaluator process lives on. The single star at the beginning of the printout signals that the exception has been caught.

Command 49 successfully inserts the tuple into the ETS table.

The halt() command exits the Erlang runtime system.

**JCL Mode**

When the shell starts, it starts a single evaluator process. This process, together with any local processes which it spawns, is referred to as a job. Only the current job, which is said to be connected, can perform operations with standard IO. All other jobs, which are said to be detached, are blocked if they attempt to use standard IO.

All jobs which do not use standard IO run in the normal way.

The shell escape key ^G (Control G) detaches the current job and activates JCL mode. The JCL mode prompt is "--">". If "?" is entered at the prompt, the following help message is displayed:

```
-- > ?
c [nn] - connect to job
i [nn] - interrupt job
k [nn] - kill job
j - list all jobs
s [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).

?
Displays this message.

It is possible to alter the behavior of shell escape by means of the STDLIB application variable shell_esc. The value of the variable can be either jcl (erl -stdlib shell_esc jcl) or abort (erl -stdlib shell_esc abort). The first option sets ^G to activate JCL mode (which is also default behavior). The latter sets ^G to terminate the current shell and start a new one. JCL mode cannot be invoked when shell_esc is set to abort.

If you want an Erlang node to have a remote job active from the start (rather than the default local job), you start Erlang with the -remsh flag. Example: erl -sname this_node -remsh other_node@other_host

**Restricted Shell**

The shell may be started in a restricted mode. In this mode, the shell evaluates a function call only if allowed. This feature makes it possible to, for example, prevent a user from accidentally calling a function from the prompt that could harm a running system (useful in combination with the the system flag +Bi).

When the restricted shell evaluates an expression and encounters a function call or an operator application, it calls a callback function (with information about the function call in question). This callback function returns true to let the shell go ahead with the evaluation, or false to abort it. There are two possible callback functions for the user to implement:

\[ \text{local}\_\text{allowed}(	ext{Func}, \text{ArgList}, \text{State}) \rightarrow \{\text{true,NewState}\} \mid \{\text{false,NewState}\} \]


to determine if the call to the local function `Func` with arguments `ArgList` should be allowed.

```
non_local_allowed(FuncSpec, ArgList, State) ->
  (true, NewState) | (false, NewState) | (redirect, NewFuncSpec, NewArgList, NewState)
```

to determine if the call to non-local function `FuncSpec` (``Module,Func``) or a fun) with arguments `ArgList` should be allowed. The return value `(redirect, NewFuncSpec, NewArgList)` can be used to let the shell evaluate some other function than the one specified by `FuncSpec` and `ArgList`.

These callback functions are in fact called from local and non-local evaluation function handlers, described in the `erl_eval` 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}`.

**Prompting**

The default shell prompt function displays the name of the node (if the node can be part of a distributed system) and the current command number. The user can customize the prompt function by calling `shell:prompt_func/1` or by setting the application configuration parameter `shell_prompt_func` for the application `STDLIB`.

A customized prompt function is stated as a tuple `{Mod, Func}`. The function is called as `Mod:Func(L)`, where `L` is a list of key-value pairs created by the shell. Currently there is only one pair: `{history, N}`, where `N` is the current command number. The function should return a list of characters or an atom. This constraint is due to the Erlang I/O-protocol. Unicode characters beyond codepoint 255 are allowed in the list. Note that in restricted mode the call `Mod:Func(L)` must be allowed or the default shell prompt function will be called.

**Exports**

```
history(N) -> integer() >= 0
```

Types:
N = integer() >= 0
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() >= 0
Types:
   N = integer() >= 0
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) -> boolean()
Types:
   Bool = boolean()
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.

prompt_func(PromptFunc) -> PromptFunc2
Types:
   PromptFunc = PromptFunc2 = default | {module(), atom()}
Sets the shell prompt function to PromptFunc. The previous prompt function is returned.

start_restricted(Module) -> {error, Reason}
Types:
   Module = module()
   Reason = code:load_error_rsn()
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() -> no_return()
Exits a restricted shell and starts a normal shell. The function is meant to be called from the shell.

strings(Strings) -> Strings2
Types:
   Strings = Strings2 = boolean()
Sets pretty printing of lists to Strings. The previous value of the flag is returned.
The flag can also be set by the STDLIB application variable shell_strings. The default is true which means that lists of integers will be printed using the string syntax, when possible. The value false means that no lists will be printed using the string syntax.
shell_default

Erlang module

The functions in shell_default are called when no module name is given in a shell command.

Consider the following shell dialogue:

```
1 > lists:reverse("abc").
"cba"
2 > c(foo).
{ok, foo}
```

In command one, the module lists is called. In command two, no module name is specified. The shell searches the modules user_default followed by shell_default for the function foo/1.

shell_default is intended for "system wide" customizations to the shell. user_default is intended for "local" or individual user customizations.

Hint

To add your own commands to the shell, create a module called user_default and add the commands you want. Then add the following line as the first line in your .erlang file in your home directory.

```
code:load_abs("$PATH/user_default").
```

$PATH is the directory where your user_default module can be found.
This module provides functions for starting Erlang slave nodes. All slave nodes which are started by a master will terminate automatically when the master terminates. All TTY output produced at the slave will be sent back to the master node. File I/O is done via the master.

Slave nodes on other hosts than the current one are started with the program `rsh`. The user must be allowed to `rsh` to the remote hosts without being prompted for a password. This can be arranged in a number of ways (refer to the `rsh` documentation for details). A slave node started on the same host as the master inherits certain environment values from the master, such as the current directory and the environment variables. For what can be assumed about the environment when a slave is started on another host, read the documentation for the `rsh` program.

An alternative to the `rsh` program can be specified on the command line to `erl` as follows: `-rsh Program`.

The slave node should use the same file system at the master. At least, Erlang/OTP should be installed in the same place on both computers and the same version of Erlang should be used.

Currently, a node running on Windows NT can only start slave nodes on the host on which it is running.

The master node must be alive.

Exports

```
start(Host) -> {ok, Node} | {error, Reason}
start(Host, Name) -> {ok, Node} | {error, Reason}
start(Host, Name, Args) -> {ok, Node} | {error, Reason}
```

| Types: |
| Host = Name = atom() |
| Args = string() |
| Node = node() |
| Reason = timeout | no_rsh | {already_running, Node} |

Starts a slave node on the host `Host`. Host names need not necessarily be specified as fully qualified names; short names can also be used. This is the same condition that applies to names of distributed Erlang nodes.

The name of the started node will be `Name@Host`. If no name is provided, the name will be the same as the node which executes the call (with the exception of the host name part of the node name).

The slave node resets its `user` process so that all terminal I/O which is produced at the slave is automatically relayed to the master. Also, the file process will be relayed to the master.

The `Args` argument is used to set `erl` command line arguments. If provided, it is passed to the new node and can be used for a variety of purposes. See `erl(1)`

As an example, suppose that we want to start a slave node at host `H` with the node name `Name@H`, and we also want the slave node to have the following properties:
- directory `Dir` should be added to the code path;
- the Mnesia directory should be set to `M`;
- the unix `DISPLAY` environment variable should be set to the display of the master node.

The following code is executed to achieve this:
E = " -env DISPLAY " ++ net_adm:localhost() ++ ":0 ",
Arg = " -mnesia_dir " ++ M ++ ":pa " ++ Dir ++ E,
slave:start(H, Name, Arg).

If successful, the function returns \{ok, Node\}, where Node is the name of the new node. Otherwise it returns \{error, Reason\}, where Reason can be one of:

**timeout**

The master node failed to get in contact with the slave node. This can happen in a number of circumstances:

- Erlang/OTP is not installed on the remote host
- the file system on the other host has a different structure to the the master
- the Erlang nodes have different cookies.

**no_rsh**

There is no `rsh` program on the computer.

**{already_running, Node}**

A node with the name Name@Host already exists.

```erlang
start_link(Host) -> {ok, Node} | \{error, Reason\}
start_link(Host, Name) -> {ok, Node} | \{error, Reason\}
start_link(Host, Name, Args) -> {ok, Node} | \{error, Reason\}
```

**Types:**

- **Host** = Name = atom()
- **Args** = string()
- **Node** = node()
- **Reason** = timeout | no_rsh | \{already_running, Node\}

Starts a slave node in the same way as `start/1,2,3`, except that the slave node is linked to the currently executing process. If that process terminates, the slave node also terminates.

See `start/1,2,3` for a description of arguments and return values.

```erlang
stop(Node) -> ok
```

**Types:**

- **Node** = node()

Stops (kills) a node.

```erlang
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:

```bash
% erl -name abc -s slave pseudo klacke@super x --
```
pseudo(Master, ServerList) -> ok
Types:
   Master = node()
   ServerList = [atom()]
Starts a number of pseudo servers. A pseudo server is a server with a registered name which does absolutely nothing but pass on all message to the real server which executes at a master node. A pseudo server is an intermediary which only has the same registered name as the real server.
For example, if we have started a slave node N and want to execute pxw graphics code on this node, we can start the server pxw_server as a pseudo server at the slave node. The following code illustrates:

```
rpc:call(N, slave, pseudo, [node(), [pxw_server]]).
```

relay(Pid) -> no_return()
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 $B = \{ x \in A : S(x) \}$. 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 $A$ that belong to $B$. Two sets are **disjoint** if their intersection is the empty set. The **difference** of two sets $A$ and $B$ is the set that contains all elements of $A$ that do not belong to $B$. The **symmetric difference** of two sets is the set that contains those element that belong to either of the two sets, but not both. The union of a collection of sets is the smallest set that contains all the elements that belong to at least one set of the collection. The **intersection** of a non-empty collection of sets is the set that contains all elements that belong to every set of the collection.

The **Cartesian product** of two sets $X$ and $Y$, denoted $X \times Y$, is the set $\{a : a = (x, y) \text{ for some } x \in X \text{ and for some } y \in Y\}$. A **relation** is a subset of $X \times Y$. Let $R$ be a relation. The fact that $(x, y)$ belongs to $R$ is written as $x R y$. Since relations are sets, the definitions of the last paragraph (subset, union, and so on) apply to relations as well. The **domain** of $R$ is the set $\{x : x R y \text{ for some } y \in Y\}$. The **range** of $R$ is the set $\{y : x R y \text{ for some } x \in X\}$. The **converse** of $R$ is the set $\{a : a = (y, x) \text{ for some } (x, y) \in R\}$. If $A$ is a subset of $X$, then the **image** of $A$ under $R$ is the set $\{y : x R y \text{ for some } x \in A\}$, and if $B$ is a subset of $Y$, then the **inverse image** of $B$ is the set $\{x : x R y \text{ for some } y \in B\}$. If $R$ is a relation from $X$ to $Y$ and $S$ is a relation from $Y$ to $Z$, then the **relative product** of $R$ and $S$ is the relation $T$ from $X$ to $Z$ defined so that $x T z$ if and only if there exists an element $y$ in $Y$ such that $x R y$ and $y S z$. The **restriction** of $R$ to $A$ is the set $S$ defined so that $x S y$ if and only if there exists an element $x$ in $A$ such that $x R y$. If $S$ is a restriction of $R$ to $A$, then $R$ is an **extension** of $S$ to $X$. 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 $x S y$ if $x R y$ 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 $x R y$ if $x S y$ or $x = y$, then $R$ is the **weak** relation corresponding to $S$. A relation $R$ in $X$ is **reflexive** if $x R x$ for every element $x$ of $X$; it is **symmetric** if $x R y$ implies that $y R x$; and it is **transitive** if $x R y$ and $y R z$ imply that $x R z$.

A **function** $F$ is a relation, a subset of $X \times Y$, such that the domain of $F$ is equal to $X$ and such that for every $x$ in $X$ there is a unique element $y$ in $Y$ with $(x, y)$ in $F$. The latter condition can be formulated as follows: if $x F y$ and $x F z$ 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 $x F y$, 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 $F_2$.  

Ericsson AB. All Rights Reserved.: STDLIB | 403
Sometimes, when the range of a function is more important than the function itself, the function is called a \textit{family}. The domain of a family is called the \textit{index set}, and the range is called the \textit{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 \( \textit{a family in X} \) is used for such a family. When the indexed set is a set of subsets of a set \( X \), then we call \( x \) a \textit{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 \textit{union of the family} \( x \). If \( x \) is non-empty (the indexed set is non-empty), the \textit{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 \textit{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 \textit{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 \textit{equivalence class} of \( x \) with respect to \( R \) is the set of all those elements \( y \) of \( X \) for which \( x R y \) 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 \textit{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 \textit{binary relations}. We call a set of ordered sets \((x[1], ..., x[n])\) an \textit{(n-ary) relation}, and say that the relation is a subset of the Cartesian product \( X[1] \times ... \times X[n] \) where \( x[i] \) is an element of \( X[i] \), \( 1 \leq i \leq n \). The \textit{projection} of an n-ary relation \( R \) onto coordinate \( i \) is the set \( \{x[i] : (x[1], ..., x[i], ..., x[n]) \in R \text{ for some } x[j] \in X[j], 1 \leq j \leq n \text{ and } i \neq j \} \). The projections of a binary relation \( R \) onto the first and second coordinates are the domain and the range of \( R \) respectively. The relative product of binary relations can be generalized to n-ary relations as follows. Let \( TR \) be an ordered set \((R[1], ..., R[n])\) of binary relations from \( X \) to \( Y[i] \) and \( S \) a binary relation from \((Y[1] \times ... \times Y[n])\) to \( Z \). The \textit{relative product} of \( TR \) and \( S \) is the binary relation \( T \) from \( X \) to \( Z \) defined so that \( x T z \) if and only if there exists an element \( y[i] \) in \( Y[i] \) for each \( 1 \leq i \leq n \) such that \( x R[i] y[i] \) and \( (y[1], ..., y[n]) \in S \). Now let \( TR \) be a an ordered set \((R[1], ..., R[n])\) of binary relations from \( X[i] \) to \( Y[i] \) and \( S \) a subset of \((X[1] \times ... \times X[n])\). The \textit{multiple relative product} of \( TR \) and \( S \) is defined to be the set \( \{z : z = ((x[1], ..., x[n]), (y[1][1], ..., y[m][m])) \text{ for some } (x[1], ..., x[n]) \in S \text{ and for some } (x[i], y[i]) \in R[i], 1 \leq i \leq n \} \). The \textit{natural join} of an n-ary relation \( R \) and an m-ary relation \( S \) on coordinate \( i \) and \( j \) is defined to be the set \( \{z : z = (x[1], ..., x[n], y[1], ..., y[j-1], y[j+1], ..., y[m]) \text{ for some } (x[1], ..., x[n]) \in R \text{ and for some } (y[1], ..., y[m]) \in S \text{ such that } x[i] = y[j]\} \).

The sets recognized by this module will be represented by elements of the relation \textit{Sets}, defined as the smallest set such that:

- for every atom \( T \) except \( '_' \) and for every term \( (T, X) \) belongs to \( \text{Sets} \) (atomic sets);
- \( (_, [], []) \) belongs to \( \text{Sets} \) (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 \( \text{Sets} \) for every \( 1 \leq i \leq n \) then \( (T, X) \) belongs to \( \text{Sets} \) (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 \( \text{Sets} \) for every \( 1 \leq i \leq n \), then \( (T, X) \) belongs to \( \text{Sets} \) (typed unordered sets).

An \textit{external set} is an element of the range of \( \text{Sets} \). A \textit{type} is an element of the domain of \( \text{Sets} \). If \( S \) is an element of \( \text{Sets} \), then \( T \) is a \textit{valid type} of \( X, T \) is the type of \( S \), and \( X \) is the external set of \( S \). \textit{from_term/2} creates a set from a type and an Erlang term turned into an external set.

The actual sets represented by \( \text{Sets} \) are the elements of the range of the function \textit{Set} from \( \text{Sets} \) to Erlang terms and sets of Erlang terms:

- \( \text{Set}(T, \text{Term}) = \text{Term} \), where \( T \) is an atom;
- \( \text{Set}((T[1], ..., T[n]), \{X[1], ..., X[n]\}) = \text{Set}(T[1], X[1]), ..., \text{Set}(T[n], X[n])\);  
- \( \text{Set}(T, \{X[1], ..., X[n]\}) = \{\text{Set}(T, X[1]), ..., \text{Set}(T, X[n])\}; \)
- \( \text{Set}(T, []) = \{\} \).

When there is no risk of confusion, elements of \( \text{Sets} \) will be identified with the sets they represent. For instance, if \( U \) is the result of calling \textit{union/2} with \( \text{S1} \) and \( \text{S2} \) as arguments, then \( U \) is said to be the union of \( \text{S1} \) and \( \text{S2} \). A more precise formulation would be that \( \text{Set}(U) \) is the union of \( \text{Set}(\text{S1}) \) and \( \text{Set}(\text{S2}) \).
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`, 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`, `projection/2`, `restriction/3`, `substitution/2`) accept an Erlang function as a means to modify each element of a given unordered set. Such a function, called SetFun in the following, can be specified as a functional object (fun), a tuple `{external, Fun}`, or an integer. If SetFun is specified as a fun, the fun is applied to each element of the given set and the return value is assumed to be a set. If SetFun is specified as a tuple `{external, Fun}`, Fun is applied to the external set of each element of the given set and the return value is assumed to be an external set. Selecting the elements of an unordered set as external sets and assembling a new unordered set from a list of external sets is in the present implementation more efficient than modifying each element as a set. However, this optimization can only be utilized when the elements of the unordered set are atomic or ordered sets. It must also be the case that the type of the elements matches some clause of Fun (the type of the created set is the result of applying Fun to the type of the given set), and that Fun does nothing but selecting, duplicating or rearranging parts of the elements. Specifying a SetFun as an integer I is equivalent to specifying `{external, fun(A) -> element(I, A) end}`, but is to be preferred since it makes it possible to handle this case even more efficiently. Examples of SetFuns:

```erlang
fun sofs:union/1
    fun(S) -> sofs:partition(1, S) end
{external, fun(A) -> A end}
{external, fun({_,C}) -> {C,A} end}
{external, fun({_,_,C}) -> C end}
{external, fun({_,{_,{_,E}=C}}) -> {E,{E,C}} end}
2
```

The order in which a SetFun is applied to the elements of an unordered set is not specified, and may change in future versions of sofs.

The execution time of the functions of this module is dominated by the time it takes to sort lists. When no sorting is needed, the execution time is in the worst case proportional to the sum of the sizes of the input arguments and the returned value. A few functions execute in constant time: `from_external`, `is_empty_set`, `is_set`, `is_sofs_set`, `to_external`, `type`.

The functions of this module exit the process with a `badarg`, `bad_function`, or `type_mismatch` message when given badly formed arguments or sets the types of which are not compatible.

When comparing external sets the operator `==/2` is used.

Data Types

`anyset() = ordset() | a_set()`

Any kind of set (also included are the atomic sets).

`binary_relation() = relation()`

A binary relation.
external_set() = term()
  An external set.
family() = a_function()
  A family (of subsets).
a_function() = relation()
  A function.
ordset()
  An ordered set.
relation() = a_set()
  An n-ary relation.
a_set()
  An unordered set.
set_of_sets() = a_set()
  An unordered set of unordered sets.
set_fun() = integer() >= 1
  | {external, fun((external_set()) -> external_set())}
  | fun((anyset()) -> anyset())
  A SetFun.
spec_fun() = {external, fun((external_set()) -> boolean())}
  | fun((anyset()) -> boolean())
type() = term()
  A type.
tuple_of(T)
  A tuple where the elements are of type T.

Exports

da_function(Tuples) -> Function
da_function(Tuples, Type) -> Function
Types:
  Function = a_function()
  Tuples = [tuple()]
  Type = type()
Creates a function. a_function(F, T) is equivalent to from_term(F, T), if the result is a function. If no type is explicitly given, [{atom, atom}] is used as type of the function.

canonical_relation(SetOfSets) -> BinRel
Types:
BinRel = \textbf{binary\textunderscore relation()}
SetOfSets = \textbf{set\textunderscore of\textunderscore sets()}

Returns the binary relation containing the elements (E, Set) such that Set belongs to SetOfSets and E belongs to Set. If SetOfSets is a \textit{partition} of a set X and R is the equivalence relation in X induced by SetOfSets, then the returned relation is the \textit{canonical map} from X onto the equivalence classes with respect to R.

1> Ss = sofs:from_string([[a,b],[b,c]]),
   CR = sofs:canonical\_relation(Ss),
   sofs:to_external(CR).
   \[
   \{\{a,\{a,b\}\},\{b,\{a,b\}\},\{b,\{b,c\}\},\{c,\{b,c\}\}\}
   \]

\textbf{composite(Function1, Function2) -> Function3}

Types:
   
   \textbf{Function1 = Function2 = Function3 = a\_function()}

Returns the \textit{composite} of the functions Function1 and Function2.

1> F1 = sofs:a\_function([[a,1],[b,2],[c,2]]),
   F2 = sofs:a\_function([[1,x],[2,y],[3,z]]),
   F = sofs:composite(F1, F2),
   sofs:to_external(F).
   \[
   \{\{a,x\},\{b,y\},\{c,y\}\}
   \]

\textbf{constant\_function(Set, AnySet) -> Function}

Types:
   
   \textbf{AnySet = anyset()}
   \textbf{Function = a\_function()}
   \textbf{Set = a\_set()}

Creates the \textit{function} that maps each element of the set Set onto AnySet.

1> S = sofs:set([a,b]),
   E = sofs:from\_term(1),
   R = sofs:constant\_function(S, E),
   sofs:to\_external(R).
   \[
   \{\{a,1\},\{b,1\}\}
   \]

\textbf{converse(BinRel1) -> BinRel2}

Types:
   
   \textbf{BinRel1 = BinRel2 = binary\_relation()}

Returns the \textit{converse} of the binary relation BinRel1.

1> R1 = sofs:relation([[1,a],[2,b],[3,a]]),
   R2 = sofs:converse(R1),
   sofs:to\_external(R2).
difference(Set1, Set2) -> Set3
Types:
    Set1 = Set2 = Set3 = a_set()
Returns the *difference* of the sets Set1 and Set2.

digraph_to_family(Graph) -> Family
digraph_to_family(Graph, Type) -> Family
Types:
    Graph = digraph:graph()
    Family = family()
    Type = type()
Creates a *family* 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* 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 = a_set()
Returns the *domain* of the binary relation BinRel.

drestriction(BinRel1, Set) -> BinRel2
Types:
    BinRel1 = BinRel2 = binary_relation()
    Set = a_set()
Returns the difference between the binary relation BinRel1 and the *restriction* of BinRel1 to Set.

drestriction(R, S) is equivalent to difference(R, restriction(R, S)).
drestriction(SetFun, Set1, Set2) -> Set3
Types:
   SetFun = set_fun()
   Set1 = Set2 = Set3 = a_set()
Returns a subset of Set1 containing those elements that do not yield an element in Set2 as the result of applying SetFun.

drestriction(F, S1, S2) is equivalent to difference(S1, restriction(F, S1, S2)).

empty_set() -> Set
Types:
   Set = a_set()
Returns the *untyped empty set*. empty_set() is equivalent to from_term([], ['_'_]).

extension(BinRel1, Set, AnySet) -> BinRel2
Types:
   AnySet = anyset()
   BinRel1 = BinRel2 = binary_relation()
   Set = a_set()
Returns the *extension* of BinRel1 such that for each element E in Set that does not belong to the *domain* of BinRel1, BinRel2 contains the pair (E, AnySet).

family(Tuples) -> Family
gfamily(Tuples, Type) -> Family
Types:
   Family = family()
   Tuples = [tuple()]
   Type = type()
Creates a *family of subsets*. family(F, T) is equivalent to from_term(F, T), if the result is a family. If no type 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, 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.

```prolog
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 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 of Family1[i].

```prolog
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 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 of Family1[i].

```prolog
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 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 of Family1[i].
If Family1[i] is an empty set for some i, then the process exits with a badarg message.

```prolog
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)).
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, then Family3 is the family such that the index set is the intersection of Family1's and Family2's index sets, and Family3[i] is the intersection of Family1[i] and Family2[i].

1> F1 = sofs:family([[a,[1,2]], [b,[3,4]], [c,[5,6]]]),
    F2 = sofs:family([[b,[4,5]], [c,[7,8]], [d,[9,10]]]),
    F3 = sofs:family_intersection(F1, F2),
    sofs:to_external(F3).
[[b,[4]], [c,[]]]

family_projection(SetFun, Family1) -> Family2
Types:
    SetFun = set_fun()
    Family1 = Family2 = family()
If Family1 is a family then Family2 is the family with the same index set as Family1 such that Family2[i] is the result of calling SetFun with Family1[i] as argument.

1> F1 = sofs:from_term([[a,[[1,2],[2,3]]],[b,[]]]),
    F2 = sofs:family_projection(fun sofs:union/1, F1),
    sofs:to_external(F2).
[[a,[1,2,3]], [b,[]]]

family_range(Family1) -> Family2
Types:
    Family1 = Family2 = family()
If Family1 is a family 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 of Family1[i].

1> FR = sofs:from_term([[a,[[1,a],[2,b],[3,c]]],[b,[]],[c,[[4,d],[5,e]]]]),
    F = sofs:family_range(FR),
    sofs:to_external(F).
[[a,[a,b,c]], [b,[]], [c,[d,e]]]

family_specification(Fun, Family1) -> Family2
Types:

Ericsson AB. All Rights Reserved.: STDLIB | 411
Fun = spec_fun()
Family1 = Family2 = family()

If Family1 is a family, then Family2 is the restriction 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 of Family1[i], otherwise Fun is applied to Family1[i].

```erl
1> F1 = sofs:family([{{a,[1,2,3]},{b,[1,2]},{c,[1]}},
                     SpecFun = fun(S) -> sofs:no_elements(S) =:= 2 end,
                     F2 = sofs:family_specification(SpecFun, F1),
                     sofs:to_external(F2).

[{b,[1,2]}]
```

digraph:graph() family_to_digraph(Family, GraphType) -> Graph

Types:
Graph = digraph:graph()
Family = family()
GraphType = [digraph:d_type()]

Creates a directed graph from the family Family. For each pair (a, {b[1], ..., b[n]}) of Family, the vertex a as well the edges (a, b[i]) for 1 <= i <= n are added to a newly created directed graph.

If no graph type is given digraph:new/0 is used for creating the directed graph, otherwise the GraphType argument is passed on as second argument to digraph:new/1.

It F is a family, it holds that F is a subset of digraph_to_family(digraph_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.

digraph:graph() family_to_relation(Family) -> BinRel

Types:
Family = family()
BinRel = binary_relation()

If Family is a family, 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].

```erl
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}}
```

digraph:graph() family_union(Family1) -> Family2

Types:
Family1 = Family2 = family()

If Family1 is a family 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 of Family1[i].
family_union(F) is equivalent to family_projection(fun sofs:union/1, F).

family_union(Family1, Family2) -> Family3
Types:
  Family1 = Family2 = Family3 = family()
If Family1 and Family2 are families, 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.

field(BinRel) -> Set
Types:
  BinRel = binary_relation()
  Set = a_set()
Returns the field of the binary relation BinRel.

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 ExternalSet and the type Type. It is assumed that Type is a valid type of ExternalSet.

from_sets(ListOfSets) -> Set
Types:
  Set = a_set()
  ListOfSets = [anyset()]
Returns the unordered set containing the sets of the list ListOfSets.
Ericsson AB. All Rights Reserved.: STDLIB

```
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\}]]
```

The `from_sets` function takes a tuple of sets and returns an ordered set containing the sets of the non-empty tuple.

```
from_sets(TupleOfSets) -> Ordset
Types:
   Ordset = ordset()
   TupleOfSets = tuple_of(anyset())
Returns the ordered set containing the sets of the non-empty tuple TupleOfSets.
```

The `from_term` function creates an element of `Sets` by traversing the term `Term`, sorting lists, removing duplicates and verifying a valid type for the obtained external set.

```
from_term(Term) -> AnySet
from_term(Term, Type) -> AnySet
Types:
   AnySet = anyset()
   Term = term()
   Type = type()
Creates an element of Sets by traversing the term Term, sorting lists, removing duplicates and deriving or verifying a valid type for the so obtained external set. An explicitly given type 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:
```

```
S = sofs:from_term([\{\"foo\"\},[1,1],\{\"foo\"\},[2,2]]),
sofs:to_external(S).
[\{\"foo\"\},[1]],\{\"foo\"\},[2]]
```

`from_term` can be used for creating atomic or ordered sets. The only purpose of such a set is that of later building unordered sets since all functions in this module that do anything operate on unordered sets. Creating unordered sets from a collection of ordered sets may be the way to go if the ordered sets are big and one does not want to waste heap by rebuilding the elements of the unordered set. An example showing that a set can be built "layer by layer":

```
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,\{6,5\}\}]]
```

Other functions that create sets are `from_external/2` and `from_sets/1`. Special cases of `from_term/2` are `a_function/1,2`, `empty_set/0`, `family/1,2`, `relation/1,2`, and `set/1,2`.

```
image(BinRel, Set1) -> Set2
Types:
```

414 | Ericsson AB. All Rights Reserved.: STDLIB
BinRel = binary_relation()
Set1 = Set2 = a_set()

Returns the image of the set Set1 under the binary relation BinRel.

```erlang
1> R = sofs:relation([\{1,a\},\{2,b\},\{2,c\},\{3,d\}\]),
   S1 = sofs:set([1,2]),
   S2 = sofs:image(R, S1),
   sofs:to_external(S2).
[a,b,c]
```

intersection(SetOfSets) -> Set
Types:
   Set = a_set()
   SetOfSets = set_of_sets()

Returns the intersection of the set of sets SetOfSets.
Intersecting an empty set of sets exits the process with a badarg message.

```erlang
intersection(Set1, Set2) -> Set3
Types:
   Set1 = Set2 = Set3 = a_set()

Returns the intersection of Set1 and Set2.

intersection_of_family(Family) -> Set
Types:
   Family = family()
   Set = a_set()

Returns the intersection of the family Family.
Intersecting an empty family exits the process with a badarg message.

```erlang
1> F = sofs:family([\{a,\[0,2,4\]\},\{b,\[0,1,2\]\},\{c,\[2,3\]\}\]),
   S = sofs:intersection_of_family(F),
   sofs:to_external(S).
[2]
```

inverse(Function1) -> Function2
Types:
   Function1 = Function2 = a_function()

Returns the inverse of the function Function1.

```erlang
1> R1 = sofs:relation([\{1,a\},\{2,b\},\{3,c\}\]),
   R2 = sofs:inverse(R1),
   sofs:to_external(R2).
```
inverse_image(BinRel, Set1) -> Set2

Types:
   BinRel = binary_relation()
   Set1 = Set2 = a_set()

Returns the inverse image of Set1 under the binary relation BinRel.

```erlang
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 = boolean()
   BinRel = binary_relation()

Returns true if the binary relation BinRel is a function or the untyped empty set, false otherwise.

```erlang
is_disjoint(Set1, Set2) -> Bool

Types:
   Bool = boolean()
   Set1 = Set2 = a_set()

Returns true if Set1 and Set2 are disjoint, false otherwise.

is_empty_set(AnySet) -> Bool

Types:
   AnySet = anyset()
   Bool = boolean()

Returns true if AnySet is an empty unordered set, false otherwise.

is_equal(AnySet1, AnySet2) -> Bool

Types:
   AnySet1 = AnySet2 = anyset()
   Bool = boolean()

Returns true if the AnySet1 and AnySet2 are equal, false otherwise. This example shows that ==/2 is used when comparing sets for equality:

```erlang
1> S1 = sofs:set([1.0]),
   S2 = sofs:set([1]),
   sofs:is_equal(S1, S2).
   true
```
is_set(AnySet) -> Bool
Types:
   AnySet = anyset()
   Bool = boolean()
Returns true if AnySet is an unordered set, and false if AnySet is an ordered set or an atomic set.

is_sofs_set(Term) -> Bool
Types:
   Bool = boolean()
   Term = term()
Returns true if Term is an unordered set, an ordered set or an atomic set, false otherwise.

is_subset(Set1, Set2) -> Bool
Types:
   Bool = boolean()
   Set1 = Set2 = a_set()
Returns true if Set1 is a subset of Set2, false otherwise.

is_type(Term) -> Bool
Types:
   Bool = boolean()
   Term = term()
Returns true if the term Term is a type.

join(Relation1, I, Relation2, J) -> Relation3
Types:
   Relation1 = Relation2 = Relation3 = relation()
   I = J = integer() >= 1
Returns the natural join of the relations Relation1 and Relation2 on coordinates I and J.

```erlang
1> R1 = sofs:relation([{a,x,1},{b,y,2}]),
   R2 = sofs:relation([{1,f,g},{1,h,i},{2,3,4}]),
   J = sofs:join(R1, 3, R2, 1),
   sofs:to_external(J).
[{a,x,1,f,g},{a,x,1,h,i},{b,y,2,3,4}]
```

multiple_relative_product(TupleOfBinRels, BinRel1) -> BinRel2
Types:
   TupleOfBinRels = tuple_of(BinRel)
   BinRel = BinRel1 = BinRel2 = binary_relation()
If TupleOfBinRels is a non-empty tuple \{R[1], ..., R[n]\} of binary relations and BinRel1 is a binary relation, then BinRel2 is the multiple relative product of the ordered set \{R[i], ..., R[n]\} and BinRel1.
1> Ri = sofs:relation([\{a,1\},\{b,2\},\{c,3\}\]),
R = sofs:relation([\{a,b\},\{b,c\},\{c,a\}\]),
MP = sofs:multiple_relative_product([Ri, Ri], R),
sofs:to_external(sofs:range(MP)).
\{[1,2],[2,3],[3,1]\}

**no_elements(ASet)** -> NoElements

Types:

- ASet = a_set() | ordset()
- NoElements = integer() >= 0

Returns the number of elements of the ordered or unordered set ASet.

**partition(SetOfSets)** -> Partition

Types:

- SetOfSets = set_of_sets()
- Partition = a_set()

Returns the partition 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 = a_set()

Returns the partition 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 = a_set()

Returns a pair of sets that, regarded as constituting a set, forms a partition 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)}.

partition(F, S1, S2) is equivalent to {restriction(F, S1, S2), drestriction(F, S1, S2)}.

partition_family(SetFun, Set) -> Family
Types:
    Family = family()
    SetFun = set_fun()
    Set = a_set()
Returns the family Family where the indexed set is a partition 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.

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).

product(TupleOfSets) -> Relation
Types:
    Relation = relation()
    TupleOfSets = tuple_of(a_set())
Returns the Cartesian product 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).

product(Set1, Set2) -> BinRel
Types:
    BinRel = binary_relation()
    Set1 = Set2 = a_set()
Returns the Cartesian product of Set1 and Set2.

1> S1 = sofs:set([1,2]),
S2 = sofs:set([a,b]),
R = sofs:product(S1, S2),
sofs:to_external(R).
product(S1, S2) is equivalent to product([{S1, S2}]).

projection(SetFun, Set1) -> Set2
Types:
  SetFun = set_fun()
  Set1 = Set2 = a_set()
Returns the set created by substituting each element of Set1 by the result of applying SetFun to the element.
If SetFun is a number i >= 1 and Set1 is a relation, then the returned set is the projection of Set1 onto coordinate i.

1> S1 = sofs:from_term([{1,a},{2,b},{3,a}]),
   S2 = sofs:projection(2, S1),
   sofs:to_external(S2).
[a,b]

range(BinRel) -> Set
Types:
  BinRel = binary_relation()
  Set = a_set()
Returns the range of the binary relation BinRel.

1> R = sofs:relation([{1,a},{1,b},{2,b},{2,c}]),
   S = sofs:range(R),
   sofs:to_external(S).
[a,b,c]

relation(Tuples) -> Relation
relation(Tuples, Type) -> Relation
Types:
  N = integer()
  Type = N | type()
  Relation = relation()
  Tuples = [tuple()]
Creates a relation. relation(R, T) is equivalent to from_term(R, T), if T is a type and the result is a relation. If Type is an integer N, then [{atom, ..., atom}]), where the size of the tuple is N, is used as type of the relation. If no type is explicitly given, the size of the first tuple of Tuples is used if there is such a tuple.
relation([]) is equivalent to relation([], 2).

relation_to_family(BinRel) -> Family
Types:
Family = \textit{family()}
BinRel = \textit{binary\_relation()}

Returns the family Family such that the index set is equal to the domain of the binary relation BinRel, and Family[i] is the image of the set of i under BinRel.

```erlang
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]]]
```

\textbf{relative\_product(ListOfBinRels) -> BinRel2}

\textbf{relative\_product(ListOfBinRels, BinRel1) -> BinRel2}

\textbf{Types:}

- \textit{ListOfBinRels} = [BinRel, ...]
- \textit{BinRel} = BinRel1 = BinRel2 = \textit{binary\_relation()}

If ListOfBinRels is a non-empty list \([R[1], \ldots, R[n]]\) of binary relations and BinRel1 is a binary relation, then BinRel2 is the relative product of the ordered set \((R[i], \ldots, R[n])\) and BinRel1.

If BinRel1 is omitted, the relation of equality between the elements of the Cartesian product of the ranges of \(R[i]\), range \(R[1] \times \ldots \times \text{range } R[n]\), is used instead (intuitively, nothing is "lost").

```erlang
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 list of one element is not identified with the element itself.

\textbf{relative\_product(BinRel1, BinRel2) -> BinRel3}

\textbf{Types:}

- \textit{BinRel1} = BinRel2 = BinRel3 = \textit{binary\_relation()}

Returns the relative product of the binary relations BinRel1 and BinRel2.

```
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).
```

Returns the relative product of the converse of the binary relation BinRel1 and the binary relation BinRel2.
relative_product1(R1, R2) is equivalent to relative_product(converse(R1), R2).

restriction(BinRel1, Set) -> BinRel2
Types:
   BinRel1 = BinRel2 = binary_relation()
   Set = a_set()
Returns the restriction of the binary relation BinRel1 to Set.

restriction(SetFun, Set1, Set2) -> Set3
Types:
   SetFun = set_fun()
   Set1 = Set2 = Set3 = a_set()
Returns a subset of Set1 containing those elements that yield an element in Set2 as the result of applying SetFun.

set(Terms) -> Set
set(Terms, Type) -> Set
Types:
   Set = a_set()
   Terms = [term()]
   Type = type()
Creates an unordered set. set(L, T) is equivalent to from_term(L, T), if the result is an unordered set. If no type is explicitly given, [atom] is used as type of the set.

specification(Fun, Set1) -> Set2
Types:
   Fun = spec_fun()
   Set1 = Set2 = a_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 of each element, otherwise Fun is applied to each element.
sofs

R2 = sofs:relation([\{x,1\}, \{x,2\}, \{y,3\}\]),
S1 = sofs:from_sets([R1,R2]),
S2 = sofs:specification(fun sofs:is_a_function/1, S1),
sofs:to_external(S2).

strict_relation(BinRel1) -> BinRel2
Types:
   BinRel1 = BinRel2 = binary_relation()
Returns the *strict relation* 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 = a_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))).
   \[
   \{a,1\}, \{b,2\}\]
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).
   \[
   \{\{a,1\}, \{b,2\}, \{c,3\}\}\
   \]

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:

```prolog
images2(SetOfSets, BinRel) ->
    CR = sofs:canonical_relation(SetOfSets),
    R = sofs:relative_product1(CR, BinRel),
    sofs:relation_to_family(R).
```

```prolog
symdiff(Set1, Set2) -> Set3
```

Types:
- `Set1 = Set2 = Set3 = a_set()`

Returns the *symmetric difference* (or the Boolean sum) of `Set1` and `Set2`.

```prolog
1> S1 = sofs:set([1,2,3]),
   S2 = sofs:set([2,3,4]),
   P = sofs:symdiff(S1, S2),
   sofs:to_external(P).
[1,4]
```

```prolog
symmetric_partition(Set1, Set2) -> {Set3, Set4, Set5}
```

Types:
- `Set1 = Set2 = Set3 = Set4 = Set5 = a_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`.

```prolog
to_external(AnySet) -> ExternalSet
```

Types:
- `ExternalSet = external_set()`
- `AnySet = anyset()`

Returns the *external set* of an atomic, ordered or unordered set.

```prolog
to_sets(ASet) -> Sets
```

Types:
- `ASet = a_set() | ordset()`
- `Sets = tuple_of(AnySet) | [AnySet]`
- `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 of an atomic, ordered or unordered set.

union(SetOfSets) -> Set
Types:
    Set = a_set()
    SetOfSets = set_of_sets()
Returns the union of the set of sets SetOfSets.

union(Set1, Set2) -> Set3
Types:
    Set1 = Set2 = Set3 = a_set()
Returns the union of Set1 and Set2.

union_of_family(Family) -> Set
Types:
    Family = family()
    Set = a_set()
Returns the union of the family Family.

1> F = sofs:family([[a,[0,2,4]],[b,[0,1,2]],[c,[2,3]]]),
S = sofs:union_of_family(F),
sofs:to_external(S).
[0,1,2,3,4]

weak_relation(BinRel1) -> BinRel2
Types:
    BinRel1 = BinRel2 = binary_relation()
Returns a subset S of the weak relation W corresponding to the binary relation BinRel1. Let F be the field of BinRel1. The subset S is defined so that x S y if x W y for some x in F and for some y in F.

1> R1 = sofs:relation([[1,1],[1,2],[3,1]]),
R2 = sofs:weak_relation(R1),
sofs:to_external(R2).
[[1,1],[1,2],[2,2],[3,1],[3,3]]

See Also
dict(3), digraph(3), orddict(3), ordsets(3), sets(3)
string

Erlang module

This module contains functions for string processing.

Exports

\texttt{len(String) -> Length}

Types:
\begin{itemize}
\item \texttt{String = string()}
\item \texttt{Length = integer() \geq 0}
\end{itemize}

Returns the number of characters in the string.

\texttt{equal(String1, String2) -> boolean()}

Types:
\begin{itemize}
\item \texttt{String1 = String2 = string()}
\end{itemize}

Tests whether two strings are equal. Returns \texttt{true} if they are, otherwise \texttt{false}.

\texttt{concat(String1, String2) -> String3}

Types:
\begin{itemize}
\item \texttt{String1 = String2 = String3 = string()}
\end{itemize}

Concatenates two strings to form a new string. Returns the new string.

\texttt{chr(String, Character) -> Index}
\texttt{rchr(String, Character) -> Index}

Types:
\begin{itemize}
\item \texttt{String = string()}
\item \texttt{Character = char()}
\item \texttt{Index = integer() \geq 0}
\end{itemize}

Returns the index of the first/last occurrence of \texttt{Character} in \texttt{String}. \texttt{0} is returned if \texttt{Character} does not occur.

\texttt{str(String, SubString) -> Index}
\texttt{rstr(String, SubString) -> Index}

Types:
\begin{itemize}
\item \texttt{String = SubString = string()}
\item \texttt{Index = integer() \geq 0}
\end{itemize}

Returns the position where the first/last occurrence of \texttt{SubString} begins in \texttt{String}. \texttt{0} is returned if \texttt{SubString} does not exist in \texttt{String}. For example:

\begin{verbatim}
> string:str(" Hello Hello World World ", "Hello World").
8
\end{verbatim}
span(String, Chars) -> Length
cspan(String, Chars) -> Length

Types:
  String = Chars = string()
  Length = integer() >= 0

Returns the length of the maximum initial segment of String, which consists entirely of characters from (not from) Chars.

For example:

```prolog
> string:span("    abcdef", " 	").
 5
> string:cspan("    abcdef", " 	").
 0
```

substr(String, Start) -> SubString
substr(String, Start, Length) -> SubString

Types:
  String = SubString = string()
  Start = integer() >= 1
  Length = integer() >= 0

Returns a substring of String, starting at the position Start, and ending at the end of the string or at length Length.

For example:

```prolog
> substr("Hello World", 4, 5).
  "lo Wo"
```

tokens(String, SeparatorList) -> Tokens

Types:
  String = SeparatorList = string()
  Tokens = [Token :: nonempty_string()]

Returns a list of tokens in String, separated by the characters in SeparatorList.

For example:

```prolog
> tokens("abc defxghix jkl", "x ").
["abc", "def", "ghi", "jkl"]
```

Note that, as shown in the example above, two or more adjacent separator characters in String will be treated as one. That is, there will not be any empty strings in the resulting list of tokens.

join(StringList, Separator) -> String

Types:
string

StringList = [string()]  
Separator = String = string()  

Returns a string with the elements of StringList separated by the string in Separator.  

For example:

```
> join(["one", "two", "three"], ", ").  
"one, two, three"
```

chars(Character, Number) -> String  
chars(Character, Number, Tail) -> String  

Types:  
Character = char()  
Number = integer() >= 0  
Tail = 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() >= 0  

Returns a string containing String repeated Number times.

words(String) -> Count  
words(String, Character) -> Count  

Types:  
String = string()  
Character = char()  
Count = integer() >= 1  

Returns the number of words in String, separated by blanks or Character.  

For example:

```
> words(" Hello old boy!", $o).  
4
```

sub_word(String, Number) -> Word  
sub_word(String, Number, Character) -> Word  

Types:  
String = Word = string()  
Number = integer()  
Character = char()  

Returns the word in position Number of String. Words are separated by blanks or Characters.
For example:

```prolog
> string:sub_word(" Hello old boy !",3,$o).
"ld b"
```

```prolog
strip(String :: string()) -> string()
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:

```prolog
> string:strip("...Hello.....", both, $.)
"Hello"
```

```prolog
left(String, Number) -> Left
left(String, Number, Character) -> Left
```

Types:

```
String = Left = string()
Number = integer() >= 0
Character = char()
```

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:

```prolog
> string:left("Hello",10,$.).
"Hello....."
```

```prolog
right(String, Number) -> Right
right(String, Number, Character) -> Right
```

Types:

```
String = Right = string()
Number = integer() >= 0
Character = char()
```

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:

```erlang
> string:right("Hello", 10, $). ".....Hello"
```

centre(String, Number) -> Centered
centre(String, Number, Character) -> Centered

Types:
- String = Centered = string()
- Number = integer() >= 0
- Character = char()

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() >= 1

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:

```erlang
sub_string("Hello World", 4, 8).
"lo Wo"
```

to_float(String) -> {Float, Rest} | {error, Reason}

Types:
- String = string()
- Float = float()
- Rest = string()
- Reason = no_float | not_a_list

Argument String is expected to start with a valid text represented float (the digits being ASCII values). Remaining characters in the string after the float are returned in Rest.

Example:

```erlang
> {F1,Fs} = string:to_float("1.0-1.0e-1"),
> {F2,[]} = string:to_float(Fs),
> F1+F2.
0.9
> string:to_float("3/2=1.5").
{error,no_float}
> string:to_float("-1.5eX").
```
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}
```

to_lower(String) -> Result
to_lower(Char) -> CharResult
to_upper(String) -> Result
to_upper(Char) -> CharResult

Types:
- String = Result = io_lib:latin1_string()
- Char = CharResult = char()

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 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.

The function terminate_child/2 can be used for children under simple_one_for_one supervisors by giving the child's `pid()` as the second argument. If instead the child specification identifier is used, terminate_child/2 will return `{error,simple_one_for_one}`.

Because a simple_one_for_one supervisor could have many children, it shuts them all down at same time. So, order in which they are stopped is not defined. For the same reason, it could have an overhead with regards to the Shutdown 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:

```erlang
child_spec() = {Id,StartFunc,Restart,Shutdown,Type,Modules}
```
Id = term()
StartFunc = {M,F,A}
M = F = atom()
A = [term()]
Restart = permanent | transient | temporary
Shutdown = brutal_kill | int()>0 | infinity
Type = worker | supervisor
Modules = [Module] | dynamic
Module = atom()

- **Id** is a name that is used to identify the child specification internally by the supervisor.
- **StartFunc** defines the function call used to start the child process. It should be a module-function-arguments tuple \( (M, F, A) \) used as \( \text{apply}(M, F, A) \).

The start function **must create and link to** the child process, and should return \( \{\text{ok,Child}\} \) or \( \{\text{ok,Child,Info}\} \) where **Child** is the pid of the child process and **Info** an arbitrary term which is ignored by the supervisor.

The start function can also return **ignore** if the child process for some reason cannot be started, in which case the child specification will be kept by the supervisor (unless it is a temporary child) but the non-existing child process will be ignored.

If something goes wrong, the function may also return an error tuple \( \{\text{error,Error}\} \).

Note that the **start_link** functions of the different behaviour modules fulfill the above requirements.

- **Restart** defines when a terminated child process should be restarted. A **permanent** child process should always be restarted, a **temporary** child process should never be restarted (even when the supervisor's restart strategy is **rest_for_one** or **one_for_all** and a sibling's death causes the temporary process to be terminated) and a **transient** child process should be restarted only if it terminates abnormally, i.e. with another exit reason than **normal**, **shutdown** or \( \{\text{shutdown,Term}\} \).

- **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 number of milliseconds, 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. It is also allowed to set it to **infinity** if the child process is a worker.

**Warning:**

Be careful by setting the **Shutdown** strategy to **infinity** when the child process is a worker. Because, in this situation, the termination of the supervision tree depends on the child process, it must be implemented in a safe way and its cleanup procedure must always return.

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 \( [\text{Module}] \), 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.

Data Types

child() = undefined | pid()
child_id() = term()

Not a pid().

child_spec() =
  {Id :: child_id(),
   StartFunc :: mfargs(),
   Restart :: restart(),
   Shutdown :: shutdown(),
   Type :: worker(),
   Modules :: modules()}

mfargs() =
  {M :: module(), F :: atom(), A :: [term()] | undefined}

A (the argument list) has the value undefined if Restart is temporary.

modules() = [module()] | dynamic
restart() = permanent | transient | temporary
shutdown() = brutal_kill | timeout()
strategy() = one_for_all
  | one_for_one
  | rest_for_one
  | simple_one_for_one
sup_ref() = (Name :: atom())
  | {Name :: atom(), Node :: node()}
  | {global, Name :: atom()}
  | {via, Module :: module(), Name :: any()}
  | pid()

worker() = worker | supervisor

Exports

start_link(Module, Args) -> startlink_ret()
start_link(SupName, Module, Args) -> startlink_ret()

Types:
  SupName = sup_name()
  Module = module()
  Args = term()
  startlink_ret() = {ok, pid()}
    | ignore
    | {error, startlink_err()}
  startlink_err() = {already_started, pid()}
    | {shutdown, term()}
    | term()
  sup_name() = {local, Name :: atom()}
    | {global, Name :: atom()}
Creates a supervisor process as part of a supervision tree. The function will, among other things, ensure that the supervisor is linked to the calling process (its supervisor).

The created supervisor process calls Module:init/1 to find out about restart strategy, maximum restart frequency and child processes. To ensure a synchronized start-up procedure, start_link/2, 3 does not return until Module:init/1 has returned and all child processes have been started.

If SupName={local, Name} the supervisor is registered locally as Name using register/2. If SupName={global, Name} the supervisor is registered globally as Name using global:register_name/2. If SupName={via, Module, Name} the supervisor is registered as Name using the registry represented by Module. The Module callback should export the functions register_name/2, unregister_name/1 and send/2, which should behave like the corresponding functions in global. Thus, {via, global, Name} is a valid reference.

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, {already_started,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 supervisor will first terminate all already started child processes with reason shutdown and then terminate itself and return {error, {shutdown, Reason}}.

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.
• \{via, Module, Name\}, if the supervisor is registered through an alternative process registry.

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 = sup_ref()
- \Id = pid() | child_id()
- \Result = ok | \{error, Error\}
- \Error = not_found | simple_one_for_one

Tells the supervisor \SupRef to terminate the given child.

If the supervisor is not simple_one_for_one, \Id must be the child specification identifier. The process, if there is one, is terminated and, unless it is a temporary child, the child specification is kept by the supervisor. The child process may 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.

If the child is temporary, the child specification is deleted as soon as the process terminates. This means that delete_child/2 has no meaning and restart_child/2 can not be used for these children.

If the supervisor is simple_one_for_one, \Id must be the child process' pid(). If the specified process is alive, but is not a child of the given supervisor, the function will return \{error, not_found\}. If the child specification identifier is given instead of a pid(), the function will return \{error, simple_one_for_one\}.

If successful, the function returns ok. If there is no child specification with the specified \Id, the function returns \{error, not_found\}.

See start_child/2 for a description of SupRef.

### delete_child(SupRef, \Id) -> Result

Types:
SupRef = sup_ref()
Id = child_id()
Result = ok | {error, Error}
   Error = running | restarting | not_found | simple_one_for_one

Tells the supervisor SupRef to delete the child specification identified by Id. The corresponding child process must not be running, use terminate_child/2 to terminate it.

See start_child/2 for a description of SupRef.

If successful, the function returns ok. If the child specification identified by Id exists but the corresponding child process is running or about to be restarted, the function returns {error, running} or {error, restarting} respectively. If the child specification identified by Id does not exist, the function returns {error, not_found}.

restart_child(SupRef, Id) -> Result
Types:
   SupRef = sup_ref()
   Id = child_id()
   Result = {ok, Child :: child()}
           | {ok, Child :: child(), Info :: term()}
           | {error, Error}
   Error = running
           | restarting
           | not_found
           | simple_one_for_one
           | term()

Tells the supervisor SupRef to restart a child process corresponding to the child specification identified by Id. The child specification must exist and the corresponding child process must not be running.

Note that for temporary children, the child specification is automatically deleted when the child terminates, and thus it is not possible to restart such children.

See start_child/2 for a description of SupRef.

If the child specification identified by Id does not exist, the function returns {error, not_found}. If the child specification exists but the corresponding process is already running, the function returns {error, running}.

If the child process start function returns {ok, Child} or {ok, Child, Info}, the pid is added to the supervisor and the function returns the same value.

If the child process start function returns ignore, the pid remains set to undefined and the function returns {ok, undefined}.

If the child process start function returns an error tuple or an erroneous value, or if it fails, the function returns {error, Error} where Error is a term containing information about the error.

which_children(SupRef) -> [{Id, Child, Type, Modules}]
Types:
SupRef = \textit{sup\_ref()}
Id = \textit{child\_id()} | undefined
Child = \textit{child()} | restarting
Type = \textit{worker()}
Modules = \textit{modules()}

Returns a newly created list with information about all child specifications and child processes belonging to the supervisor \textit{SupRef}.

Note that calling this function when supervising a large number of children under low memory conditions can cause an out of memory exception.

See \textit{start\_child/2} for a description of \textit{SupRef}.

The information given for each child specification/process is:

\begin{itemize}
  \item \textbf{Id} - as defined in the child specification or undefined in the case of a \texttt{simple\_one\_for\_one} supervisor.
  \item \textbf{Child} - the pid of the corresponding child process, the atom \texttt{restarting} if the process is about to be restarted or undefined if there is no such process.
  \item \textbf{Type} - as defined in the child specification.
  \item \textbf{Modules} - as defined in the child specification.
\end{itemize}

count\_children(SupRef) \rightarrow \text{PropListOfCounts}

Types:
\begin{verbatim}
SupRef = \textit{sup\_ref()}
PropListOfCounts = [Count]
    Count = \{specs, ChildSpecCount :: integer() >= 0\}
    | \{active, ActiveProcessCount :: integer() >= 0\}
    | \{supervisors, ChildSupervisorCount :: integer() >= 0\}
    | \{workers, ChildWorkerCount :: integer() >= 0\}
\end{verbatim}

Returns a property list (see \texttt{proplists}) containing the counts for each of the following elements of the supervisor's child specifications and managed processes:

\begin{itemize}
  \item \texttt{specs} - the total count of children, dead or alive.
  \item \texttt{active} - the count of all actively running child processes managed by this supervisor.
  \item \texttt{supervisors} - the count of all children marked as \texttt{child\_type = supervisor} in the spec list, whether or not the child process is still alive.
  \item \texttt{workers} - the count of all children marked as \texttt{child\_type = worker} in the spec list, whether or not the child process is still alive.
\end{itemize}

check\_childs specs(ChildSpecs) \rightarrow \text{Result}

Types:
\begin{verbatim}
ChildSpecs = [\textit{child\_spec()}]
Result = \text{ok} | \{\text{error, Error :: term()}\}
\end{verbatim}

This function takes a list of child specification as argument and returns \texttt{ok} if all of them are syntactically correct, or \{\text{error, Error}:: term()} otherwise.

\section*{CALLBACK FUNCTIONS}

The following functions should be exported from a \texttt{supervisor} callback module.
Exports

Module:init(Args) -> Result
Types:
  Args = term()
  Result = {ok,{{RestartStrategy,MaxR,MaxT},[ChildSpec]}} | ignore
  RestartStrategy = strategy()
  MaxR = integer()>=0
  MaxT = integer()>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), gen_fsm(3), gen_server(3), sys(3)
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**

```erlang
start_link(Module, Args) -> Result
start_link(SupBridgeName, Module, Args) -> Result
```

Types:

```erlang
SupBridgeName = {local, Name} | {global, Name}
Name = atom()
Module = module()
Args = term()
Result = {ok, Pid} | ignore | {error, Error}
Error = {already_started, Pid} | term()
Pid = pid()
```

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 `SupBridgeName={via,Module,Name}` the supervisor_bridge is registered as `Name` using a registry represented by `Module`. The `Module` callback should export the functions `register_name/2`, `unregister_name/1` and `send/2`, which should behave like the corresponding functions in `global`. Thus, `{via,global,GlobalName}` is a valid reference. 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,Errorr}` where `Error` is a term with information about the error, and the supervisor_bridge terminates with reason `Error`.

---

440 | Ericsson AB. All Rights Reserved.: STDLIB
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 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), sys(3)
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 \texttt{gen_server}.

The default timeout is 5000 ms, unless otherwise specified. The \texttt{timeout} defines the time period to wait for the process to respond to a request. If the process does not respond, the function evaluates \texttt{exit({timeout, \{M, F, A\}})}.

The functions make reference to a debug structure. The debug structure is a list of \texttt{dbg_opt()}. \texttt{dbg_opt()} is an internal data type used by the \texttt{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 \texttt{\{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 \texttt{sys:handle_system_msg/6} is called in order to handle the request.

- **Shutdown messages.** If the process traps exits, it must be able to handle an shut-down request from its parent, the supervisor. The message \texttt{\{'EXIT', Parent, Reason\}} from the parent is an order to terminate. The process must terminate when this message is received, normally with the same \texttt{Reason} as \texttt{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 \texttt{gen_event} processes. This message is \texttt{\{get_modules, From\}}. The reply to this message is \texttt{From ! \{modules, Modules\}}, where \texttt{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 \texttt{system_events} which are then treated in the debug function. For example, \texttt{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.

### Data Types

\begin{verbatim}
name() = pid() | atom() | \{global, atom()\} 
system_event() = \{in, Msg :: term()\} 
                   | \{in, Msg :: term(), From :: term()\} 
                   | \{out, Msg :: term(), To :: term()\} 
                   | term() 

dbg_opt() 
See above. 
\end{verbatim}
DBG_FUN() =
    fun((FuncState :: term(),
         Event :: system_event(),
         ProcState :: term()) ->
        done | (NewFuncState :: term())

FORMAT_FUN() =
    fun((Device :: io:device() | file:io_device(),
         Event :: system_event(),
         Extra :: term()) ->
        any())

Exports

log(Name, Flag) -> ok | {ok, [system_event()]}  
log(Name, Flag, Timeout) -> ok | {ok, [system_event()]}  

Types:  
  Name = name()  
  Flag = true | {true, N :: integer() >= 1} | false | get | print  
  Timeout = timeout()

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) -> ok | {error, open_file}
log_to_file(Name, Flag, Timeout) -> ok | {error, open_file}  

Types:  
  Name = name()  
  Flag = (FileName :: string()) | false  
  Timeout = timeout()

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) -> ok | {ok, Statistics}  
statistics(Name, Flag, Timeout) -> ok | {ok, Statistics}  

Types:  
  Name = name()  
  Flag = true | false | get  
  Statistics = [StatisticsTuple] | no_statistics  
  StatisticsTuple = {start_time, DateTime1}  
                  | {current_time, DateTime2}  
                  | {reductions, integer() >= 0}  
                  | {messages_in, integer() >= 0}
| {messages_out, integer() >= 0}
DateTime1 = DateTime2 = `file:datetime()`
Timeout = timeout()

Enables or disables the collection of statistics. If Flag is get, the statistical collection is returned.

trace(Name, Flag) -> ok
trace(Name, Flag, Timeout) -> ok

Types:
Name = `name()`
Flag = boolean()
Timeout = timeout()

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) -> ok
no_debug(Name, Timeout) -> ok

Types:
Name = `name()`
Timeout = timeout()

Turns off all debugging for the process. This includes functions that have been installed explicitly with the install function, for example triggers.

suspend(Name) -> ok
suspend(Name, Timeout) -> ok

Types:
Name = `name()`
Timeout = timeout()

Suspends the process. When the process is suspended, it will only respond to other system messages, but not other messages.

resume(Name) -> ok
resume(Name, Timeout) -> ok

Types:
Name = `name()`
Timeout = timeout()

Resumes a suspended process.

change_code(Name, Module, OldVsn, Extra) -> ok | {error, Reason}
change_code(Name, Module, OldVsn, Extra, Timeout) ->
   ok | {error, Reason}

Types:
Name = \texttt{name()}
Module = \texttt{module()}
OldVsn = \texttt{undefined | term()}
Extra = \texttt{term()}
Timeout = \texttt{timeout()}
Reason = \texttt{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 \texttt{Module:system_code_change/4} is called. OldVsn is the old version of the Module.

get_status(Name) \rightarrow \text{Status}
get_status(Name, Timeout) \rightarrow \text{Status}

Types:
\begin{itemize}
\item Name = \texttt{name()}
\item Timeout = \texttt{timeout()}
\item Status = \{\text{status, Pid :: pid(), \{module, Module :: module()\}, \{SItem\}}\}
\item SItem = (PDict :: [{Key :: term(), Value :: term()}])
| (SysState :: \{running | suspended\})
| (Parent :: \texttt{pid()})
| (Dbg :: \{\texttt{dbg\_opt()}\})
| (Misc :: \texttt{term()})
\end{itemize}

Gets the status of the process.

The value of Misc varies for different types of processes. For example, a \texttt{gen_server} process returns the callback module's state, a \texttt{gen_fsm} process returns information such as its current state name and state data, and a \texttt{gen_event} process returns information about each of its registered handlers. Callback modules for \texttt{gen_server}, \texttt{gen_fsm}, and \texttt{gen_event} can also customise the value of Misc by exporting a \texttt{format_status/2} function that contributes module-specific information; see \texttt{gen_server:format_status/2}, \texttt{gen_fsm:format_status/2}, and \texttt{gen_event:format_status/2} for more details.

get_state(Name) \rightarrow \text{State}
get_state(Name, Timeout) \rightarrow \text{State}

Types:
\begin{itemize}
\item Name = \texttt{name()}
\item Timeout = \texttt{timeout()}
\item State = \texttt{term()}
\end{itemize}

Gets the state of the process.

\textbf{Note:}

These functions are intended only to help with debugging. They are provided for convenience, allowing developers to avoid having to create their own state extraction functions and also avoid having to interactively extract state from the return values of \texttt{get_status/1} or \texttt{get_status/2} while debugging.
The value of State varies for different types of processes. For a gen_server process, the returned State is simply the callback module's state. For a gen_fsm process, State is the tuple {CurrentStateName, CurrentStateData}. For a gen_event process, State a list of tuples, where each tuple corresponds to an event handler registered in the process and contains {Module, Id, HandlerState}, where Module is the event handler's module name, Id is the handler's ID (which is the value false if it was registered without an ID), and HandlerState is the handler's state.

If the callback module exports a system_get_state/1 function, it will be called in the target process to get its state. Its argument is the same as the Misc value returned by get_status/1,2, and the system_get_state/1 function is expected to extract the callback module's state from it. The system_get_state/1 function must return {ok, State} where State is the callback module's state.

If the callback module does not export a system_get_state/1 function, get_state/1,2 assumes the Misc value is the callback module's state and returns it directly instead.

If the callback module's system_get_state/1 function crashes or throws an exception, the caller exits with error {callback_failed, {Module, system_get_state}, {Class, Reason}} where Module is the name of the callback module and Class and Reason indicate details of the exception.

The system_get_state/1 function is primarily useful for user-defined behaviours and modules that implement OTP special processes. The gen_server, gen_fsm, and gen_event OTP behaviour modules export this function, and so callback modules for those behaviours need not supply their own.

To obtain more information about a process, including its state, see get_status/1 and get_status/2.

replace_state(Name, StateFun) -> NewState
replace_state(Name, StateFun, Timeout) -> NewState

Types:
Name = name()
StateFun = fun((State :: term()) -> NewState :: term())
Timeout = timeout()
NewState = term()

Replaces the state of the process, and returns the new state.

Note:
These functions are intended only to help with debugging, and they should not be be called from normal code. They are provided for convenience, allowing developers to avoid having to create their own custom state replacement functions.

The StateFun function provides a new state for the process. The State argument and NewState return value of StateFun vary for different types of processes. For a gen_server process, State is simply the callback module's state, and NewState is a new instance of that state. For a gen_fsm process, State is the tuple {CurrentStateName, CurrentStateData}, and NewState is a similar tuple that may contain a new state name, new state data, or both. For a gen_event process, State is the tuple {Module, Id, HandlerState} where Module is the event handler's module name, Id is the handler's ID (which is the value false if it was registered without an ID), and HandlerState is the handler's state. NewState is a similar tuple where Module and Id shall have the same values as in State but the value of HandlerState may be different. Returning a NewState whose Module or Id values differ from those of State will result in the event handler's state remaining unchanged. For a gen_event process, StateFun is called once for each event handler registered in the gen_event process.
If a StateFun function decides not to effect any change in process state, then regardless of process type, it may simply return its State argument.

If a StateFun function crashes or throws an exception, then for gen_server and gen_fsm processes, the original state of the process is unchanged. For gen_event processes, a crashing or failing StateFun function means that only the state of the particular event handler it was working on when it failed or crashed is unchanged; it can still succeed in changing the states of other event handlers registered in the same gen_event process.

If the callback module exports a system_replace_state/2 function, it will be called in the target process to replace its state using StateFun. Its two arguments are StateFun and Misc, where Misc is the same as the Misc value returned by get_status/1,2. A system_replace_state/2 function is expected to return {ok, NewState, NewMisc} where NewState is the callback module's new state obtained by calling StateFun, and NewMisc is a possibly new value used to replace the original Misc (required since Misc often contains the callback module's state within it).

If the callback module does not export a system_replace_state/2 function, replace_state/2,3 assumes the Misc value is the callback module's state, passes it to StateFun and uses the return value as both the new state and as the new value of Misc.

If the callback module's system_replace_state/2 function crashes or throws an exception, the caller exits with error {callback_failed, {Module, system_replace_state}, {Class, Reason}} where Module is the name of the callback module and Class and Reason indicate details of the exception. If the callback module does not provide a system_replace_state/2 function and StateFun crashes or throws an exception, the caller exits with error {callback_failed, StateFun, {Class, Reason}}.

The system_replace_state/2 function is primarily useful for user-defined behaviours and modules that implement OTP special processes. The gen_server, gen_fsm, and gen_event OTP behaviour modules export this function, and so callback modules for those behaviours need not supply their own.

install(Name, FuncSpec) -> ok
install(Name, FuncSpec, Timeout) -> ok

Types:
  Name = name()
  FuncSpec = {Func, FuncState}
  Func = dbg_fun()
  FuncState = term()
  Timeout = timeout()

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) -> ok
remove(Name, Func, Timeout) -> ok

Types:
  Name = name()
  Func = dbg_fun()
  Timeout = timeout()

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
       | {log, integer() >= 1}
       | statistics
       | {log_to_file, FileName}
       | {install, FuncSpec}
  FileName = file:name()
  FuncSpec = {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(Debug, FormFunc, Extra, Event) -> [dbg_opt()]

Types:
  Debug = [dbg_opt()]
  FormFunc = format_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) -> no_return()

Types:
Msg = term()
From = {pid(), Tag :: term()}
Parent = pid()
Module = module()
Debug = [dbg_opt()]
Misc = term()

This function is used by a process module that wishes to take care of system messages. The process receives a (system, From, Msg) message and passes the Msg and From to this function.

This function never returns. It calls the function Module:system_continue(Parent, NDebug, Misc) where the process continues the execution, or Module:system_terminate(Reason, Parent, Debug, Misc) if the process should terminate. The Module must export system_continue/3, system_terminate/4, system_code_change/4, system_get_state/1 and system_replace_state/2 (see below).

The Misc argument can be used to save internal data in a process, for example its state. It is sent to Module:system_continue/3 or Module:system_terminate/4

print_log(Debug) -> ok
Types:
  Debug = [dbg_opt()]
Prints the logged system events in the debug structure using FormFunc as defined when the event was generated by a call to handle_debug/4.

Mod:system_continue(Parent, Debug, Misc) -> none()
Types:
  Parent = pid()
  Debug = [dbg_opt()]
  Misc = term()
This function is called from sys:handle_system_msg/6 when the process should continue its execution (for example after it has been suspended). This function never returns.

Mod:system_terminate(Reason, Parent, Debug, Misc) -> none()
Types:
  Reason = term()
  Parent = pid()
  Debug = [dbg_opt()]
  Misc = term()
This function is called from sys:handle_system_msg/6 when the process should terminate. For example, this function is called when the process is suspended and its parent orders shut-down. It gives the process a chance to do a clean-up. This function never returns.

Mod:system_code_change(Misc, Module, OldVsn, Extra) -> {ok, NMisc}
Types:
  Misc = term()
  OldVsn = undefined | term()
Module = atom()
Extra = term()
NMisc = term()

Called from `sys:handle_system_msg/6` when the process should perform a code change. The code change is used when the internal data structure has changed. This function converts the Misc argument to the new data structure. OldVsn is the `vsn` attribute of the old version of the Module. If no such attribute was defined, the atom `undefined` is sent.

```erlang
Mod:system_get_state(Misc) -> {ok, State}.
```

Types:
- Misc = term()
- State = term()

This function is called from `sys:handle_system_msg/6` when the process should return a term that reflects its current state. State is the value returned by `sys:get_state/2`.

```erlang
Mod:system_replace_state(StateFun, Misc) -> {ok, NState, NMisc}.
```

Types:
- StateFun = fun((State :: term()) -> NState)
- Misc = term()
- NState = term()
- NMisc = term()

This function is called from `sys:handle_system_msg/6` when the process should replace its current state. NState is the value returned by `sys:replace_state/3`.
This module provides useful functions related to time. Unless otherwise stated, time is always measured in milliseconds. All timer functions return immediately, regardless of work carried out by another process.

Successful evaluations of the timer functions yield return values containing a timer reference, denoted TRef below. By using cancel/1, the returned reference can be used to cancel any requested action. A TRef is an Erlang term, the contents of which must not be altered.

The timeouts are not exact, but should be at least as long as requested.

**Data Types**

- time() = integer() >= 0
  - Time in milliseconds.
- tref()
  - A timer reference.

**Exports**

- start() -> ok
  - Starts the timer server. Normally, the server does not need to be started explicitly. It is started dynamically if it is needed. This is useful during development, but in a target system the server should be started explicitly. Use configuration parameters for kernel for this.

- apply_after(Time, Module, Function, Arguments) -> {ok, TRef} | {error, Reason}
  - Types:
    - Time = time()
    - Module = module()
    - Function = atom()
    - Arguments = [term()]
    - TRef = tref()
    - Reason = term()
  - Evaluates apply(Module, Function, Arguments) after Time amount of time has elapsed. Returns {ok, TRef}, or {error, Reason}.

- send_after(Time, Message) -> {ok, TRef} | {error, Reason}
- send_after(Time, Pid, Message) -> {ok, TRef} | {error, Reason}
  - Types:
timer

Time = time()
Pid = pid() | (RegName :: atom())
Message = term()
TRef = tref()
Reason = term()

send_after/3
    Evaluates Pid ! Message after Time amount of time has elapsed. (Pid can also be an atom of a registered name.) Returns {ok, TRef}, or {error, Reason}.

send_after/2
    Same as send_after(Time, self(), Message).

kill_after(Time) -> {ok, TRef} | {error, Reason2}
kill_after(Time, Pid) -> {ok, TRef} | {error, Reason2}
exit_after(Time, Reason1) -> {ok, TRef} | {error, Reason2}
exit_after(Time, Pid, Reason1) -> {ok, TRef} | {error, Reason2}

Types:
  Time = time()
  Pid = pid() | (RegName :: atom())
  TRef = tref()
  Reason1 = Reason2 = term()

exit_after/3
    Send an exit signal with reason Reason1 to 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 = time()
  Module = module()
  Function = atom()
  Arguments = [term()]
  TRef = tref()
  Reason = term()

Evaluates apply(Module, Function, Arguments) repeatedly at intervals of Time. Returns {ok, TRef}, or {error, Reason}. 
send_interval(Time, Message) -> {ok, TRef} | {error, Reason}
send_interval(Time, Pid, Message) -> {ok, TRef} | {error, Reason}

Types:
  Time = time()
  Pid = pid() | (RegName :: atom())
  Message = term()
  TRef = tref()
  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}

Types:
  TRef = tref()
  Reason = term()

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 = timeout()

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(Fun) -> {Time, Value}
tc(Fun, Arguments) -> {Time, Value}
tc(Module, Function, Arguments) -> {Time, Value}

Types:
  Module = module()
  Function = atom()
  Arguments = [term()]
  Time = integer()
  In microseconds
  Value = term()

tc/3
  Evaluates apply(Module, Function, Arguments) and measures the elapsed real time as reported by os:timestamp/0. Returns {Time, Value}, where Time is the elapsed real time in microseconds, and Value is what is returned from the apply.

tc/2
  Evaluates apply(Fun, Arguments). Otherwise works like tc/3.
tc/1
  Evaluates Fun(). Otherwise works like tc/2.

now_diff(T2, T1) -> Tdiff
Types:
  T1 = T2 = erlang:timestamp()
  Tdiff = integer()
In microseconds
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
Types:
  Seconds = MilliSeconds = integer() >= 0
Returns the number of milliseconds in Seconds.

minutes(Minutes) -> MilliSeconds
Types:
  Minutes = MilliSeconds = integer() >= 0
Return the number of milliseconds in Minutes.

hours(Hours) -> MilliSeconds
Types:
  Hours = MilliSeconds = integer() >= 0
Returns the number of milliseconds in Hours.

hms(Hours, Minutes, Seconds) -> MilliSeconds
Types:
  Hours = Minutes = Seconds = MilliSeconds = integer() >= 0
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, ["\n\nHello World!\n", []]).
      {ok,TRef}
      Hello World!

The following coding example illustrates a process which performs a certain action and if this action is not completed
within a certain limit, then the process is killed.

    Pid = spawn(mod, fun, [foo, bar]),
%% If pid is not finished in 10 seconds, kill him
{ok, R} = timer:kill_after(timer:seconds(10), Pid),
...
%% We change our mind...
timer:cancel(R),
...

WARNING

A timer can always be removed by calling cancel/1.

An interval timer, i.e. a timer created by evaluating any of the functions apply_interval/4,
send_interval/3, and send_interval/2, is linked to the process towards which the timer performs its task.

A one-shot timer, i.e. a timer created by evaluating any of the functions apply_after/4, send_after/3,
send_after/2, exit_after/3, exit_after/2, kill_after/2, and kill_after/1 is not linked to
any process. Hence, such a timer is removed only when it reaches its timeout, or if it is explicitly removed by a call
to cancel/1.

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

- `encoding() = latin1 | unicode | utf8 | utf16 | {utf16, endian()} | utf32 | {utf32, endian()}`
- `endian() = big | little`
- `unicode_binary() = binary()` A binary( ) with characters encoded in the UTF-8 coding standard.
- `chardata() = charlist() | unicode_binary()`
- `charlist() = maybe_improper_list(char() | unicode_binary() | charlist(), unicode_binary() | `)
- `external_unicode_binary() = binary()` A binary( ) with characters coded in a user specified Unicode encoding other than UTF-8 (UTF-16 or UTF-32).
- `external_chardata() = external_charlist() | external_unicode_binary()`
- `external_charlist() = maybe_improper_list(char() | external_unicode_binary() | external_charlist(), external_unicode_binary() | `)
- `latin1_binary() = binary()` A binary( ) with characters coded in ISO-latin-1.
- `latin1_char() = byte()` An integer( ) representing valid latin1 character (0-255).
latin1_chardata() = latin1_charlist() | latin1_binary()
The same as iodata().

latin1_charlist() =
    maybe_improper_list(latin1_char() |
    latin1_binary() |
    latin1_charlist(),
    latin1_binary() | [])
The same as iolist().

Exports

bom_to_encoding(Bin) -> {Encoding, Length}
Types:
    Bin = binary()
        A binary() such that byte_size(Bin) >= 4.
    Encoding = latin1
        | utf8
        | {utf16, endian()}
        | {utf32, endian()}
    Length = integer() >= 0
    endian() = big | little

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) -> Result
Types:
    Data = latin1_chardata() | chardata() | external_chardata()
    Result = list()
        | {error, list(), RestData}
        | {incomplete, list(), binary()}
    RestData = latin1_chardata() | chardata() | external_chardata()

Same as characters_to_list(Data, unicode).

characters_to_list(Data, InEncoding) -> Result
Types:
    Data = latin1_chardata() | chardata() | external_chardata()
    InEncoding = encoding()
    Result = list()
        | {error, list(), RestData}
        | {incomplete, list(), binary()}
    RestData = latin1_chardata() | chardata() | external_chardata()

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
The 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 `idata()` 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 `idata()`.

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` 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 is 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, an error occurs whenever an integer
  - greater than `16#10FFFF` (the maximum Unicode character),
  - in the range `16#D800` to `16#DFFF` (invalid range reserved for UTF-16 surrogate pairs) 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);
```

458 | Ericsson AB. All Rights Reserved.: STDLIB
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 do not contain whole byte (bit-strings), a badarg exception is thrown.

characters_to_binary(Data) -> Result
Types:
  Data = latin1_chardata() | chardata() | external_chardata()
  Result = binary()
    | {error, binary(), RestData}
    | {incomplete, binary(), binary()}
  RestData = latin1_chardata() | chardata() | external_chardata()
Same as characters_to_binary(Data, unicode, unicode).

characters_to_binary(Data, InEncoding) -> Result
Types:
  Data = latin1_chardata() | chardata() | external_chardata()
  InEncoding = encoding()
  Result = binary()
    | {error, binary(), RestData}
    | {incomplete, binary(), binary()}
  RestData = latin1_chardata() | chardata() | external_chardata()
Same as characters_to_binary(Data, InEncoding, unicode).

characters_to_binary(Data, InEncoding, OutEncoding) -> Result
Types:
  Data = latin1_chardata() | chardata() | external_chardata()
  InEncoding = OutEncoding = encoding()
  Result = binary()
    | {error, binary(), RestData}
    | {incomplete, binary(), binary()}
  RestData = latin1_chardata() | chardata() | external_chardata()
Behaves as characters_to_list/2, but produces an 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, but the second element in the error or incomplete tuple will be a binary() and not a list().
encoding_to_bom(InEncoding) -> Bin

Types:

- Bin = binary()
  A binary() such that byte_size(Bin) >= 4.
- InEncoding = encoding()

Create a 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 as 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 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>hkr</td>
<td>HKEY_CLASSES_ROOT</td>
</tr>
<tr>
<td>current_user</td>
<td>HKEY_CURRENT_USER</td>
</tr>
<tr>
<td>hku</td>
<td>HKEY_CURRENT_USER</td>
</tr>
<tr>
<td>local_machine</td>
<td>HKEY_LOCAL_MACHINE</td>
</tr>
<tr>
<td>hklm</td>
<td>HKEY_LOCAL_MACHINE</td>
</tr>
<tr>
<td>users</td>
<td>HKEY_USERS</td>
</tr>
<tr>
<td>hku</td>
<td>HKEY_USERS</td>
</tr>
<tr>
<td>current_config</td>
<td>HKEY_CURRENT_CONFIG</td>
</tr>
<tr>
<td>hkcc</td>
<td>HKEY_CURRENT_CONFIG</td>
</tr>
<tr>
<td>dyn_data</td>
<td>HKEY_DYN_DATA</td>
</tr>
<tr>
<td>hkdd</td>
<td>HKEY_DYN_DATA</td>
</tr>
</tbody>
</table>

The key above could be written as "\\hklm\software\ericsson\erlang\5.0".

The win32reg module uses a current key. It works much like the current directory. From the current key, values can be fetched, sub-keys can be listed, and so on.

Under a key, any number of named values can be stored. They have name, and types, and data.

Currently, the win32reg module supports storing only the following types: REG_DWORD, which is an integer, REG_SZ which is a string and REG_BINARY which is a binary. Other types can be read, and will be returned as binaries.

There is also a "default" value, which has the empty string as name. It is read and written with the atom default instead of the name.

Some registry values are stored as strings with references to environment variables, e.g. "%SystemRoot %Windows". SystemRoot is an environment variable, and should be replaced with its value. A function expand/1 is provided, so that environment variables surrounded in % can be expanded to their values.

For additional information on the Windows registry consult the Win32 Programmer's Reference.
Data Types

reg_handle()
As returned by open/1.

name() = string() | default
value() = string() | integer() | binary()

Exports

change_key(RegHandle, Key) -> ReturnValue
Types:
    RegHandle = reg_handle()
    Key = string()
    ReturnValue = ok | {error, ErrorId :: atom()}
Changes the current key to another key. Works like cd. The key can be specified as a relative path or as an absolute path, starting with \\.

change_key_create(RegHandle, Key) -> ReturnValue
Types:
    RegHandle = reg_handle()
    Key = string()
    ReturnValue = ok | {error, ErrorId :: atom()}
Creates a key, or just changes to it, if it is already there. Works like a combination of mkdir and cd. Calls the Win32 API function RegCreateKeyEx().
The registry must have been opened in write-mode.

close(RegHandle) -> ok
Types:
    RegHandle = reg_handle()
Closes the registry. After that, the RegHandle cannot be used.

current_key(RegHandle) -> ReturnValue
Types:
    RegHandle = reg_handle()
    ReturnValue = {ok, string()}
Returns the path to the current key. This is the equivalent of pwd.
Note that the current key is stored in the driver, and might be invalid (e.g. if the key has been removed).

delete_key(RegHandle) -> ReturnValue
Types:
    RegHandle = reg_handle()
    ReturnValue = ok | {error, ErrorId :: atom()}
Deletes the current key, if it is valid. Calls the Win32 API function RegDeleteKey(). Note that this call does not change the current key. (unlike change_key_create/2.) This means that after the call, the current key is invalid.
delete_value(RegHandle, Name) -> ReturnValue
Types:
  RegHandle = reg_handle()
  Name = name()
  ReturnValue = ok | {error, ErrorId :: atom()}
Deletes a named value on the current key. The atom default is used for the the default value.
The registry must have been opened in write-mode.

expand(String) -> ExpandedString
Types:
  String = ExpandedString = string()
Expands a string containing environment variables between percent characters. Anything between two % is taken for
a environment variable, and is replaced by the value. Two consecutive % is replaced by one %.
A variable name that is not in the environment, will result in an error.

format_error(ErrorId) -> ErrorString
Types:
  ErrorId = atom()
  ErrorString = string()
Convert an POSIX errorcode to a string (by calling erl_posix_msg:message).

open(OpenModeList) -> ReturnValue
Types:
  OpenModeList = [OpenMode]
  OpenMode = read | write
  ReturnValue = {ok, RegHandle} | {error, ErrorId :: enotsup}
  RegHandle = reg_handle()
Opens the registry for reading or writing. The current key will be the root (HKEY_CLASSES_ROOT). The read flag
in the mode list can be omitted.
Use change_key/2 with an absolute path after open.

set_value(RegHandle, Name, Value) -> ReturnValue
Types:
  RegHandle = reg_handle()
  Name = name()
  Value = value()
  ReturnValue = ok | {error, ErrorId :: atom()}
Sets the named (or default) value to value. Calls the Win32 API function RegSetValueEx(). The value can be
of three types, and the corresponding registry type will be used. Currently the types supported are: REG_DWORD for
integers, REG_SZ for strings and REG_BINARY for binaries. Other types cannot currently be added or changed.
The registry must have been opened in write-mode.
sub_keys(RegHandle) -> ReturnValue

Types:
  RegHandle = reg_handle()
  ReturnValue = {ok, [SubKey]} | {error, ErrorId :: atom()}
  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:
  RegHandle = reg_handle()
  Name = name()
  ReturnValue = {ok, Value :: value()} | {error, ErrorId :: atom()}

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:
  RegHandle = reg_handle()
  ReturnValue = {ok, [ValuePair]} | {error, ErrorId :: atom()}
  ValuePair = {Name :: name(), Value :: value()}

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 extracts 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 are 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 or the zip/3 function. (They are also available as create, to resemble the erl_tar module.)

To extract files from a zip archive, use the unzip/1 or the unzip/2 function. (They are also available as extract.)

To fold a function over all files in a zip archive, use the foldl_3 function.

To return a list of the files in a zip archive, use the list_dir/1 or the list_dir/2 function. (They are also available as table.)

To print a list of files to the Erlang shell, use either the t/1 or tt/1 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, zip_get, zip_list_dir and zip_close 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_comment() = #zip_comment{comment = undefined | string()}

The record zip_comment just contains the archive comment for a zip archive.

zip_file() =
  #zip_file{name = undefined | string(),
          info = undefined | file:file_info(),
          comment = undefined | string(),
          offset = undefined | integer() >= 0,
          comp_size = undefined | integer() >= 0}

The record zip_file contains the following fields.

name

the name of the file
info
  file info as in file:read_file_info/1

comment
  the comment for the file in the zip archive

offset
  the offset of the file in the zip archive (used internally)

comp_size
  the compressed size of the file (the uncompressed size is found in info)

filename() = file:filename()
extension() = string()
extension_spec() = all
  [extension()]
  [add, [extension()]]
  [del, [extension()]]
create_option() = memory
  cooked
  verbose
  [comment, string()]
  [cwd, file:filename()]
  [compress, extension_spec()]
  [uncompress, extension_spec()]

These options are described in create/3.

Exports

zip(Name, FileList) -> RetValue
zip(Name, FileList, Options) -> RetValue
create(Name, FileList) -> RetValue
create(Name, FileList, Options) -> RetValue

Types:
  Name = file:name()
  FileList = [FileSpec]
  FileSpec = file:name()
  | {file:name(), binary()}
  | {file:name(), binary(), file:file_info()}
  Options = [Option]
  Option = create_option()
  RetValue = {ok, FileName :: filename()}
  | {ok, {FileName :: filename(), binary()}}
  | {error, 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 functions check 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 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 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]}

deleter these extensions from the list of uncompress extensions.

unzip(Archive) -> RetValue
unzip(Archive, Options) -> RetValue
extract(Archive) -> RetValue
extract(Archive, Options) -> RetValue

Types:
  Archive = file:name() | binary()
  Options = [Option]
  Option = {file_list, FileList}
          | keep_old_files
          | verbose
          | memory
          | {file_filter, FileFilter}
          | {cwd, CWD}
  FileList = [file:name()]
  FileBinList = [{{file:name()}, binary()}]
  FileFilter = fun((ZipFile) -> boolean())
  CWD = file:filename()
  ZipFile = zip_file()
  RetValue = {ok, FileList}
          | {ok, FileBinList}
          | {error, Reason :: term()}
          | {error, {Name :: file:name(), Reason :: term()}}

The unzip/1 function extracts all files from a zip archive. The unzip/2 function provides options to extract some files, and more.

If the Archive argument is given as a binary, the contents of the binary is assumed to be a zip archive, otherwise it should be a filename.

The following options are available:

{file_list, FileList}

By default, all files will be extracted from the zip archive. With the {file_list, FileList} option, the unzip/2 function will only extract the files whose names are included in FileList. The full paths, including the names of all sub directories within the zip archive, must be specified.

cooked

By default, the open/2 function will open the zip file in raw mode, which is faster but does not allow a remote (erlang) file server to be used. Adding cooked to the mode list will override the default and open the 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:set_cwd/1, but without changing the global cwd property.)

foldl(Fun, Acc0, Archive) -> {ok, Acc1} | {error, Reason}

Types:

Fun = fun((FileInArchive, GetInfo, GetBin, AccIn) -> AccOut)
FileInArchive = file:name()
GetInfo = fun(() -> file:file_info())
GetBin = fun(() -> binary())
Acc0 = Acc1 = AccIn = AccOut = term()
Archive = file:name() | {file:name(), binary()}
Reason = term()

The foldl/3 function calls Fun(FileInArchive, GetInfo, GetBin, AccIn) on successive files in the Archive, starting with AccIn == Acc0. FileInArchive is the name that the file has in the archive. GetInfo is a fun that returns info about the the file. GetBin returns the contents of the file. Both GetInfo and GetBin must be called within the Fun. Their behavior is undefined if they are called outside the context of the Fun. The Fun must return a new accumulator which is passed to the next call. foldl/3 returns the final value of the accumulator. Acc0 is returned if the archive is empty. It is not necessary to iterate over all files in the archive. The iteration may be ended prematurely in a controlled manner by throwing an exception.

For example:

```erlang
> Name = "dummy.zip".
"dummy.zip"
> {ok, {Name, Bin}} = zip:create(Name, ["foo", <<"FOO">>], ["bar", <<"BAR">>], [memory]).
{ok,"dummy.zip",
  <<80,75,3,4,20,8,0,8,0,8,74,152,97,60,171,39,212,26,3,0,0,0,3,0,0,0,0,0,...>>}
> {ok, FileSpec} = zip:foldl(fun(N, I, B, Acc) -> [{N, B(), I()} | Acc] end, [], {Name, Bin}).
ok,{["bar",<<"BAR">>],
  {file_info,3,regular,read_write,
   {{2010,3,1},{19,2,10}},
   {{2010,3,1},{19,2,10}},
   {{2010,3,1},{19,2,10}},
   54,1,0,0,0,0,0}},
  {"foo",<<"FOO">>},
  {file_info,3,regular,read_write,
   {{2010,3,1},{19,2,10}},
   {{2010,3,1},{19,2,10}},
   {{2010,3,1},{19,2,10}},
   54,1,0,0,0,0,0}}}
```
list_dir(Archive) -> RetValue
list_dir(Archive, Options) -> RetValue

table(Archive) -> RetValue
table(Archive, Options) -> RetValue

Types:
  Archive = file:name() | binary()
  RetValue = {ok, CommentAndFiles} | {error, Reason :: term()}
  CommentAndFiles = [zip_comment() | zip_file()]
  Options = [Option]
  Option = cooked

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 the zip file without the raw option.

t(Archive) -> ok

types:
  Archive = file:name() | binary() | ZipHandle
  ZipHandle = pid()

The t/1 function prints the names of all files in the zip archive Archive to the Erlang shell. (Similar to "tar t").

tt(Archive) -> ok

types:
  Archive = file:name() | binary() | ZipHandle
  ZipHandle = pid()

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 = file:name() | binary()
   ZipHandle = pid()
   Options = [Option]
   Option = cooked | memory | {cwd, CWD :: file:filename()}
   Reason = term()

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) -> {ok, Result} | {error, Reason}

Types:
   Result = [zip_comment() | zip_file()]
   ZipHandle = pid()
   Reason = term()

The `zip_list_dir/1` function returns the file list of an open zip archive. The first returned element is the zip archive comment.

zip_get(ZipHandle) -> {ok, [Result]} | {error, Reason}
zip_get(FileName, ZipHandle) -> {ok, Result} | {error, Reason}

Types:
   FileName = file:name()
   ZipHandle = pid()
   Result = file:name() | {file:name(), binary()}
   Reason = term()

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.