

STDLIB

Copyright © 1997-2018 Ericsson AB. All Rights Reserved.
STDLIB 3.4.5
April 17, 2018

Copyright © 1997-2018 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

April 17, 2018

1 STDLIB User's Guide

1.1 Introduction

1.1.1 Scope

The Standard Erlang Libraries application, **STDLIB**, is mandatory in the sense that the minimal system based on Erlang/OTP consists of **STDLIB** and **Kernel**.

STDLIB contains the following functional areas:

- Erlang shell
- Command interface
- Query interface
- Interface to standard Erlang I/O servers
- Interface to the Erlang built-in term storage BIFs
- Regular expression matching functions for strings and binaries
- Finite state machine
- Event handling
- Functions for the server of a client-server relation
- Function to control applications in a distributed manner
- Start and control of slave nodes
- Operations on finite sets and relations represented as sets
- Library for handling binary data
- Disk-based term storage
- List processing
- Maps processing

1.1.2 Prerequisites

It is assumed that the reader is familiar with the Erlang programming language.

1.2 The Erlang I/O Protocol

The I/O protocol in Erlang enables bi-directional communication between clients and servers.

- The I/O server is a process that handles the requests and performs the requested task on, for example, an I/O device.
- The client is any Erlang process wishing to read or write data from/to the I/O device.

The common I/O protocol has been present in OTP since the beginning, but has been undocumented and has also evolved over the years. In an addendum to Robert Virding's rationale, the original I/O protocol is described. This section describes the current I/O protocol.

The original I/O protocol was simple and flexible. Demands for memory efficiency and execution time efficiency have 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 section 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 the client code does not need to be aware of the I/O device that the I/O server communicates with.

1.2.1 Protocol Basics

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

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

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

- `From` is the `pid()` of the client, the process which the I/O server sends the I/O reply to.
- `ReplyAs` can be any datum and is returned in the corresponding `io_reply`. The *`io`* module monitors the the I/O server and uses the monitor reference as the `ReplyAs` datum. A more complicated client can have many outstanding I/O requests to the same I/O server and can use different references (or something else) to differentiate among the incoming I/O replies. Element `ReplyAs` is to be considered opaque by the I/O server.

  Notice that the `pid()` of the I/O server is not explicitly present in tuple `io_reply`. The reply can be sent from any process, not necessarily the actual I/O server.
- `Request` and `Reply` are described below.

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

1.2.2 Output Requests

To output characters on an I/O device, the following `Request`s exist:

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

- `Encoding` is `unicode` or `latin1`, meaning that the characters are (in case of binaries) encoded as UTF-8 or ISO Latin-1 (pure bytes). A well-behaved I/O server is also to return an error indication if list elements contain integers > 255 when `Encoding` is set to `latin1`.

  Notice that this does not in any way tell how characters are to be put on the I/O device or handled by the I/O server. Different I/O servers can handle the characters however they want, this only 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.

  Notice also that byte-oriented data is simplest sent using the ISO Latin-1 encoding.
- `Characters` are the data to be put on the I/O 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 that is called to produce the data (like *`io_lib:format/2`*).

  `Args` is a list of arguments to the function. The function is to produce data in the specified `Encoding`. The I/O server is to call the function as `apply(Mod, Func, Args)` and put the returned data on the I/O 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 is to be sent back to the client.

The I/O server replies to the client with an `io_reply` tuple, where element `Reply` is one of:

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

- `Error` describes the error to the client, which can do whatever it wants with it. The *`io`* module typically returns it "as is".

For backward compatibility, the following `Request`s are also to be handled by an I/O server (they are not to be present after Erlang/OTP R15B):

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

These are to behave as `{put_chars, latin1, Characters}` and `{put_chars, latin1, Module, Function, Args}`, respectively.

1.2.3 Input Requests

To read characters from an I/O device, the following `Request`s 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 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 always sees lists of integers, never binaries, but the list can contain numbers > 255 if `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 I/O device. `Prompt` is often ignored by the I/O server; if set to `''`, it is always to be ignored (and results in nothing being written to the I/O device).
- `Module`, `Function`, and `ExtraArgs` denote a function and arguments to determine when enough data is written. The function is to take two more arguments, the last state, and a list of characters. The function is to return one of:

  ```
  {done, Result, RestChars}
  {more, Continuation}
  ```

  `Result` can be any Erlang term, but if it is a `list()`, the I/O server can 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 is called with the data the I/O server finds on its I/O device, returning one of:

  - `{done, Result, RestChars}` when enough data is read. In this case `Result` is sent to the client and `RestChars` is kept in the I/O server as a buffer for later input.
  - `{more, Continuation}`, which indicates that more characters are needed to complete the request.

  `Continuation` is sent as the state in later calls to the function when more characters are available. When no more characters are available, the function must return `{done, eof, Rest}`. The initial state is the empty list. The data when an end of file is reached on the IO device is the atom `eof`.

  An emulation of the `get_line` request can be (inefficiently) implemented using the following functions:

```
-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.
```

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

A fixed number of characters is requested using the following `Request`:

```
{get_chars, Encoding, Prompt, N}
```

- `Encoding` and `Prompt` as for `get_until`.
- `N` is the number of characters to be read from the I/O device.

A single line (as in former example) is requested with the following `Request`:

```
{get_line, Encoding, Prompt}
```

- `Encoding` and `Prompt` as for `get_until`.

Clearly, `get_chars` and `get_line` could be implemented with the `get_until` request (and indeed they were originally), but demands for efficiency have made these additions necessary.

The I/O server replies to the client with an `io_reply` tuple, where element `Reply` is one of:

```
Data
eof
{error, Error}
```

- `Data` is the characters read, in list or binary form (depending on the I/O server mode, see the next section).
- `eof` is returned when input end is reached and no more data is available to the client process.
- `Error` describes the error to the client, which can do whatever it wants with it. The *`io`* module typically returns it as is.

For backward compatibility, the following `Request`s are also to be handled by an I/O server (they are not to be present after Erlang/OTP R15B):

```
{get_until, Prompt, Module, Function, ExtraArgs}
{get_chars, Prompt, N}
{get_line, Prompt}
```

These are to behave as `{get_until, latin1, Prompt, Module, Function, ExtraArgs}`, `{get_chars, latin1, Prompt, N}`, and `{get_line, latin1, Prompt}`, respectively.

1.2.4 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 element `Data` 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 is sent in the standard Erlang Unicode format, that is, UTF-8 (notice that the function of the `get_until` request still gets list data regardless of the I/O server mode).

Notice that 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/2,3`* request is sent to an I/O server). The client must be prepared for data received as answers to those requests to be in various forms. However, the I/O server is to convert the results to binaries whenever possible (that is, when the function supplied to `get_until` returns a list). This is done in the example in section *An Annotated and Working Example I/O Server*.

An I/O server in binary mode affects the data sent to the client, so that it must be able to handle binary data. For convenience, the modes of an I/O server can be set and retrieved using the following I/O requests:

```
{setopts, Opts}
```

- `Opts` is a list of options in the format recognized by the *`proplists`* module (and by the I/O server).

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 unicode/latin1)
```

Options `binary` and `encoding` are common for all I/O servers in OTP, while `echo` and `expand` are valid only for this I/O server. Option `unicode` notifies how characters are put on the physical I/O device, that is, if the terminal itself 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 is to send one of the following as `Reply`:

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

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

To retrieve options, the following request is used:

```
getopts
```

This 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:

```
OptList
{error, Error}
```

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

1.2.5 Multiple I/O Requests

The `Request` element can in itself contain many `Request`s by using the following format:

```
{requests, Requests}
```

- `Requests` is a list of valid `io_request` tuples for the protocol. They must be executed in the order that they appear in the list. The execution is to continue until one of the requests results 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 following valid results in the reply, depending on the requests in the list:

```
ok
{ok, Data}
{ok, Options}
{error, Error}
```

1.2.6 Optional I/O Request

The following I/O request is optional to implement and a client is to be prepared for an error return:

```
{get_geometry, Geometry}
```

- `Geometry` is the atom `rows` or the atom `columns`.

The I/O server is to send the `Reply` as:

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

- `N` is the number of character rows or columns that the I/O device has, if applicable to the I/O device handled by the I/O server, otherwise `{error, enotsup}` is a good answer.

1.2.7 Unimplemented Request Types

If an I/O server encounters a request that it does not recognize (that is, the `io_request` tuple has the expected format, but the `Request` is unknown), the I/O server is to send a valid reply with the error tuple:

```
{error, request}
```

This makes it possible to extend the protocol with optional requests and for the clients to be somewhat backward compatible.

1.2.8 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, a process handling incoming requests, usually both I/O-requests and other I/O device-specific requests (positioning, closing, and so on).

The example I/O server stores characters in an ETS table, making up a fairly crude RAM file.

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

```
-module(ets_io_server).

-export([start_link/0, init/0, loop/1, until_newline/3, until_enough/3]).

-define(CHARS_PER_REC, 10).

-record(state, {
   table,
   position, % absolute
   mode % binary | list
  }).

start_link() ->
    spawn_link(?MODULE,init,[]).

init() ->
    Table = ets:new(noname,[ordered_set]),
    ?MODULE:loop(#state{table = Table, position = 0, mode=list}).

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.
```

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

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 I/O 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 can confuse the reader.

First, we examine the reply function:

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

It sends the `io_reply` tuple back to the client, providing element `ReplyAs` received in the request along with the result of the request, as described earlier.

We need to handle some requests. First the requests for writing characters:

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

The `Encoding` says 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 function *`unicode:characters_to_list/2`*. The conversion function conveniently accepts the encoding types `unicode` and `latin1`, so we can use `Encoding` directly.

When `Module`, `Function`, and `Arguments` are provided, we apply it and do the same with the result as if the data was provided directly.

We handle the requests for retrieving data:

```
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 can be inefficient, but that depends on the frequency of the different requests. Before we start implementing functions `put_chars/2` and `get_until/5`, we examine the few remaining requests:

```
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});
```

Request `get_geometry` has no meaning for this I/O server, so the reply is `{error, enotsup}`. The only option we handle is `binary`/`list`, 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.

We need to handle backward compatibility and the *`file`* module (which uses the old requests until backward compatibility with pre-R13 nodes is no longer needed). Notice that the I/O server does 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);
```

`{error, request}` must be returned if the request is not recognized:

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

Next we 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 function `io_reply`).

Requests `getopts` and `setopts` are also simple to handle. We only change or read the 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;
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.

Request `getopts` is to return a list of `{Option, Value}` tuples. This 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 this I/O server is fairly generic (except for request `rewind` handled in the main loop and the creation of an ETS table). Most I/O servers contain code similar to this one.

To make the example runnable, we start implementing the reading and writing of the data to/from the ETS table. First function `put_chars/3`:

```
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 only split the list in runs of a predefined size and put each run in the table at the current position (and forward). Functions `split_data/3` and `apply_update/2` are implemented below.

Now we want to read data from the table. Function `get_until/5` reads data and applies the function until it says that 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;
 {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 request `setopts`. By default, all OTP I/O servers send data back to the client as lists, but switching mode to `binary` can increase efficiency if the I/O server handles it in an appropriate way. The implementation of `get_until` is difficult 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. However, this example does not optimize anything.

It is important though that the returned data is of the correct type depending on the options set. We therefore convert the lists to binaries in the correct encoding **if possible** before returning. The function supplied in the `get_until` request tuple can, as its final result return anything, so only functions returning lists can get them converted to binaries.

If the request contains encoding tag `unicode`, the lists can contain all Unicode code points and the binaries are to be in UTF-8. If the encoding tag is `latin1`, the client is only to get characters in the range `0..255`. Function `check/2` takes care of not returning arbitrary Unicode code points in lists if the encoding was specified as `latin1`. If the function does not return a list, the check cannot be performed and the result is that of the supplied function untouched.

To manipulate the table we implement the following utility functions:

```
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 provides an error tuple if Unicode code points > 255 are to be returned if the client requested `latin1`.

The two functions `until_newline/3` and `until_enough/3` are helpers used together with function `get_until/5` to implement `get_chars` and `get_line` (inefficiently):

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

until_enough([],eof,_N) ->
    {done,eof,[]};
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 are to be provided in `get_until` requests.

To complete the I/O server, we only need to read and write the table in an appropriate way:

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

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

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

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

The table is read or written in chunks of `?CHARS_PER_REC`, overwriting when necessary. The implementation is clearly 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, for example, the *`io`* module or even the *`file`* module. It is as simple as that to implement a fully fledged I/O server in Erlang.

1.3 Using Unicode in Erlang

1.3.1 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 specified a default encoding in binaries that all Unicode-aware modules are to handle in the future.

Here is an overview what has been done so far:

- The functionality described in EEP10 was implemented in Erlang/OTP R13A.
- Erlang/OTP R14B01 added support for Unicode filenames, but it was not complete and was by default disabled on platforms where no guarantee was given for the filename encoding.
- With Erlang/OTP R16A came support for UTF-8 encoded source code, with enhancements to many of the applications to support both Unicode encoded filenames and support for UTF-8 encoded files in many

circumstances. Most notable is the support for UTF-8 in files read by *`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.
- In Erlang/OTP 20.0, atoms and function can contain Unicode characters. Module names, application names, and node names are still restricted to the ISO Latin-1 range.

  Support was added for normalizations forms in `unicode` and the `string` module now handles utf8-encoded binaries.

This section outlines the current Unicode support and gives some recipes for working with Unicode data.

1.3.2 Understanding Unicode

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

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

As an example, contemplate the issue of converting between upper and lower case letters. Reading the standard makes you realize that there is not a simple one to one mapping in all scripts, for example:

- In German, the letter "ß" (sharp s) is in lower case, but the uppercase equivalent is "SS".
- In Greek, the letter "#" has two different lowercase forms, "#" in word-final position and "#" elsewhere.
- In Turkish, both dotted and dotless "i" exist in lower case and upper case forms.
- Cyrillic "I" has usually no lowercase form.
- Languages with no concept of upper case (or lower case).

So, a conversion function must know not only one character at a time, but possibly the whole sentence, the natural language to translate to, the differences in input and output string length, and so on. Erlang/OTP has currently no Unicode `uppercase`/`lowercase` functionality with language specific handling, but publicly available libraries address these issues.

Another example is the accented characters, where the same glyph has two different representations. The Swedish letter "ö" is one example. The Unicode standard has a code point for it, but you can also write it as "o" followed by "U+0308" (Combining Diaeresis, with the simplified meaning that the last letter is to have "¨" above). They have the same glyph, user perceived character. They are for most purposes the same, but have different representations. For example, MacOS X converts all filenames 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 is done, it is usually important to normalize such characters to avoid confusion.

The list of examples can be made long. One need a kind of knowledge that was not needed when programs only considered one or two languages. The complexity of human languages and scripts has certainly made this a challenge when constructing a universal standard. Supporting Unicode properly in your program will require effort.

1.3.3 What Unicode Is

Unicode is a standard defining code points (numbers) for all known, living or dead, scripts. In principle, every 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, some **encoding standards** are 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 only a standard for representation. UTF-8 can, for example, be used to represent a very limited part of the Unicode character set (for example ISO-Latin-1) or the full Unicode range. It is only 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 was not even named. With the Unicode system there are much more than 256 characters, so a common way is needed 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 a non-issue earlier.

Different operating systems and tools support different encodings. For example, Linux and MacOS X have chosen the UTF-8 encoding, which is backward compatible with 7-bit ASCII and therefore affects programs written in plain English the least. Windows 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 following are the most widely spread encodings:

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 with numbers < 256, which exactly corresponds 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 backward 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.
>
> Notice that UTF-8 is **not** compatible with bytewise representation for code points from 128 through 255, so an ISO Latin-1 bytewise representation is generally incompatible 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, and some high numbers four bytes. Some programs, libraries, and operating systems claiming to use UTF-16 only allow 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, which is why UTF-16 exists in both a big-endian and a 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 straightforward representation. Each character is stored in one single 32-bit number. There is no need for escapes or any variable number 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-endian and little-endian.

UCS-4

> Basically the same as UTF-32, but without some Unicode semantics, defined by IEEE, and has little use as a separate encoding standard. For all normal (and possibly abnormal) use, UTF-32 and UCS-4 are interchangeable.

Certain number ranges are 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. This is possibly because the UTF-16 encoding standard, from the beginning, was expected to be able to hold all Unicode characters in one 16-bit entity, but was then extended, leaving a hole in the Unicode range to handle backward compatibility.

Code point 16#FEFF is used for Byte Order Marks (BOMs) and use of that character is not encouraged in other contexts. It is valid though, as the character "ZWNBS" (Zero Width Non Breaking Space). BOMs are used to identify encodings and byte order for programs where such parameters are not known in advance. BOMs are more seldom used than expected, but can become more widely spread as they provide the means for programs to make educated guesses about the Unicode format of a certain file.

1.3.4 Areas of Unicode Support

To support Unicode in Erlang, problems in various areas have been addressed. This section describes each area briefly and more thoroughly later in this User's Guide.

Representation

To handle Unicode characters in Erlang, a common representation in both lists and binaries is needed. 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, which is why library functions must be able to handle them. In some cases functionality has been added to already existing interfaces (as the *`string`* module now can handle strings with any code points). In some cases new functionality or options have been 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 must decide on an encoding when 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 Erlang I/O-system has been designed (or at least used) in a way where you expect any I/O server to handle any string data. That is, however, no longer the case when working with Unicode characters. The Erlang programmer must now know the capabilities of the device where the data ends up. Also, ports in Erlang are byte-oriented, so an arbitrary string of (Unicode) characters cannot 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 (for example, in the shell). There exists syntactic representation of any Unicode character without displaying the glyph (instead written as `\x{HHH}`). Unicode data can therefore usually be displayed even if the terminal as such does not support the whole Unicode range.

Filenames

Filenames can be stored as Unicode strings in different ways depending on the underlying operating system and file system. This can be handled fairly easy by a program. The problems arise when the file system is inconsistent in its encodings. For example, Linux allows files to be named with any sequence of bytes, leaving to each program to interpret those bytes. On systems where these "transparent" filenames are used, Erlang must be informed about the filename encoding by a startup flag. The default is bytewise interpretation, which is usually wrong, but allows for interpretation of **all** filenames.

The concept of "raw filenames" can be used to handle wrongly encoded filenames if one enables Unicode filename translation (`+fnu`) on platforms where this is not the default.

Source code encoding

The Erlang source code has support for the UTF-8 encoding and bytewise encoding. The default in Erlang/OTP R16B was bytewise (`latin1`) encoding. It was changed to UTF-8 in Erlang/OTP 17.0. You can control the encoding by a comment like the following in the beginning of the file:

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

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, and so on, 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, function names, and atoms containing Unicode characters with code points > 255. Module names, application names, and node names are still restricted to the ISO Latin-1 range. Binary literals, where you use type `/utf8`, can also be expressed using Unicode characters > 255. Having module names or application names using characters other than 7-bit ASCII can cause trouble on operating systems with inconsistent file naming schemes, and can hurt portability, so it is not recommended.

EEP 40 suggests that the language is also to allow for Unicode characters > 255 in variable names. Whether to implement that EEP is yet to be decided.

1.3.5 Standard Unicode Representation

In Erlang, strings are lists of integers. A string was until Erlang/OTP R13 defined to be encoded in the ISO Latin-1 (ISO 8859-1) character set, which is, code point by code point, a subrange of the Unicode character set.

The standard list encoding for strings was therefore easily extended to handle the whole Unicode range. A Unicode string in Erlang is a list containing integers, where each integer is a valid Unicode code point and represents 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, for example, *`erlang:iolist_to_binary/1`* or can be sent directly to a port. If the string contains Unicode characters > 255, an encoding must be decided upon and the string is to 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 can be converted into a binary, effectively using bytewise encoding: one byte per character. This was 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 is to handle 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. However, library functions dealing with binaries and Unicode in general only deal with the default encoding.

Character data can be combined from many 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* 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())
```

1.3.6 Basic Language Support

As from Erlang/OTP R16, Erlang source files can be written in UTF-8 or bytewise (`latin1`) encoding. For information about how to state the encoding of an Erlang source file, see the *epp(3)* module. As from Erlang/OTP R16, strings and comments can be written using Unicode. As from Erlang/OTP 20, also atoms and functions can be written using Unicode. Modules, applications, and nodes must still be named using characters from the ISO Latin-1 character set. (These restrictions in the language are independent of the encoding of the source file.)

Bit Syntax

The bit syntax contains types for handling binary data in the three main encodings. The types are named `utf8`, `utf16`, and `utf32`. The `utf16` and `utf32` types can be in a big-endian or a 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 syntax `\OOO` (backslash followed by three octal numbers) and `\xHH` (backslash followed by `x`, followed by two hexadecimal characters), namely `\x{H ...}` (backslash followed by `x`, followed by left curly bracket, any number of hexadecimal digits, and a terminating right curly bracket). This allows 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> $#.
1089
```

Heuristic String Detection

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

```
1> [97,98,99].
"abc"
2> <<97,98,99>>.
<<"abc">>
3> <<195,165,195,164,195,182>>.
<<"åäö"/utf8>>
```

Here the shell detects lists containing printable characters or binaries containing printable characters in bytewise or UTF-8 encoding. But what is a printable character? One view is that anything the Unicode standard thinks is printable, is also printable according to the heuristic detection. The result is then that almost any list of integers are deemed a string, and all sorts of characters are printed, maybe also characters that your terminal lacks in its font set (resulting in some unappreciated generic output). Another way is to keep it backward compatible so that only the ISO Latin-1 character set is used to detect a string. A third way is to let the user decide exactly what Unicode ranges that are to be viewed as characters.

As from Erlang/OTP R16B you can select the ISO Latin-1 range or the whole Unicode range by supplying startup flag `+pc latin1` or `+pc unicode`, respectively. For backward compatibility, `latin1` is default. This only controls how heuristic string detection is done. More ranges are expected to be added in the future, enabling tailoring of the heuristics to the language and region relevant to the user.

The following examples show the two 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> <<208,174,208,189,208,184,208,186,208,190,208,180>>.
<<208,174,208,189,208,184,208,186,208,190,208,180>>
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]

Eshell V5.10.1  (abort with ^G)
1> [1024].
"#"
2> [1070,1085,1080,1082,1086,1076].
"######"
3> [229,228,246].
"åäö"
4> <<208,174,208,189,208,184,208,186,208,190,208,180>>.
<<"######"/utf8>>
5> <<229/utf8,228/utf8,246/utf8>>.
<<"åäö"/utf8>>
```

In the examples, you can see that the default Erlang shell interprets only characters from the ISO Latin1 range as printable and only detects lists or binaries with those "printable" characters as containing string data. The valid UTF-8

binary containing the Russian word "######", is not printed as a string. When started with all Unicode characters printable (`+pc unicode`), the shell outputs anything containing printable Unicode data (in binaries, either UTF-8 or bytewise encoded) as string data.

These heuristics are also used by *`io:format/2`*, *`io_lib:format/2`*, and friends when modifier `t` is used with `~p` or `~P`:

```
$ 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> io:format("~tp~n",[{<<"ääö">>, <<"ääö"/utf8>>, <<208,174,208,189,208,184,208,186,208,190,208,180>>}]).
{<<"ääö">>,<<"ääö"/utf8>>,<<208,174,208,189,208,184,208,186,208,190,208,180>>}
ok
```

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

Eshell V5.10.1  (abort with ^G)
1> io:format("~tp~n",[{<<"ääö">>, <<"ääö"/utf8>>, <<208,174,208,189,208,184,208,186,208,190,208,180>>}]).
{<<"ääö">>,<<"ääö"/utf8>>,<<"######"/utf8>>}
ok
```

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

1.3.7 The Interactive Shell

The interactive Erlang shell, when started to a terminal or started using command `werl` 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**, which are freely available, and then select that font in the Erlang shell application.

On Unix-like operating systems, the terminal is to be able to handle UTF-8 on input and output (this is done by, for example, modern versions of XTerm, KDE Konsole, and the Gnome terminal) and your locale settings must be proper. As an example, a `LANG` environment variable can be set as follows:

```
$ echo $LANG
en_US.UTF-8
```

Most systems handle variable `LC_CTYPE` before `LANG`, so if that is set, it must be set to `UTF-8`:

```
$ echo $LC_CTYPE
en_US.UTF-8
```

The `LANG` or `LC_CTYPE` setting are to be consistent with what the terminal is capable of. There is no portable way for Erlang to ask the 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 call *`io:getopts()`* 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 have probably found 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 a KDE environment, select **KDE Control Center (Personal Settings)** > **Regional and Accessibility** > **Keyboard Layout**.

On Windows XP, select **Control Panel** > **Regional and Language Options**, select tab **Language**, and click button **Details...** in the square named **Text Services and Input Languages**.

Your environment probably provides similar means of changing the keyboard layout. Ensure that you have a way to switch back and forth between keyboards easily if you are not used to this. For example, entering commands using a Cyrillic character set is not easily done in the Erlang shell.

Now you are set up for some Unicode input and output. The simplest thing to do is 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> "######".
"######"
3> io:format("~ts~n", [v(2)]).
######
ok
4>
```

While strings can be input as Unicode characters, the language elements are still limited to the ISO Latin-1 character set. Only character constants and strings are allowed to be beyond that range:

```
$ erl
Erlang R16B (erts-5.10.1) [source] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.10.1  (abort with ^G)
1> $#.
958
2> ######.
* 1: illegal character
2>
```

1.3.8 Unicode Filenames

Most modern operating systems support Unicode filenames in some way. There are many 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 enforce Unicode support for filenames. All files created in the file system have names that can consistently be interpreted. In MacOS X, all filenames are retrieved in UTF-8 encoding. In Windows, each system call handling filenames has a special Unicode-aware variant, giving much the same effect. There are no filenames on these systems that are not Unicode filenames. So, the default behavior of the Erlang VM is to work in "Unicode filename translation mode". This means that a filename can be specified as a Unicode list, which is automatically translated to the proper name encoding for the underlying operating system and file system.
>
> Doing, for example, a *`file:list_dir/1`* on one of these systems can return Unicode lists with code points > 255, depending on the content of the file system.

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 filenames, but do not enforce it. On such a system, a filename containing characters with code points from 128 through 255 can be named as plain ISO Latin-1 or use UTF-8 encoding. As no consistency is enforced, the Erlang VM cannot do consistent translation of all filenames.
>
> By default on such systems, Erlang starts in `utf8` filename mode if the terminal supports UTF-8, otherwise in `latin1` mode.
>
> In `latin1` mode, filenames are bytewise encoded. This allows for list representation of all filenames in the system. However, a a file named "Östersund.txt", appears in *`file:list_dir/1`* either as "Östersund.txt" (if the filename was encoded in bytewise 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 want). If you use Unicode filename translation on such a system, non-UTF-8 filenames are ignored by functions like `file:list_dir/1`. They can be retrieved with function *`file:list_dir_all/1`*, but wrongly encoded filenames appear as "raw filenames".

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

On operating systems with mandatory Unicode filenames, this means that you more easily conform to the filenames of other (non-Erlang) applications. You can also process filenames that, at least on Windows, were inaccessible (because of having names that could not be represented in ISO Latin-1). Also, you avoid creating incomprehensible filenames on MacOS X, as the `vfs` layer of the operating system accepts all your filenames as UTF-8 does not rewrite them.

For most systems, turning on Unicode filename translation is no problem even if it uses transparent file naming. Very few systems have mixed filename encodings. A consistent UTF-8 named system works perfectly in Unicode filename mode. It was still, however, considered experimental in Erlang/OTP R14B01 and is still not the default on such systems.

Unicode filename translation is turned on with switch `+fnu`. On Linux, a VM started without explicitly stating the filename translation mode defaults to `latin1` as the native filename encoding. On Windows and MacOS X, the default behavior is that of Unicode filename translation. Therefore *`file:native_name_encoding/0`* by default returns `utf8` on those systems (Windows does not use UTF-8 on the file system level, but this can safely be ignored by the Erlang programmer). The default behavior can, as stated earlier, be changed using option `+fnu` or `+fnl` to the VM, see the *`erl`* program. If the VM is started in Unicode filename translation mode, `file:native_name_encoding/0`

returns atom `utf8`. Switch `+fnu` can be followed by `w`, `i`, or `e` to control how wrongly encoded filenames are to be reported.

- `w` means that a warning is sent to the `error_logger` whenever a wrongly encoded filename is "skipped" in directory listings. `w` is the default.
- `i` means that wrongly encoded filenames are silently ignored.
- `e` means that the API function returns an error whenever a wrongly encoded filename (or directory name) is encountered.

Notice that *`file:read_link/1`* always returns an error if the link points to an invalid filename.

In Unicode filename mode, filenames given to BIF `open_port/2` with option `{spawn_executable,...}` are also interpreted as Unicode. So is the parameter list specified in option `args` available when using `spawn_executable`. The UTF-8 translation of arguments can be avoided using binaries, see section *Notes About Raw Filenames*.

Notice that the file encoding options specified when opening a file has nothing to do with the filename encoding convention. You can very well open files containing data encoded in UTF-8, but having filenames in bytewise (`latin1`) encoding or conversely.

Note:

Erlang drivers and NIF-shared objects still cannot be named with names containing code points > 127. This limitation will be removed in a future release. However, Erlang modules can, but it is definitely not a good idea and is still considered experimental.

Notes About Raw Filenames

Raw filenames were introduced together with Unicode filename support in ERTS 5.8.2 (Erlang/OTP R14B01). The reason "raw filenames" were introduced in the system was to be able to represent filenames, specified in different encodings on the same system, consistently. It can seem practical to have the VM automatically translate a filename that is not in UTF-8 to a list of Unicode characters, but this would open up for both duplicate filenames and other inconsistent behavior.

Consider a directory containing a file named "björn" in ISO Latin-1, while the Erlang VM is operating in Unicode filename mode (and therefore expects UTF-8 file naming). The ISO Latin-1 name is not valid UTF-8 and one can 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 filename)? The VM converts the filename 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 UTF-8 encoding and one not. If `file:list_dir/1` would automatically convert the ISO Latin-1 filename to a list, we would get two identical filenames as the result. To avoid this, we must differentiate between filenames that are properly encoded according to the Unicode file naming convention (that is, UTF-8) and filenames that are invalid under the encoding. By the common function `file:list_dir/1`, the wrongly encoded filenames are ignored in Unicode filename translation mode, but by function *`file:list_dir_all/1`* the filenames with invalid encoding are returned as "raw" filenames, that is, as binaries.

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

To force Unicode filename translation mode on systems where this is not the default was considered experimental in Erlang/OTP R14B01. This was because the initial implementation did not ignore wrongly encoded filenames, so that

raw filenames could spread unexpectedly throughout the system. As from Erlang/OTP R16B, the wrongly encoded filenames are only retrieved by special functions (such as `file:list_dir_all/1`). Since the impact on existing code is therefore much lower it is now supported. Unicode filename 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 filenames encoded as UTF-8. Enforcing the UTF-8 encoding regardless of the mode the Erlang VM is started in can in some circumstances be a good idea, as the convention of using UTF-8 filenames is spreading.

Notes About MacOS X

The `vfs` layer of MacOS X enforces UTF-8 filenames in an aggressive way. Older versions did this by refusing to create non-UTF-8 conforming filenames, 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 flag `+fnl` or to use a raw filename in bytewise (`latin1`) encoding. If using a raw filename, with a bytewise encoding containing characters from 127 through 255, to create a file, the file cannot be opened using the same name as the one used to create it. There is no remedy for this behavior, except keeping the filenames in the correct encoding.

MacOS X reorganizes the filenames so that the representation of accents, and so on, uses the "combining characters". For example, character ö is represented as code points `[111,776]`, where `111` is character `o` and `776` is the special accent character "Combining Diaeresis". This way of normalizing Unicode is otherwise very seldom used. Erlang normalizes those filenames in the opposite way upon retrieval, so that filenames using combining accents are not passed up to the Erlang application. In Erlang, filename "björn" is retrieved as `[98,106,246,114,110]`, not as `[98,106,117,776,114,110]`, although the file system can think differently. The normalization into combining accents is redone when accessing files, so this can usually be ignored by the Erlang programmer.

1.3.9 Unicode in Environment and Parameters

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

If Unicode filenames are enabled, the calls to *`os:getenv/0,1`*, *`os:putenv/2`*, and *`os:unsetenv/1`* handle Unicode strings. On Unix-like platforms, the built-in functions 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 are used, and code points > 255 are allowed.

On Unix-like operating systems, parameters are expected to be UTF-8 without translation if Unicode filenames are enabled.

1.3.10 Unicode-Aware Modules

Most of the modules in Erlang/OTP are Unicode-unaware in the sense that they have no notion of Unicode and should not have. Typically they handle non-textual or byte-oriented data (such as `gen_tcp`).

Modules handling textual data (such as *`io_lib`* and *`string`* 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, so modules like `string` work well for Unicode strings with little need for conversion or extension.

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

`unicode`

> The *`unicode`* module is clearly Unicode-aware. It contains functions for conversion between different Unicode formats and some utilities for identifying byte order marks. Few programs handling Unicode data 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 many functions require binaries to be in UTF-8, and there are modifiers to format control sequences to allow for output of Unicode strings.

`file`, `group`, `user`

I/O-servers throughout the system can handle Unicode data and have options for converting data upon output or input to/from the device. As shown earlier, the *`shell`* module has support for Unicode terminals and the *`file`* module allows for translation to and from various Unicode formats on disk.

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 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 centered on matching in binaries, the Unicode support is UTF-8-centered.

`wx`

The graphical library *`wx`* has extensive support for Unicode text.

The *`string`* module works perfectly for Unicode strings and ISO Latin-1 strings, except the language-dependent functions *`string:uppercase/1`* and *`string:lowercase/1`*. These two functions can never function correctly for Unicode characters in their current form, as there are language and locale issues to consider when converting text between cases. Converting case in an international environment is a large subject not yet addressed in OTP.

1.3.11 Unicode Data in Files

Although Erlang 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 and I/O servers, are not generally Unicode capable.

Ports are always byte-oriented, so before sending data that you are not sure is bytewise-encoded to a port, ensure to encode it in a proper Unicode encoding. Sometimes this means that only part of the data must be encoded as, for example, UTF-8. Some parts can be binary data (like a length indicator) or something else that must 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.

A file can have an encoding option that makes it generally usable by the *`io`* module (for example `{encoding,utf8}`), but is by default opened as a byte-oriented file. The *`file`* module is byte-oriented, so only ISO Latin-1 characters can be written using that module. Use the `io` module if Unicode data is to be output to a file with other `encoding` than `latin1` (bytewise encoding). It is slightly confusing that a file opened with, for example, `file:open(Name,[read,{encoding,utf8}])` cannot be properly read using `file:read(File,N)`, but using the `io` module to retrieve the Unicode data from it. The reason is that `file:read` and `file:write` (and friends) are purely byte-oriented, and should be, as that is the way to access files other than text files, byte by byte. As with ports, you can write encoded data into a file by "manually" converting the data to the encoding of choice (using the *`unicode`* module or the bit syntax) and then output it on a bytewise (`latin1`) encoded file.

Recommendations:

- Use the *`file`* module for files opened for bytewise access (`{encoding,latin1}`).
- Use the *`io`* module when accessing files with any other encoding (for example `{encoding,uf8}`).

Functions reading Erlang syntax from files recognize the `coding:` comment and can therefore handle Unicode data on input. When writing Erlang terms to a file, you are advised to insert such comments when applicable:

```
$ 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[{\"######\",4711}].\n"/utf8>>).
ok
2> file:consult("test.term").
{ok,[[{"######",4711}]]}
```

1.3.12 Summary of Options

The Unicode support is controlled by both command-line switches, some standard environment variables, and the OTP version you are using. Most options affect mainly how Unicode data is displayed, not the 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 `LANG` and `LC_CTYPE` environment variables

The language setting in the operating system mainly affects the shell. The terminal (that is, the group leader) operates with `{encoding, unicode}` only if the environment tells it that UTF-8 is allowed. This setting is to correspond to the terminal you are using.

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

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

The `+pc` {`unicode`|`latin1`} flag to *`erl(1)`*

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

You can check this option by calling *`io:printable_range/0`*, which returns `unicode` or `latin1`. To be compatible with future (expected) extensions to the settings, rather use *`io_lib:printable_list/1`* to check if a list is printable according to the setting. That function takes into account new possible settings returned from `io:printable_range/0`.

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

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

- `+fnl` means bytewise interpretation of filenames, which was the usual way to represent ISO Latin-1 filenames before UTF-8 file naming got widespread.
- `+fnu` means that filenames 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 environment variables `LANG` and `LC_CTYPE`. This is optimistic heuristics indeed, nothing enforces a user to have a terminal with the same encoding as the file system, but this is usually the case. This is the default on all Unix-like operating systems, except MacOS X.

The filename translation mode can be read with function *`file:native_name_encoding/0`*, which returns `latin1` (bytewise 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` (bytewise encoding) was returned. As from Erlang/OTP 17.0, `utf8` is returned.

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

io:setopts/1,2 and flags `-oldshell`/`-noshell`

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

You can set the encoding of a file or other I/O server with function *io:setopts/2*. This can also be set when opening a file. Setting the terminal (or other `standard_io` server) unconditionally to option `{encoding,utf8}` implies that UTF-8 encoded characters are written to the device, regardless of how Erlang was started or the user's environment.

Opening files with option `encoding` is convenient when writing or reading text files in a known encoding.

You can retrieve the `encoding` setting for an I/O server with function *io:getopts()*.

1.3.13 Recipes

When starting with Unicode, one often stumbles over some common issues. This section describes some methods of dealing with Unicode data.

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 remaining file. If such a file is to be read, the first few bytes (depending on encoding) are not part of the text. This code outlines how to open a file that is believed to have a BOM, and sets the files encoding and position for further sequential reading (preferably using the *io* module).

Notice that error handling is omitted from the code:

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

Function *unicode:bom_to_encoding/1* identifies the encoding from a binary of at least four bytes. It returns, along with a term suitable for setting the encoding of the file, the byte length of the BOM, so that the file position can be set accordingly. Notice that function *file:position/2* always works on byte-offsets, so that the byte length of the BOM is needed.

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

```
open_bom_file_for_writing(File,Encoding) ->
    {ok,F} = file:open(File,[write,binary]),
    ok = file:write(File,unicode:encoding_to_bom(Encoding)),
    io:setopts(F,[{encoding,Encoding}]),
    {ok,F}.
```

The file is in both these cases then best processed using the *io* module, as the functions in that module can handle code points beyond the ISO Latin-1 range.

Formatted I/O

When reading and writing to Unicode-aware entities, like a file opened for Unicode translation, you probably want to format text strings using the functions in the *io* module or the *io_lib* module. For backward compatibility reasons, these functions do not accept any list as a string, but require a special **translation modifier** when working with Unicode texts. The modifier is `t`. When applied to control character `s` in a formatting string, it accepts all Unicode code points and expects binaries to be in UTF-8:

```
1> io:format("~ts~n",[<<"åäö"/utf8>>]).
åäö
ok
2> io:format("~s~n",[<<"åäö"/utf8>>]).
Ã¥Ã¤Ã¶
ok
```

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

As long as the data is always lists, modifier `t` can be used for any string, but when binary data is involved, care must be taken to make the correct choice of formatting characters. A bytewise-encoded binary is also interpreted as a string, and printed even when using `~ts`, but it can be mistaken for a valid UTF-8 string. Avoid therefore using the `~ts` control if the binary contains bytewise-encoded characters and not UTF-8.

Function *io_lib:format/2* behaves similarly. It is defined to return a deep list of characters and the output can easily be converted to binary data for outputting on any device 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` then fails. However, if the I/O server you want to communicate with is Unicode-aware, the returned list can still be used directly:

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

Eshell V5.10.1 (abort with ^G)
1> io_lib:format("~ts~n", ["##########"]).
["##########","\n"]
2> io:put_chars(io_lib:format("~ts~n", ["##########"])).
##########
ok
```

The Unicode string is returned as a Unicode list, which is recognized as such, as the Erlang shell uses the Unicode encoding (and is started with all Unicode characters considered printable). The Unicode list is valid input to function *io:put_chars/2*, so data can be output on any Unicode-capable device. If the device is a terminal, characters are output in format `\x{H...}` 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).

So, you can always send Unicode data to the `standard_io` device. Files, however, accept only 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 encoding of characters in binary data is known before 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 BOMs in the files to identify them.

UTF-8 is designed so 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). The *unicode* module can be used to determine if data can be interpreted as UTF-8:

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

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

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

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

Another option is to try to read the whole file in UTF-8 encoding and see if it fails. Here we need to read the file using function *io:get_chars/3*, as we have to read characters with a code point > 255:

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

loop_through_file2(_,eof) ->
    utf8;
loop_through_file2(_,{error,_Err}) ->
    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 can sometimes have a list of UTF-8 bytes. This is not a regular string of Unicode characters, as each list element 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:

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

Double UTF-8 Encoding

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

By far the most common situation where this occurs, 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:

```
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
                          %% bytes in a list!
  file:close(C). %% The file catastrophe.txt contains more or less unreadable
                 %% garbage!
```

Ensure 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

STDLIB

Application

The STDLIB application 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. For more information about configuration parameters, see the *app(4)* module in Kernel.

`shell_esc = icl | abort`

Can be used to change the behavior of the Erlang shell when **^G** is pressed.

`restricted_shell = module()`

Can be used to run the Erlang shell in restricted mode.

`shell_catch_exception = boolean()`

Can be used to set the exception handling of the evaluator process of Erlang shell.

`shell_history_length = integer() >= 0`

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()`

Can be used to set a customized Erlang shell prompt function.

`shell_saved_results = integer() >= 0`

Can be used to determine how many results are saved by the Erlang shell.

`shell_strings = boolean()`

Can be used to determine how the Erlang shell outputs lists of integers.

See Also

app(4), *application(3)*, *shell(3)*

array

Erlang module

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

Arrays uses **zero**-based indexing. This is a deliberate design choice and differs from other Erlang data structures, for example, 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 that has been explicitly set to the same value as the default one (compare *`reset/2`*). If you need to differentiate between unset and set entries, ensure 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 to set an entry successfully, all indices in the range [0,`I`] stay accessible unless the array size is explicitly changed by calling *`resize/2`*.

Examples:

Create a fixed-size array with entries 0-9 set to `undefined`:

```
A0 = array:new(10).
10 = array:size(A0).
```

Create an extendible array and set entry 17 to `true`, causing the array to grow automatically:

```
A1 = array:set(17, true, array:new()).
18 = array:size(A1).
```

Read back a stored value:

```
true = array:get(17, A1).
```

Accessing an unset entry returns 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. Notice that arrays cannot be directly compared for equality.

```
array() = array(term())
array_indx() = 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_indx(), Type}
```

Exports

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

Gets the value used for uninitialized entries.

See also *new/2*.

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

Fixes the array size. This prevents it from growing automatically upon insertion.

See also *set/3* and *relax/1*.

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

Types:

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

Folds the array elements using the specified 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_indx(), Value :: Type, Acc :: A) -> B)
```

Folds the array elements right-to-left using the specified 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)**

Converts 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)**

Converts 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 non-negative integers, the call fails with reason `badarg`.

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

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

Gets the value of entry `I`. If `I` is not a non-negative 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, the default value for any index `I` greater than `size(Array)-1` is returned.

See also *set/3*.

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

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

is_fix(Array :: *array()*) -> boolean()

Checks 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)**

Maps the specified function onto each array element. 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*.

new() -> *array()*

Creates a new, extendible array with initial size zero.

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

new(Options :: *array_opts()*) -> *array()*

Creates a new array according to the specified otions. 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 array size; this also implies `{fixed, true}`. If `N` is not a non-negative integer, the call fails with reason `badarg`.

`fixed` or `{fixed, true}`

Creates a fixed-size array. See also *fix/1*.

`{fixed, false}`

Creates an extendible (non-fixed-size) array.

`{default, Value}`

Sets the default value for the array to `Value`.

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

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

Examples:

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

creates a fixed-size array of size 100.

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

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

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

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

See also *fix/1*, *from_list/2*, *get/2*, *new/0*, *new/2*, *set/3*.

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

Creates a new array according to the specified size and options. If `Size` is not a non-negative integer, the call fails with reason `badarg`. By default, the array has fixed size. Notice that any size specifications in `Options` override parameter `Size`.

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

Example:

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

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

See also *new/1*.

relax(Array :: *array*(Type)) -> *array*(Type)

Makes the array resizable. (Reverses the effects of *fix/1*.)

See also *fix/1*.

reset(I :: *array_indx()*, Array :: *array*(Type)) -> *array*(Type)

Resets entry `I` to the default value for the array. If the value of entry `I` is the default value, the array is returned unchanged. Reset never changes the array size. Shrinking can be done explicitly by calling *resize/2*.

If `I` is not a non-negative integer, or if the array has fixed size and `I` is larger than the maximum index, the call fails with reason `badarg`; compare *set/3*

See also *new/2*, *set/3*.

resize(Array :: *array*(Type)) -> *array*(Type)

Changes the array size to that reported by *sparse_size/1*. If the specified array has fixed size, also the resulting array has fixed size.

See also *resize/2*, *sparse_size/1*.

resize(Size :: integer() >= 0, Array :: *array*(Type)) ->
** *array*(Type)**

Change the array size. If `Size` is not a non-negative integer, the call fails with reason `badarg`. If the specified array has fixed size, also the resulting array has fixed size.

set(I :: *array_indx()*, Value :: Type, Array :: *array*(Type)) ->
** *array*(Type)**

Sets entry `I` of the array to `Value`. If `I` is not a non-negative 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 grows to size `I+1`.

See also *get/2*, *reset/2*.

size(Array :: *array()*) -> integer() >= 0

Gets 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_indx()*, Value :: Type, Acc :: A) -> B)

Folds the array elements using the specified 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)

Folds the array elements right-to-left using the specified 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)

Maps the specified function onto each array element, 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

Gets the number of entries in the array up until the last non-default-valued entry. That is, 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)**

Converts 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)

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

See also *from_orddict/2*, *sparse_to_orddict/1*.

assert.hrl

Name

The include file `assert.hrl` provides macros for inserting assertions in your program code.

Include the following directive in the module from which the function is called:

```
-include_lib("stdlib/include/assert.hrl").
```

When an assertion succeeds, the assert macro yields the atom `ok`. When an assertion fails, an exception of type `error` is generated. The associated error term has the form `{Macro, Info}`. `Macro` is the macro name, for example, `assertEqual`. `Info` is a list of tagged values, such as `[{module, M}, {line, L}, ...]`, which gives more information about the location and cause of the exception. All entries in the `Info` list are optional; do not rely programatically on any of them being present.

Each assert macro has a corresponding version with an extra argument, for adding comments to assertions. These can for example be printed as part of error reports, to clarify the meaning of the check that failed. For example, `?assertEqual(0, fib(0), "Fibonacci is defined for zero")`. The comment text can be any character data (string, UTF8-binary, or deep list of such data), and will be included in the error term as `{comment, Text}`.

If the macro `NOASSERT` is defined when `assert.hrl` is read by the compiler, the macros are defined as equivalent to the atom `ok`. The test will not be performed and there is no cost at runtime.

For example, using `erlc` to compile your modules, the following disables all assertions:

```
erlc -DNOASSERT=true *.erl
```

(The value of `NOASSERT` does not matter, only the fact that it is defined.)

A few other macros also have effect on the enabling or disabling of assertions:

- If `NODEBUG` is defined, it implies `NOASSERT` (unless `DEBUG` is also defined, which overrides `NODEBUG`).
- If `ASSERT` is defined, it overrides `NOASSERT`, that is, the assertions remain enabled.

If you prefer, you can thus use only `DEBUG`/`NODEBUG` as the main flags to control the behavior of the assertions (which is useful if you have other compiler conditionals or debugging macros controlled by those flags), or you can use `ASSERT`/`NOASSERT` to control only the assert macros.

Macros

`assert(BoolExpr)`
`assert(BoolExpr, Comment)`

Tests that `BoolExpr` completes normally returning `true`.

`assertNot(BoolExpr)`
`assertNot(BoolExpr, Comment)`

Tests that `BoolExpr` completes normally returning `false`.

`assertMatch(GuardedPattern, Expr)`
`assertMatch(GuardedPattern, Expr, Comment)`

Tests that `Expr` completes normally yielding a value that matches `GuardedPattern`, for example:

```
?assertMatch({bork, _}, f())
```

Notice that a guard `when ...` can be included:

```
?assertMatch({bork, X} when X > 0, f())
```

`assertNotMatch(GuardedPattern, Expr)`
`assertNotMatch(GuardedPattern, Expr, Comment)`

Tests that `Expr` completes normally yielding a value that does not match `GuardedPattern`.

As in `assertMatch`, `GuardedPattern` can have a `when` part.

`assertEqual(ExpectedValue, Expr)`
`assertEqual(ExpectedValue, Expr, Comment)`

Tests that `Expr` completes normally yielding a value that is exactly equal to `ExpectedValue`.

`assertNotEqual(ExpectedValue, Expr)`
`assertNotEqual(ExpectedValue, Expr, Comment)`

Tests that `Expr` completes normally yielding a value that is not exactly equal to `ExpectedValue`.

`assertException(Class, Term, Expr)`
`assertException(Class, Term, Expr, Comment)`

Tests that `Expr` completes abnormally with an exception of type `Class` and with the associated `Term`. The assertion fails if `Expr` raises a different exception or if it completes normally returning any value.

Notice that both `Class` and `Term` can be guarded patterns, as in `assertMatch`.

`assertNotException(Class, Term, Expr)`
`assertNotException(Class, Term, Expr, Comment)`

Tests that `Expr` does not evaluate abnormally with an exception of type `Class` and with the associated `Term`. The assertion succeeds if `Expr` raises a different exception or if it completes normally returning any value.

As in `assertException`, both `Class` and `Term` can be guarded patterns.

`assertError(Term, Expr)`
`assertError(Term, Expr, Comment)`

Equivalent to `assertException(error, Term, Expr)`

`assertExit(Term, Expr)`
`assertExit(Term, Expr, Comment)`

Equivalent to `assertException(exit, Term, Expr)`

`assertThrow(Term, Expr)`
`assertThrow(Term, Expr, Comment)`

Equivalent to `assertException(throw, Term, Expr)`

See Also

`compile(3)`, *`erlc(3)`*

base64

Erlang module

Provides base64 encode and decode, see **RFC 2045**.

Data Types

ascii_string() = [1..255]
ascii_binary() = binary()

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

Exports

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

Types:

Base64 = *ascii_string()* | *ascii_binary()*
Data = *ascii_binary()*
DataString = *ascii_string()*

Decodes a base64-encoded string to plain ASCII. See **RFC 4648**.

`mime_decode/1` and `mime_decode_to_string/1` strip away illegal characters, while `decode/1` and `decode_to_string/1` only strip away whitespace characters.

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

Types:

Data = *ascii_string()* | *ascii_binary()*
Base64 = *ascii_binary()*
Base64String = *ascii_string()*

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

beam_lib

Erlang module

This module 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 recognized names and the corresponding identifiers are as follows:

- `atoms ("Atom")`
- `attributes ("Attr")`
- `compile_info ("CInf")`
- `debug_info ("Dbgi")`
- `exports ("ExpT")`
- `imports ("ImpT")`
- `indexed_imports ("ImpT")`
- `labeled_exports ("ExpT")`
- `labeled_locals ("LocT")`
- `locals ("LocT")`

Debug Information/Abstract Code

Option `debug_info` can be specified to the Compiler (see *`compile(3)`*) to have debug information, such as *Erlang Abstract Format*, stored in the `debug_info` chunk. Tools such as Debugger and Xref require the debug information to be included.

Warning:

Source code can be reconstructed from the debug information. To prevent this, use encrypted debug information (see below).

The debug information can also be removed from BEAM files using *`strip/1`*, *`strip_files/1`*, and/or *`strip_release/1`*.

Reconstruct Source Code

The following example shows how to reconstruct Erlang source code from the debug information in a BEAM file `Beam`:

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

Encrypted Debug Information

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

To use encrypted debug information, a key must be provided to the compiler and `beam_lib`. The key is specified as a string. It is recommended that the string 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 is scrambled using *`erlang:md5/1`* to generate the 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.

The key can be provided in the following two ways:

- Use Compiler option `{debug_info_key,Key}`, see *`compile(3)`* and function *`crypto_key_fun/1`* to register a fun that returns the key whenever `beam_lib` must decrypt the debug information.

  If no such fun is registered, `beam_lib` instead searches for an `.erlang.crypt` file, see the next section.
- Store the key in a text file named `.erlang.crypt`.

  In this case, 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` implicitly creates a crypto key fun and registers it.

File `.erlang.crypt` is to contain a single list of tuples:

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

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

`Key` in the first tuple where both `Mode` and `Module` match is used.

The following 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^"}].
```

The following is a slightly more complicated example of an `.erlang.crypt` providing one key for module `t` and another key for all other modules:

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

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

Data Types

`beam() = module() | 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()} |
```

```
    {debug_info, debug_info()} |
    {attributes, [attrib_entry()]} |
    {compile_info, [compinfo_entry()]} |
    {exports, [{atom(), arity()}]} |
    {labeled_exports, [labeled_entry()]} |
    {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()
```

"Attr" | "CInf" | "Dbgi" | "ExpT" | "ImpT" | "LocT" | "AtU8"

```
dataB() = binary()
debug_info() =
    {DbgiVersion :: atom(), Backend :: module(), Data :: term()} |
    no_debug_info
```

The format stored in the `debug_info` chunk. To retrieve particular code representation from the backend, `Backend:debug_info(Format, Module, Data, Opts)` must be invoked. `Format` is an atom, such as `erlang_v1` for the Erlang Abstract Format or `core_v1` for Core Erlang. `Module` is the module represented by the beam file and `Data` is the value stored in the debug info chunk. `Opts` is any list of values supported by the `Backend`. `Backend:debug_info/4` must return `{ok, Code}` or `{error, Term}`.

Developers must always invoke the `debug_info/4` function and never rely on the `Data` stored in the `debug_info` chunk, as it is opaque and may change at any moment. `no_debug_info` means that chunk `"Dbgi"` is present, but empty.

```
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 chunk `"Abst"` is present, but empty.

For modules compiled with OTP 20 onwards, the `abst_code` chunk is automatically computed from the `debug_info` chunk.

```
forms() = [erl_parse:abstract_form() | erl_parse:form_info()]
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 |
    debug_info |
    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 | debug_info} |
    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

```
all_chunks(File :: beam()) ->
              {ok, beam_lib, [{chunkid(), dataB()}]}
```

Reads chunk data for all chunks.

```
build_module(Chunks) -> {ok, Binary}
```

Types:

```
Chunks = [{chunkid(), dataB()}]
Binary = binary()
```

Builds a BEAM module (as a binary) from a list of chunks.

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

Types:

```
Beam = beam()
ChunkRefs = [chunkref()]
```

Reads chunk data for selected chunks references. 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 references. 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 option `allow_missing_chunks` is specified, a result is returned even if chunks are missing. In the result list, any missing chunks are represented as `{ChunkRef,missing_chunk}`. Notice however that if chunk `"Atom"` is missing, that is considered a fatal error and the return value is an `error` tuple.

```
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`*.

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

```
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 chunk `"CInf"` (the chunk containing the compilation information that 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()
```

Compares the BEAM files in two directories. Only files with extension `".beam"` are compared. BEAM files that exist only in directory `Dir1` (`Dir2`) are returned in `Only1` (`Only2`). BEAM files that exist in both directories

but are considered different by `cmp/2` are returned as pairs {`Filename1`, `Filename2`}, where `Filename1` (`Filename2`) exists in directory `Dir1` (`Dir2`).

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

Types:

```
CryptoKeyFun = crypto_fun()
Reason = badfun | exists | term()
crypto_fun() = fun((crypto_fun_arg()) -> term())
crypto_fun_arg() =
    init | clear | {debug_info, mode(), module(), file:filename()}
mode() = des3_cbc
```

Registers an unary fun that is called if `beam_lib` must read an `debug_info` chunk that has been encrypted. The fun is held in a process that is started by the function.

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

The fun must handle the following arguments:

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

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

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

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

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

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

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

Types:

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

Compares the BEAM files in two directories as *`cmp_dirs/2`*, but the names of files that exist in only one directory or are different are presented on standard output.

`format_error(Reason) -> io_lib:chars()`

Types:

```
Reason = term()
```

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

`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.

```
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).

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

Types:

```
Beam1 = Beam2 = beam()
```

Removes all chunks from a BEAM file except those needed by the loader. In particular, the debug information (chunk `debug_info` and `abstract_code`) is removed.

```
strip_files(Files) ->
              {ok, [{module(), Beam}]} |
              {error, beam_lib, info_rsn()}
```

Types:

```
Files = [beam()]
Beam = beam()
```

Removes all chunks except those needed by the loader from BEAM files. In particular, the debug information (chunk `debug_info` and `abstract_code`) is removed. The returned list contains one element for each specified filename, in the same order as in `Files`.

```
strip_release(Dir) ->
                {ok, [{module(), file:filename()}]} |
```

{error, beam_lib, Reason}

Types:

Dir = atom() | *file:filename()*

Reason = {not_a_directory, term()} | *info_rsn()*

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

version(Beam) ->
{ok, {module(), [Version :: term()]}} |
{error, beam_lib, *chnk_rsn()*}

Types:

Beam = *beam()*

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

Examples:

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

binary

Erlang module

This module contains functions for manipulating byte-oriented binaries. Although the majority of functions could be provided 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 provided according to Erlang Enhancement Proposal (EEP) 31.

> **Note:**
>
> The library handles byte-oriented data. For bitstrings that are not binaries (does not contain whole octets of bits) a `badarg` exception is 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 representaion 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 binary `Subject` as an integer. If `Pos` >= `byte_size(Subject)`, a `badarg` exception is raised.

`bin_to_list(Subject) -> [byte()]`

Types:

```
Subject = binary()
```

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()`s, each representing the value of one byte. `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()

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 functions *`match/3`*, *`matches/3`*, *`split/3`*, or *`replace/4`*. The `cp()` returned is guaranteed to be a `tuple()` to allow programs to distinguish it from non-precompiled search patterns.

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

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

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

copy(Subject) -> binary()

Types:

Subject = binary()

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 always creates a new binary, even if `N = 1`. By using *`copy/1`* on a binary referencing a larger binary, one can free up the larger binary for garbage collection.

> **Note:**
>
> By deliberately copying a single binary to avoid referencing a larger binary, one can, 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 can be a good idea.

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

`decode_unsigned(Subject) -> Unsigned`

Types:

```
Subject = binary()
Unsigned = integer() >= 0
```

Same as `decode_unsigned(Subject, big)`.

`decode_unsigned(Subject, Endianness) -> Unsigned`

Types:

```
Subject = binary()
Endianness = big | little
Unsigned = integer() >= 0
```

Converts the binary digit representation, in big endian 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
```

Same as `encode_unsigned(Unsigned, big)`.

`encode_unsigned(Unsigned, Endianness) -> binary()`

Types:

```
Unsigned = integer() >= 0
Endianness = big | little
```

Converts a positive integer to the smallest possible representation in a binary digit representation, either big endian or little endian.

Example:

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

`first(Subject) -> byte()`

Types:

Subject = binary()

Returns the first byte of 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 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 list `Binaries`.

Example:

```
1> binary:longest_common_prefix([<<"erlang">>, <<"ergonomy">>]).
2
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 list `Binaries`.

Example:

```
1> binary:longest_common_suffix([<<"erlang">>, <<"fang">>]).
3
2> binary:longest_common_suffix([<<"erlang">>, <<"perl">>]).
0
```

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

match(Subject, Pattern) -> Found | nomatch

Types:

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

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 returns `{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 specified part is searched. Return values still have offsets from the beginning of `Subject`. A negative `Length` is allowed as described in section Data Types in this manual.

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

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

If `{scope, {Start,Length}}` is specified in the options such that `Start` > size of `Subject`, `Start` + `Length` < 0 or `Start` + `Length` > size of `Subject`, a `badarg` exception is raised.

matches(Subject, Pattern) -> Found

Types:

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

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()}
```

As *`match/2`*, but `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:

```
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 a pattern is found, an empty list is returned.

For a description of `Pattern`, see *`compile_pattern/1`*. For a description of available options, see *`match/3`*.

If `{scope, {Start,Length}}` is specified in the options such that `Start` > size of `Subject`, `Start + Length` < 0 or `Start + Length` is > size of `Subject`, a `badarg` exception is raised.

part(Subject, PosLen) -> binary()

Types:

Subject = binary()

PosLen = *part()*

Extracts the part of binary `Subject` described by `PosLen`.

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

```
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()

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 subbinary), it can be useful to get the size of the referenced binary. This function can be used in a program to trigger the use of *`copy/1`*. By copying a binary, one can dereference the original, possibly large, binary that a smaller binary is a reference to.

Example:

```
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 `gb_sets:set()` if it references a binary more than twice the data size we want to keep. Of course, different rules apply when copying to different programs.

Binary sharing occurs 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` can 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 does 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()`

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 subpart of `Subject` giving raise to the replacement is to be inserted in the result, option `{insert_replaced, InsPos}` inserts the matching part into `Replacement` at the specified position (or positions) before inserting `Replacement` into `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[b-b]c[d-d]e">>
```

If any position specified in `InsPos` > size of the replacement binary, a `badarg` exception is raised.

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()]

Same as `split(Subject, Pattern, [])`.

split(Subject, Pattern, Options) -> Parts

Types:

Subject = binary()
Pattern = binary() | [binary()] | *cp()*
Options = [Option]
Option = {scope, *part()*} | trim | global | trim_all
Parts = [binary()]

Splits `Subject` into a list of binaries based on `Pattern`. If option `global` is not specified, only the first occurrence of `Pattern` in `Subject` gives rise to a split.

The parts of `Pattern` 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`*. Notice 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 are kept in the result. See the example below.

trim

Removes trailing empty parts of the result (as does `trim` in *`re:split/3`*.

trim_all

Removes all empty parts of the result.

global

Repeats the split until `Subject` is exhausted. Conceptually option `global` 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 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`*.

c

Erlang module

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

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

Exports

`bt(Pid) -> ok | undefined`

Types:

`Pid = pid()`

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

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

Types:

`Module = atom()`
`Options = [compile:option()]`
`Filter = fun((compile:option()) -> boolean())`
`ModuleName = module()`

Compiles and then purges and loads the code for a module. `Module` can be either a module name or a source file path, with or without `.erl` extension. `Options` defaults to `[]`.

If `Module` is an atom and is not the path of a source file, then the code path is searched to locate the object file for the module and extract its original compiler options and source path. If the source file is not found in the original location, *`filelib:find_source/1`* is used to search for it relative to the directory of the object file.

The source file is compiled with the the original options appended to the given `Options`, the output replacing the old object file if and only if compilation succeeds. A function `Filter` can be specified for removing elements from from the original compiler options before the new options are added.

Notice that purging the code means that any processes lingering in old code for the module are killed without warning. For more information, see `code/3`.

`cd(Dir) -> ok`

Types:

`Dir = file:name()`

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

Example:

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

flush() -> ok

Flushes any messages sent to the shell.

help() -> ok

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

i() -> ok
ni() -> ok

`i/0` displays system information, 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)`.

Notice that purging the code means that any processes lingering in old code for the module are killed without warning. For more information, see `code/3`.

lc(Files) -> ok

Types:

Files = [File]
File

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

For information about `File`, see *`file:filename()`*.

lm() -> [*code:load_ret()*]

Reloads all currently loaded modules that have changed on disk (see `mm()`). Returns the list of results from calling `l(M)` for each such `M`.

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 lists 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`.

mm() -> [module()]

Lists all modified modules. Shorthand for *`code:modified_modules/0`*.

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 = *file:name()*
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(File, Options ++ [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 pid `<X.Y.Z>`. This function is only to 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.

uptime() -> ok

Prints the node uptime (as specified by `erlang:statistics(wall_clock)`) in human-readable form.

xm(ModSpec) -> void()

Types:

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

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

y(File) -> YeccRet

Types:

File = name()
YeccRet

Generates an LALR-1 parser. Equivalent to:

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

For information about `File = name()`, see *filename(3)*. For information about `YeccRet`, see *yecc:file/2*.

y(File, Options) -> YeccRet

Types:

File = name()
Options, YeccRet

Generates an LALR-1 parser. Equivalent to:

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

For information about `File = name()`, see *`filename(3)`*. For information about `Options` and `YeccRet`, see *`yecc:file/2`*.

See Also

`filename(3)`, *`compile(3)`*, *`erlang(3)`*, *`yecc(3)`*, *`xref(3)`*

calendar

Erlang module

This module provides computation of local and universal time, day of the week, and many 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` in this module both return date and time. The is because separate functions for date and time can result in a date/time combination that is displaced by 24 hours. This occurs 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 specified date and time is the number of seconds up to and including the specified date and time.

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

Different definitions exist for the week of the year. This module contains a week of the year implementation conforming to the ISO 8601 standard. As the week number for a specified date can fall on the previous, the current, or on the next year, it is important to specify both the year and the week number. Functions `iso_week_number/0` and `iso_week_number/1` return a tuple of the year and the week number.

Data Types

datetime() = {*date()*, *time()*}
datetime1970() = {{*year1970()*, *month()*, *day()*}, *time()*}
date() = {*year()*, *month()*, *day()*}
year() = integer() >= 0

Year cannot be abbreviated. For example, 93 denotes year 93, not 1993. The valid range depends on the underlying operating system. The date tuple must denote a valid date.

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

Exports

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

Types:

Date = *date()*
Year = *year()*
Month = *month()*
Day = *day()*

Computes the number of gregorian days starting with year 0 and ending at the specified date.

datetime_to_gregorian_seconds(DateTime) -> Seconds

Types:

DateTime = *datetime()*
Seconds = integer() >= 0

Computes the number of gregorian seconds starting with year 0 and ending at the specified date and time.

day_of_the_week(Date) -> *daynum()*
day_of_the_week(Year, Month, Day) -> *daynum()*

Types:

Date = *date()*
Year = *year()*
Month = *month()*
Day = *day()*

Computes the day of the week from the specified `Year`, `Month`, and `Day`. Returns the day of the week as `1`: Monday, `2`: Tuesday, and so on.

gregorian_days_to_date(Days) -> *date()*

Types:

Days = integer() >= 0

Computes the date from the specified number of gregorian days.

gregorian_seconds_to_datetime(Seconds) -> *datetime()*

Types:

Seconds = integer() >= 0

Computes the date and time from the specified number of gregorian seconds.

is_leap_year(Year) -> boolean()

Types:

Year = *year()*

Checks if the specified year is a leap year.

iso_week_number() -> *yearweeknum()*

Returns tuple `{Year, WeekNum}` representing the ISO week number for the actual date. To determine the actual date, use function *`local_time/0`*.

iso_week_number(Date) -> *yearweeknum()*

Types:

Date = *date()*

Returns tuple `{Year, WeekNum}` representing the ISO week number for the specified date.

last_day_of_the_month(Year, Month) -> LastDay

Types:

Year = *year()*
Month = *month()*
LastDay = *ldom()*

Computes the number of days in a month.

local_time() -> *datetime()*

Returns the local time reported by the underlying operating system.

local_time_to_universal_time(DateTime1) -> DateTime2

Types:

DateTime1 = DateTime2 = *datetime1970()*

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, as it is 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()*

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, as the local time is illegal (it has never occured).

`[DstDateTimeUTC, DateTimeUTC]`

For a local `{Date1, Time1}` during the period that is repeated when switching **from** daylight saving time, two corresponding UTCs exist; 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 only one corresponding UTC exists.

now_to_datetime(Now) -> *datetime1970()*

Types:

Now = *erlang:timestamp()*

Returns Universal Coordinated Time (UTC) converted from the return value from *`erlang:timestamp/0`*.

now_to_local_time(Now) -> *datetime1970()*

Types:

Now = *erlang:timestamp()*

Returns local date and time converted from the return value from *`erlang:timestamp/0`*.

now_to_universal_time(Now) -> *datetime1970()*

Types:

Now = *erlang:timestamp()*

Returns Universal Coordinated Time (UTC) converted from the return value from *`erlang:timestamp/0`*.

seconds_to_daystime(Seconds) -> {Days, Time}

Types:

Seconds = Days = integer()
Time = *time()*

Converts a specified number of seconds into days, hours, minutes, and seconds. `Time` is always non-negative, but `Days` is negative if argument `Seconds` is.

seconds_to_time(Seconds) -> *time()*

Types:

Seconds = *secs_per_day()*
secs_per_day() = 0..86400

Computes the time from the specified number of seconds. `Seconds` must be less than the number of seconds per day (86400).

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

Types:

`T1 = T2 = datetime()`
`Days = integer()`
`Time = time()`

Returns the difference between two `{Date, Time}` tuples. `T2` is to 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) -> secs_per_day()`

Types:

`Time = time()`
`secs_per_day() = 0..86400`

Returns the number of seconds since midnight up to the specified time.

`universal_time() -> datetime()`

Returns the Universal Coordinated Time (UTC) reported by the underlying operating system. Returns local time if universal time is unavailable.

`universal_time_to_local_time(DateTime) -> datetime()`

Types:

`DateTime = datetime1970()`

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

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

Types:

`Date = date()`
`Year = Month = Day = 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 one of the following rules is valid:

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

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

Date and Time Source

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

The following fapply:

- 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

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

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

Three types of Dets tables exist:

- `set`. A table of this type 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.
- `bag`. A table of this type has zero or more different objects with a given key.
- `duplicate_bag`. A table of this type 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 is not properly closed, Dets automatically repairs 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 many Erlang processes (users) open the same Dets table, they 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 terminates abnormally.

Note:

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

As 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. The Dets functions are therefore much slower than the corresponding *`ets(3)`* 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 option `repair` set to `force`.

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

All Dets functions return `{error, Reason}` if an error occurs (*`first/1`* and *`next/2`* are exceptions, they exit the process with the error tuple). If badly formed arguments are specified, all functions exit the process with a `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 section *Match Specification in Erlang* in ERTS User's Guide and the *ms_transform(3)* module.

no_slots() = default | integer() >= 0
object() = tuple()
object_cont()

Opaque continuation used by *match_object/1* and *match_object/3*.

pattern() = atom() | tuple()

For a description of patterns, see *ets:match/2*.

select_cont()

Opaque continuation used by *select/1* and *select/3*.

tab_name() = term()
type() = bag | duplicate_bag | set

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 specifying value `bchunk` to option `format` of function *init_table/3* The Mnesia application uses this function for copying open tables.

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

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

`bchunk/2` returns a tuple `{Continuation2, Data}`, where `Data` is a list of objects. `Continuation2` is another continuation that is to be passed on to a subsequent call to `bchunk/2`. With a series of calls to `bchunk/2`, all table objects can be extracted.

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

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

Types:

Name = *tab_name()*

Reason = term()

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 that is not properly closed, Dets automatically tries to repair it.

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

Types:

Name = *tab_name()*

Key = Reason = term()

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

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

Types:

Name = *tab_name()*

Reason = term()

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

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

Types:

Name = *tab_name()*

Object = *object()*

Reason = term()

Deletes all instances of a specified object from a table. If a table is of type `bag` or `duplicate_bag`, this function can be used to delete only some of the objects with a specified key.

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

Types:

Name = *tab_name()*

Key = term()

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

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

If an error occurs, the process is exited with an error tuple `{error, Reason}`. The error tuple is not returned, as it cannot be distinguished from a key.

There are two reasons why `first/1` and `next/2` are not to be used: they are not efficient, and they prevent the use of key `'$end_of_table'`, as this atom is used to indicate the end of the table. If possible, use functions *`match`*, *`match_object`*, and *`select`* 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 table `Name` together with an extra argument `AccIn`. The table elements are traversed in unspecified order. `Function` must return a new accumulator that 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 table `Name` and then inserts all the objects of the ETS table `EtsTab`. The objects are inserted in unspecified order. As `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 table `Name` as a list of tuples:

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

```
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 |
        safe_fixed_monotonic_time |
        size |
        type
    Value = term()
```

Returns the information associated with `Item` for 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 autosave 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`.
- `{hash, Hash}` - Describes which BIF is used to calculate the hash values of the objects stored in the Dets table. Possible values of `Hash`:
  - `phash` - Implies that the `erlang:phash/2` BIF is used.
  - `phash2` - Implies that the `erlang:phash2/1` BIF is used.
- `{memory, integer() >= 0}` - The file size, in bytes. The same value is associated with item `file_size`.
- `{no_keys, integer >= 0()}` - The number of different keys stored in the table.
- `{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.
- `{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_monotonic_time, 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 can be any number of processes in the list. If the table is not fixed, `SafeFixed` is the atom `false`.

  `FixedAtTime` corresponds to the result returned by *`erlang:monotonic_time/0`* at the time of fixation. The use of `safe_fixed_monotonic_time` is *time warp safe*.
- `{safe_fixed, SafeFixed}` - The same as `{safe_fixed_monotonic_time, SafeFixed}` except the format and value of `FixedAtTime`.

  `FixedAtTime` corresponds to the result returned by *`erlang:timestamp/0`* at the time of fixation. Notice that when the system uses single or multi *time warp modes*, this can produce strange results. This is because the use of `safe_fixed` is not *time warp safe*. Time warp safe code must use `safe_fixed_monotonic_time` instead.

```
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 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. Notice that the input functions are called by the process that handles requests to the Dets table, not by the calling process.

When called with argument `read`, 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 is called exactly once, and if an error occurs, the last function is called with argument `close`, the reply of which is ignored.

If the table type is `set` and more than one object exists 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. Avoid duplicate keys, otherwise the file becomes 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 starts to grow when `init_table/2` returns, which significantly slows 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 option `min_no_slots` below.

Argument `Options` 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 to be stored in the table. The `open_file/2` option with the same name is ignored, unless the table is created, in which case performance can be enhanced by supplying an estimate when initializing the table.
- `{format, Format}` - Specifies the format of the objects returned by 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 option `min_no_slots`.

`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 table `Name`. If there already exists some object with a key matching the key of any of the specified 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 table `Name`, using *`init_table/3`* with 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 file `Filename` is a Dets table, otherwise `false`.

lookup(Name, Key) -> Objects | {error, Reason}

Types:

Name = *tab_name()*
Key = term()
Objects = [*object()*]
Reason = term()

Returns a list of all objects with key `Key` stored in table `Name`, for example:

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

If the table type is `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 type is `bag` or `duplicate_bag`, the function returns a list of arbitrary length.

Notice 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 matching a specified 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 previous call to `match/1` or `match/3`.

When all table objects are 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 table `Name` that matches `Pattern` a list of bindings in some unspecified order. For a description of patterns, see *`ets:match/2`*. If the keypos'th element of `Pattern` is unbound, all table objects are matched. If the keypos'th element is bound, only the objects with the correct 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 table `Name` and returns a non-empty list of the bindings that match `Pattern` in some unspecified order. For a description of patterns, see *`ets:match/2`*.

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 table objects are matched. If the keypos'th element is unbound, all table objects are matched, `N` objects at a time, until at least one object matches or the end of the table is 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. All objects with the same key are always matched at the same time, which implies that more than N objects can sometimes be matched.

The table is always to be protected using *`safe_fixtable/2`* before calling `match/3`, otherwise errors can 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 table `Name`. For a description of patterns, see *`ets:match/2`*.

If the keypos'th element of `Pattern` is bound, only the objects with the correct 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 previous call to `match_object/1` or `match_object/3`.

When all table objects are 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 table `Name` that match `Pattern` in some unspecified order. For a description of patterns, see *`ets:match/2`*.

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

Using the `match_object` functions for traversing all table objects 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 table `Name` and returns a non-empty list of the objects that match `Pattern` in some unspecified order. For a description of patterns, see *`ets:match/2`*.

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 table objects are matched. If the keypos'th element is unbound, all table objects are matched, `N` objects at a time, until at least one object matches or the end of the table is 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. All matching objects with the same key are always returned in the same reply, which implies that more than N objects can sometimes be returned.

The table is always to be protected using *`safe_fixtable/2`* before calling `match_object/3`, otherwise errors can 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. Returns `true` if one or more table elements has key `Key`, otherwise `false`.

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

Types:

Name = *tab_name()*
Key1 = Key2 = term()

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

If an error occurs, the process is exited with an error tuple `{error, Reason}`.

To find the first key in the table, use *`first/1`*.

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

Types:

Filename = *file:name()*
Reference = reference()
Reason = term()

Opens an existing table. If the table is not properly closed, it is repaired. The returned reference is to be used as the table name. 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()}
Reason = term()
```

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

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

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

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

- `{access, access()}` - Existing tables can be opened in read-only mode. A table that is opened in read-only mode is not subjected to the automatic file reparation algorithm if it is later opened after a crash. Defaults to `read_write`.
- `{auto_save, auto_save()}` - The autosave 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 requires no reparation when reopened after an uncontrolled emulator halt. If the interval is the atom `infinity`, autosave is disabled. Defaults to 180000 (3 minutes).
- `{estimated_no_objects, no_slots()}` - Equivalent to option `min_no_slots`.
- `{file, file:name()}` - The name of the file to be opened. Defaults to the table name.
- `{max_no_slots, no_slots()}` - The maximum number of slots to be used. Defaults to 32 M, which is the maximal value. Notice that a higher value can increase the table fragmentation, and a smaller value can decrease the fragmentation, at the expense of execution time.
- `{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 to be stored in the table. Defaults to 256, which is the minimum value.
- `{keypos, keypos()}` - The position of the element of each object to be used as key. Defaults to 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 can sound like an anomaly, but can enhance the performance of applications that 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. Defaults to `false`.
- `{repair, Value}` - `Value` can be either a `boolean()` or the atom `force`. The flag specifies if the Dets server is to invoke the automatic file reparation algorithm. Defaults to `true`. If `false` is specified, no attempt is made to repair the file, and `{error, {needs_repair, FileName}}` is returned if the table must be repaired.

Value `force` means that a reparation is made even if the table is properly closed. This is a seldom needed option.

Option `repair` is ignored if the table is already open.

- `{type, type()}` - The table type. Defaults to `set`.

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

Types:

`Pid = pid()`
`Name = tab_name()`

Returns the table name 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 are 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.

For more information and examples, see the *`ets(3)`* module.

Note:

This function is rarely needed in application code. It is used by application Mnesia to provide 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 can be subject to change in future releases, while this interface remains for backward compatibility.

`safe_fixtable(Name, Fix) -> ok`

Types:

`Name = tab_name()`
`Fix = boolean()`

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

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

It is not guaranteed that calls to `first/1`, `next/2`, or select and match functions work as expected even if the table is fixed; the limited support for concurrency provided by the *`ets(3)`* module is not yet provided by 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 starts to grow when the table is released, which significantly slows down access to the table for a period of time.

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

Types:

```
Continuation = Continuation2 = select_cont()
Selection = [term()]
Reason = term()
```

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

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

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

Types:

```
Name = tab_name()
MatchSpec = match_spec()
Selection = [term()]
Reason = term()
```

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

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

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

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

Types:

```
Name = tab_name()
MatchSpec = match_spec()
N = default | integer() >= 0
Continuation = select_cont()
Selection = [term()]
Reason = term()
```

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

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

If the keypos'th element of `MatchSpec` is bound, the match specification is applied to all objects of the table with the correct key(s). If the keypos'th element of `MatchSpec` is unbound, the match specification is applied to all objects of the table, `N` objects at a time, until at least one object matches or the end of the table is 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. All objects with the same key are always handled at the same time, which implies that the match specification can be applied to more than N objects.

The table is always to be protected using *`safe_fixtable/2`* before calling `select/3`, otherwise errors can occur when calling `select/1`.

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

Types:

`Name = tab_name()`
`MatchSpec = match_spec()`
`N = integer() >= 0`
`Reason = term()`

Deletes each object from table `Name` such that applying match specification `MatchSpec` to the object returns value `true`. For a description of match specifications, see the *ERTS User's Guide*. 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 correct 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`. Returns the list of objects associated with slot `I`. If `I` > `n`, `'$end_of_table'` is returned.

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

Types:

`Name = tab_name()`
`Reason = term()`

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

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

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

Types:

Name = *tab_name()*
Options = Option | [Option]
Option = {n_objects, Limit} | {traverse, TraverseMethod}
Limit = default | integer() >= 1
TraverseMethod = first_next | select | {select, *match_spec()*}
QueryHandle = *qlc:query_handle()*

Returns a Query List Comprehension (QLC) query handle. The *qlc(3)* module provides a query language aimed mainly for Mnesia, but ETS tables, Dets tables, and lists are also recognized by qlc as sources of data. Calling *dets:table/1,2* is the means to make Dets table Name usable to qlc.

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

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

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

```
1> dets:open_file(t, []),
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 equivalent to the former, which can be verified using function qlc:info/1:

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

qlc:info/1 returns information about a query handle. 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 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 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 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 returned immediately.

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

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

See Also

ets(3), *mnesia(3)*, *qlc(3)*

dict

Erlang module

This module provides a `Key-Value` dictionary. The representation of a dictionary is not defined.

This module provides the same interface as the *`orddict(3)`* module. 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(term(), term())`

Exports

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

Types:

`Dict1 = Dict2 = dict(Key, Value)`

Appends a new `Value` to the current list of values associated with `Key`.

See also section *Notes*.

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

Types:

`Dict1 = Dict2 = dict(Key, Value)`
`ValList = [Value]`

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.

See also section *Notes*.

`erase(Key, Dict1) -> Dict2`

Types:

`Dict1 = Dict2 = dict(Key, Value)`

Erases all items with a given key from a dictionary.

`fetch(Key, Dict) -> Value`

Types:

`Dict = dict(Key, Value)`

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

See also section *Notes*.

fetch_keys(Dict) -> Keys

Types:

Dict = *dict*(Key, Value :: term())
Keys = [Key]

Returns a list of all keys in dictionary `Dict`.

take(Key, Dict) -> {Value, Dict1} | error

Types:

Dict = Dict1 = *dict*(Key, Value)
Key = Value = term()

This function returns value from dictionary and a new dictionary without this value. Returns `error` if the key is not present in the dictionary.

filter(Pred, Dict1) -> Dict2

Types:

Pred = fun((Key, Value) -> boolean())
Dict1 = Dict2 = *dict*(Key, Value)

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

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

Types:

Dict = *dict*(Key, Value)

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

See also section *Notes*.

fold(Fun, Acc0, Dict) -> Acc1

Types:

Fun = fun((Key, Value, AccIn) -> AccOut)
Dict = *dict*(Key, Value)
Acc0 = Acc1 = AccIn = AccOut = Acc

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

from_list(List) -> Dict

Types:

Dict = *dict*(Key, Value)
List = [{Key, Value}]

Converts the `Key-Value` list `List` to dictionary `Dict`.

is_empty(Dict) -> boolean()

Types:

Dict = *dict()*

Returns `true` if dictionary `Dict` has no elements, otherwise `false`.

is_key(Key, Dict) -> boolean()

Types:

Dict = *dict*(Key, Value :: term())

Tests if `Key` is contained in dictionary `Dict`.

map(Fun, Dict1) -> Dict2

Types:

Fun = fun((Key, Value1) -> Value2)
Dict1 = *dict*(Key, Value1)
Dict2 = *dict*(Key, Value2)

Calls `Fun` on successive keys and values of dictionary `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)

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, `Fun` is called with the key and both values to return a new value. `merge` can be defined as follows, but is faster:

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

new() -> *dict()*

Creates a new dictionary.

size(Dict) -> integer() >= 0

Types:

Dict = *dict()*

Returns the number of elements in dictionary `Dict`.

store(Key, Value, Dict1) -> Dict2

Types:

Dict1 = Dict2 = *dict*(Key, Value)

Stores a `Key-Value` pair in dictionary `Dict2`. If `Key` already exists in `Dict1`, the associated value is replaced by `Value`.

`to_list(Dict) -> List`

Types:

`Dict = dict(Key, Value)`
`List = [{Key, Value}]`

Converts dictionary `Dict` to a list representation.

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

Types:

`Dict1 = Dict2 = dict(Key, Value)`
`Fun = fun((Value1 :: Value) -> Value2 :: Value)`

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

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

Types:

`Dict1 = Dict2 = dict(Key, Value)`
`Fun = fun((Value1 :: Value) -> Value2 :: Value)`
`Initial = Value`

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

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

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

Types:

`Dict1 = Dict2 = dict(Key, Value)`
`Increment = number()`

Adds `Increment` to the value associated with `Key` and stores this value. If `Key` is not present in the dictionary, `Increment` is stored as the first value.

This can be defined as follows, but is faster:

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

Notes

Functions `append` and `append_list` are included so that keyed values can be stored in a list **accumulator**, for example:

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

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

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

See Also

gb_trees(3), *orddict(3)*

digraph

Erlang module

This module provides a version of labeled directed graphs. What makes the graphs provided here non-proper directed graphs is that multiple edges between vertices are allowed. However, the customary definition of directed graphs is used here.

- 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 more information. Such information can be attached to the vertices and to the edges of the digraph. An annotated digraph 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 an edge is emanating from v and incident on w, then w is said to be an **out-neighbor** of v, and v is said to be an **in-neighbor** 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 path P is k-1.
- Path P is **simple** if all vertices are distinct, except that the first and the last vertices can be the same.
- Path 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 without 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()_]} | {bad_vertex, V :: _vertex()_}

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

`add_edge(G, V1, V2, Label)` is equivalent to `add_edge(G, E, V1, V2, Label)`, where `E` is a created edge. The created edge is represented by term `['$e' | N]`, where `N` is an integer >= 0.

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

If the edge would create a cycle in an *acyclic digraph*, `{error, {bad_edge, Path}}` is returned. If `G` already has an edge with value `E` connecting a different pair of vertices, `{error, {bad_edge, [V1, V2]}}` is returned. If either of `V1` or `V2` is not a vertex of digraph `G`, `{error, {bad_vertex, V}}` is returned, V = `V1` or V = `V2`.

add_vertex(G) -> _vertex()_
add_vertex(G, V) -> _vertex()_
add_vertex(G, V, Label) -> _vertex()_

Types:

G = _graph()_
V = _vertex()_
Label = _label()_

`add_vertex/3` creates (or modifies) vertex `V` of digraph `G`, using `Label` as the (new) *label* of the vertex. Returns `V`.

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

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

del_edge(G, E) -> true

Types:

G = _graph()_
E = _edge()_

Deletes edge `E` from digraph `G`.

del_edges(G, Edges) -> true

Types:

G = _graph()_
Edges = [_edge()_]

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

del_path(G, V1, V2) -> true

Types:

G = *graph()*
V1 = V2 = *vertex()*

Deletes edges from digraph G until there are no *paths* from vertex V1 to vertex V2.

A sketch of the procedure employed:

- Find an arbitrary *simple path* v[1], v[2], ..., v[k] from V1 to V2 in G.
- Remove all edges of G *emanating* from v[i] and *incident* to v[i+1] for 1 <= i < k (including multiple edges).
- Repeat until there is no path between V1 and V2.

del_vertex(G, V) -> true

Types:

G = *graph()*
V = *vertex()*

Deletes vertex V from 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 list Vertices from digraph G.

delete(G) -> true

Types:

G = *graph()*

Deletes digraph G. This call is important as digraphs are implemented with ETS. There is no garbage collection of ETS tables. However, the digraph is 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 edge E *emanating* from V1 and *incident* on V2 of digraph G. If no edge E of digraph G exists, false is returned.

edges(G) -> Edges

Types:

G = *graph()*
Edges = [*edge()*]

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

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

Types:

```
G = graph()
V = vertex()
Vertices = [vertex(), ...]
```

If a *simple cycle* of length two or more exists through vertex `V`, the cycle is returned as a list `[V, ..., V]` of vertices. If a *loop* through `V` exists, the loop is returned as a list `[V]`. If no cycles through `V` exist, `false` is returned.

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

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

Types:

```
G = graph()
V1 = V2 = vertex()
Vertices = [vertex(), ...]
```

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

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

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

Types:

```
G = graph()
V = vertex()
Vertices = [vertex(), ...]
```

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

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

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

Types:

```
G = graph()
V1 = V2 = vertex()
Vertices = [vertex(), ...]
```

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

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

`in_degree(G, V) -> integer() >= 0`

Types:

```
G = graph()
V = vertex()
```

Returns the *in-degree* of vertex `V` of digraph `G`.

`in_edges(G, V) -> Edges`

Types:

```
G = graph()
V = vertex()
Edges = [edge()]
```

Returns a list of all edges *incident* on `V` of digraph `G`, in some unspecified order.

`in_neighbours(G, V) -> Vertex`

Types:

```
G = graph()
V = vertex()
Vertex = [vertex()]
```

Returns a list of all *in-neighbors* of `V` of 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 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`

Allows *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 specified or `Type` is not a proper list, a `badarg` exception is raised.

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

Types:

```
G = graph()
```

Returns the number of edges of digraph `G`.

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

Types:

```
G = graph()
```

Returns the number of vertices of digraph `G`.

`out_degree(G, V) -> integer() >= 0`

Types:

```
G = graph()
V = vertex()
```

Returns the *out-degree* of vertex `V` of digraph `G`.

`out_edges(G, V) -> Edges`

Types:

```
G = graph()
V = vertex()
Edges = [edge()]
```

Returns a list of all edges *emanating* from `V` of digraph `G`, in some unspecified order.

`out_neighbours(G, V) -> Vertices`

Types:

G = *graph()*
V = *vertex()*
Vertices = [*vertex()*]

Returns a list of all *out-neighbors* of V of digraph G, in some unspecified order.

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

Types:

G = *graph()*
V = *vertex()*
Label = *label()*

Returns {V, Label}, where Label is the *label* of the vertex V of digraph G, or false if no vertex V of digraph G exists.

vertices(G) -> Vertices

Types:

G = *graph()*
Vertices = [*vertex()*]

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

See Also

digraph_utils(3), *ets(3)*

digraph_utils

Erlang module

This module provides algorithms based on depth-first traversal of directed graphs. For basic functions on directed graphs, see the *digraph(3)* module.

- 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 more information. Such information can be attached to the vertices and to the edges of the digraph. An annotated digraph 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 an edge is emanating from v and incident on w, then w is said to be an **out-neighbor** of v, and v is said to be an **in-neighbor** 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 path P is k-1.
- Path 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 without 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 unvisited vertices remain when all edges from the first vertex have been examined, some so far 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 that the version of directed graphs provided by the `digraph` module allows multiple edges between vertices). If the digraph has no cycles of length two or more, the reflexive and transitive closure of E is a partial ordering.
- A **subgraph** G' of G is a digraph whose vertices and edges form subsets of the vertices and edges of G.
- G' is **maximal** with respect to a property P if all other subgraphs that include the vertices of G' do not have property P.
- A **strongly connected component** is a maximal subgraph such that there is a path between each pair of vertices.
- A **connected component** is a maximal subgraph such that there is a path between each pair of vertices, considering all edges undirected.
- An **arborescence** is an acyclic digraph with a vertex V, the **root**, such that there is a unique path from V to every other vertex of G.
- A **tree** is an acyclic non-empty digraph such that there is a unique path between every pair of vertices, considering all edges undirected.

Exports

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

Types:

Digraph = *digraph:graph()*
Root = *digraph:vertex()*

Returns `{yes, Root}` if `Root` is the *root* of the arborescence `Digraph`, otherwise `no`.

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 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 vertices x and y exist 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 *cycle* is included in some strongly connected component, which implies that a *topological ordering* of the created digraph always exists.

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 digraph `Digraph` is *acyclic*.

is_arborescence(Digraph) -> boolean()

Types:

Digraph = *digraph:graph()*

Returns `true` if and only if digraph `Digraph` is an *arborescence*.

is_tree(Digraph) -> boolean()

Types:

Digraph = *digraph:graph()*

Returns `true` if and only if 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*.

postorder(Digraph) -> Vertices

Types:

Digraph = *digraph:graph()*
Vertices = [*digraph:vertex()*]

Returns all vertices of 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.

preorder(Digraph) -> Vertices

Types:

Digraph = *digraph:graph()*
Vertices = [*digraph:vertex()*]

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

reachable(Vertices, Digraph) -> 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, as paths can have length zero, the vertices of `Vertices` are included in the returned list.

reachable_neighbours(Vertices, Digraph) -> 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.

reaching(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* from the vertex to some vertex of `Vertices`. In particular, as paths can 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`. Therefore 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 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 option `type` is `inherit`, which is the default, 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 option `keep_labels` is `true`, which is the default, the *labels* of vertices and edges of `Digraph` are used for the subgraph as well. If the value is `false`, default label `[]` is used for the vertices and edges of the subgroup.

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

If any of the arguments are invalid, a `badarg` exception is raised.

topsort(Digraph) -> Vertices | false

Types:

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

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

See Also

digraph(3)

epp

Erlang module

The Erlang code preprocessor includes functions that are used by the *compile* module to preprocess macros and include files before the 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 matching the regular expression `coding\s*[:=]\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:

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

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

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

Data Types

macros() = [atom() | {atom(), term()}]

epp_handle() = pid()

Handle to the epp server.

source_encoding() = latin1 | utf8

Exports

close(Epp) -> ok

Types:

Epp = *epp_handle()*

Closes the preprocessing of a file.

default_encoding() -> *source_encoding()*

Returns the default encoding of Erlang source files.

encoding_to_string(Encoding) -> string()

Types:

Encoding = *source_encoding()*

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

format_error(ErrorDescriptor) -> *io_lib:chars()*

Types:

```
ErrorDescriptor = term()
```

Takes an `ErrorDescriptor` and returns a string that describes the error or warning. This function is usually called implicitly when processing an `ErrorInfo` structure (see section *Error Information*).

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

Types:

```
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 specified in `Options`, the return value is `{ok, Epp, Extra}` instead of `{ok, Epp}`.

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

Types:

```
FileName = file:name()
IncludePath = [DirectoryName :: file:name()]
Epp = epp_handle()
ErrorDescriptor = term()
```

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

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

Types:

```
Epp = epp_handle()
AbsForm = erl_parse:abstract_form()
Line = erl_anno:line()
ErrorInfo = erl_scan:error_info() | erl_parse:error_info()
WarningInfo = warning_info()
warning_info() = {erl_anno:location(), module(), term()}
```

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

```
parse_file(FileName, Options) ->
              {ok, [Form]} |
              {ok, [Form], Extra} |
              {error, OpenError}
```

Types:

```
FileName = file:name()
Options =
    [{includes, IncludePath :: [DirectoryName :: file:name()]} |
     {macros, PredefMacros :: macros()} |
     {default_encoding, DefEncoding :: source_encoding()} |
     extra]
Form =
    erl_parse:abstract_form() | {error, ErrorInfo} | {eof, Line}
Line = erl_anno:line()
ErrorInfo = erl_scan:error_info() | erl_parse:error_info()
Extra = [{encoding, source_encoding() | none}]
OpenError = file:posix() | badarg | system_limit
```

Preprocesses and parses an Erlang source file. Notice that tuple `{eof, Line}` returned at the end of the file is included as a "form".

If `extra` is specified in `Options`, the return value is `{ok, [Form], Extra}` instead of `{ok, [Form]}`.

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

Types:

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

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

read_encoding(FileName) -> *source_encoding()* | none
read_encoding(FileName, Options) -> *source_encoding()* | none

Types:

FileName = *file:name()*
Options = [Option]
Option = {in_comment_only, boolean()}

Read the *encoding* from a file. Returns the read encoding, or `none` if no valid encoding is found.

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.

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 is found.

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.

set_encoding(File) -> *source_encoding()* | none

Types:

File = *io:device()*

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

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

set_encoding(File, Default) -> *source_encoding()* | none

Types:

Default = *source_encoding()*
File = *io:device()*

Reads the *encoding* from an I/O device and sets the encoding of the device accordingly. The position of the I/O device referenced by `File` is not affected. If no valid encoding can be read from the I/O device, the encoding of the I/O device is set to the *encoding* specified by `Default`.

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

Error Information

`ErrorInfo` is the standard `ErrorInfo` structure that is returned from all I/O modules. The format is as follows:

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

A string describing the error is obtained with the following call:

```
Module:format_error(ErrorDescriptor)
```

See Also

erl_parse(3)

erl_anno

Erlang module

This module provides an abstract type that is used by the Erlang Compiler and its helper modules for holding data such as column, line number, and text. The data type is a collection of **annotations** as described in the following.

The Erlang Token Scanner returns tokens with a subset of the following annotations, depending on the options:

`column`

The column where the token begins.

`location`

The line and column where the token begins, or just the line if the column is unknown.

`text`

The token's text.

From this, the following annotation is derived:

`line`

The line where the token begins.

This module also supports the following annotations, which are used by various modules:

`file`

A filename.

`generated`

A Boolean indicating if the abstract code is compiler-generated. The Erlang Compiler does not emit warnings for such code.

`record`

A Boolean indicating if the origin of the abstract code is a record. Used by *Dialyzer* to assign types to tuple elements.

The functions *`column()`*, *`end_location()`*, *`line()`*, *`location()`*, and *`text()`* in the `erl_scan` module can be used for inspecting annotations in tokens.

The functions *`anno_from_term()`*, *`anno_to_term()`*, *`fold_anno()`*, *`map_anno()`*, *`mapfold_anno()`*, and *`new_anno()`*, in the `erl_parse` module can be used for manipulating annotations in abstract code.

Data Types

`anno()`

A collection of annotations.

`anno_term() = term()`

The term representing a collection of annotations. It is either a `location()` or a list of key-value pairs.

```
column() = integer() >= 1
line() = integer() >= 0
location() = line() | {line(), column()}
text() = string()
```

Exports

column(Anno) -> *column()* | undefined

Types:

Anno = *anno()*

column() = integer() >= 1

Returns the column of the annotations Anno.

end_location(Anno) -> *location()* | undefined

Types:

Anno = *anno()*

location() = *line()* | {*line()*, *column()*}

Returns the end location of the text of the annotations Anno. If there is no text, `undefined` is returned.

file(Anno) -> *filename()* | undefined

Types:

Anno = *anno()*

filename() = *file:filename_all()*

Returns the filename of the annotations Anno. If there is no filename, `undefined` is returned.

from_term(Term) -> Anno

Types:

Term = *anno_term()*

Anno = *anno()*

Returns annotations with representation Term.

See also *to_term()*.

generated(Anno) -> *generated()*

Types:

Anno = *anno()*

generated() = boolean()

Returns `true` if annotations Anno is marked as generated. The default is to return `false`.

is_anno(Term) -> boolean()

Types:

Term = any()

Returns `true` if Term is a collection of annotations, otherwise `false`.

line(Anno) -> *line()*

Types:

Anno = *anno()*
line() = integer() >= 0

Returns the line of the annotations Anno.

location(Anno) -> *location()*

Types:

Anno = *anno()*
location() = *line()* | {*line()*, *column()*}

Returns the location of the annotations Anno.

new(Location) -> *anno()*

Types:

Location = *location()*
location() = *line()* | {*line()*, *column()*}

Creates a new collection of annotations given a location.

set_file(File, Anno) -> Anno

Types:

File = *filename()*
Anno = *anno()*
filename() = *file:filename_all()*

Modifies the filename of the annotations Anno.

set_generated(Generated, Anno) -> Anno

Types:

Generated = *generated()*
Anno = *anno()*
generated() = boolean()

Modifies the generated marker of the annotations Anno.

set_line(Line, Anno) -> Anno

Types:

Line = *line()*
Anno = *anno()*
line() = integer() >= 0

Modifies the line of the annotations Anno.

set_location(Location, Anno) -> Anno

Types:

Location = *location()*
Anno = *anno()*
location() = *line()* | {*line()*, *column()*}

Modifies the location of the annotations Anno.

set_record(Record, Anno) -> Anno

Types:

Record = *record()*
Anno = *anno()*
record() = boolean()

Modifies the record marker of the annotations Anno.

set_text(Text, Anno) -> Anno

Types:

Text = *text()*
Anno = *anno()*
text() = string()

Modifies the text of the annotations Anno.

text(Anno) -> *text()* | undefined

Types:

Anno = *anno()*
text() = string()

Returns the text of the annotations Anno. If there is no text, `undefined` is returned.

to_term(Anno) -> *anno_term()*

Types:

Anno = *anno()*

Returns the term representing the annotations Anno.

See also *from_term()*.

See Also

erl_parse(3), *erl_scan(3)*

erl_eval

Erlang module

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

Data Types

```
bindings() = [{name(), value()}]
binding_struct() = orddict:orddict()
```

A binding structure.

```
expression() = erl_parse:abstract_expr()
expressions() = [erl_parse:abstract_expr()]
```

As returned by *erl_parse:parse_exprs/1* or *io:parse_erl_exprs/2*.

```
expression_list() = [expression()]
func_spec() =
    {Module :: module(), Function :: atom()} | function()
lfun_eval_handler() =
    fun((Name :: atom(),
         Arguments :: expression_list(),
         Bindings :: binding_struct()) ->
            {value,
             Value :: value(),
             NewBindings :: binding_struct()})
lfun_value_handler() =
    fun((Name :: atom(), Arguments :: [term()]) ->
            Value :: value())
local_function_handler() =
    {value, lfun_value_handler()} |
    {eval, lfun_eval_handler()} |
    none
```

Further described in section *Local Function Handler* in this module

```
name() = term()
nlfun_handler() =
    fun((FuncSpec :: func_spec(), Arguments :: [term()]) -> term())
non_local_function_handler() = {value, nlfun_handler()} | none
```

Further described in section *Non-Local Function Handler* in this module.

```
value() = term()
```

Exports

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

Types:

Name = *name()*
Value = *value()*
BindingStruct = *binding_struct()*

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

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

Types:

Name = *name()*
BindingStruct = *binding_struct()*

Returns the binding of `Name` in `BindingStruct`.

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

Returns the list of bindings contained in the 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.

```
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
```

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. For an explanation of when and how to use arguments `LocalFunctionHandler` and

NonLocalFunctionHandler, see sections *Local Function Handler* and *Non-Local Function Handler* in this module.

Returns {value, Value, NewBindings} by default. If ReturnFormat is value, only Value is returned.

```
expr_list(ExpressionList, Bindings) -> {ValueList, NewBindings}
expr_list(ExpressionList, Bindings, LocalFunctionHandler) ->
             {ValueList, 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 LocalFunctionHandler, see section *Local Function Handler* in this module.

Returns {ValueList, NewBindings}.

```
exprs(Expressions, Bindings) -> {value, Value, NewBindings}
exprs(Expressions, Bindings, LocalFunctionHandler) ->
         {value, Value, NewBindings}
exprs(Expressions,
      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 that can be returned by *io:parse_erl_exprs/2*. For an explanation of when and how to use arguments LocalFunctionHandler and NonLocalFunctionHandler, see sections *Local Function Handler* and *Non-Local Function Handler* in this module.

Returns {value, Value, NewBindings}

`new_bindings() -> *binding_struct()*`

Returns an empty 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` can be used to define a function that 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 that 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, the current bindings cannot be accessed. To signal an error, the function handler calls `exit/1` with a suitable exit value.

`{eval,Func}`

This defines a local function handler that 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 calls `exit/1` with a suitable exit value.

`none`

There is no local function handler.

Non-Local Function Handler

The optional argument `NonLocalFunctionHandler` can be used to define a function that 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 function `erlang:Op/A`).

Exceptions are calls to `erlang:apply/2,3`; neither of the function handlers are called for such calls. The argument can have the following formats:

`{value,Func}`

This defines a non-local function handler that 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 calls `exit/1` with a suitable exit value.

none

There is no non-local 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])`) never takes place.

The non-local function handler **is** however 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 non-local 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.

Known Limitation

Undocumented functions in this module are not to be used.

erl_expand_records

Erlang module

This module expands records in a module.

Exports

`module(AbsForms, CompileOptions) -> AbsForms2`

Types:

`AbsForms = AbsForms2 = [`*`erl_parse:abstract_form()`*`]`
`CompileOptions = [`*`compile:option()`*`]`

Expands all records in a module to use explicit tuple operations and adds explicit module names to calls to BIFs and imported functions. The returned module has no references to records, attributes, or code.

See Also

Section *The Abstract Format* 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 wants to write their own parse transformers. If option `{parse_transform,Module}` is passed to the compiler, a user-written function `parse_transform/2` is called by the compiler before the code is checked for errors.

Exports

`parse_transform(Forms, Options) -> Forms`

Types:

`Forms = [`*`erl_parse:abstract_form()`*` | `*`erl_parse:form_info()`*`]`
`Options = [`*`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. No support is offered for problems encountered.

See Also

`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

`add_predefined_functions(Forms) -> UpdatedForms`

Types:

```
Forms = [erl_parse:abstract_form() | erl_parse:form_info()]
UpdatedForms =
    [erl_parse:abstract_form() | erl_parse:form_info()]
```

Adds to `Forms` the code for the standard pre-defined functions (such as `module_info/0`) that are to be included in every module.

`arith_op(OpName, Arity) -> boolean()`

Types:

```
OpName = atom()
Arity = arity()
```

Returns `true` if `OpName/Arity` is an arithmetic operator, otherwise `false`.

`bif(Name, Arity) -> boolean()`

Types:

```
Name = atom()
Arity = arity()
```

Returns `true` if `Name/Arity` is an Erlang BIF that is automatically recognized by the compiler, 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`.

`guard_bif(Name, Arity) -> boolean()`

Types:

Name = atom()
Arity = arity()

Returns `true` if `Name/Arity` is an Erlang BIF that is allowed in guards, otherwise `false`.

list_op(OpName, Arity) -> boolean()

Types:

OpName = atom()
Arity = arity()

Returns `true` if `OpName/Arity` is a list 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.

send_op(OpName, Arity) -> boolean()

Types:

OpName = atom()
Arity = arity()

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

erl_lint

Erlang module

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

The errors detected include:

- Redefined and undefined functions
- Unbound and unsafe variables
- Illegal record use

The warnings detected include:

- Unused functions and imports
- Unused variables
- Variables imported into matches
- Variables exported from `if`/`case`/`receive`
- Variables shadowed in funs and list comprehensions

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

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

Data Types

```
error_info() = {erl_anno:line(), module(), error_description()}
error_description() = term()
```

Exports

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

Types:

```
ErrorDescriptor = error_description()
```

Takes an `ErrorDescriptor` and returns a string that describes the error or warning. This function is usually called implicitly when processing an `ErrorInfo` structure (see section *Error Information*).

```
is_guard_test(Expr) -> boolean()
```

Types:

```
Expr = erl_parse:abstract_expr()
```

Tests if `Expr` is a legal guard test. `Expr` is an Erlang term representing the abstract form for the expression. `erl_parse:parse_exprs(Tokens)` can be used to generate a list of `Expr`.

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

```
          {ok, Warnings} | {error, Errors, Warnings}
```

Types:

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

Checks all the forms in a module for errors. It returns:

`{ok,Warnings}`

There are no errors in the module.

`{error,Errors,Warnings}`

There are errors in the module.

As this module is of interest only to the maintainers of the compiler, and to avoid the same description in two places, the elements of `Options` that control the warnings are only described in the *`compile(3)`* module.

`AbsForms` of a module, which comes from a file that is read through `epp`, the Erlang preprocessor, can come from many files. This means that any references to errors must include the filename, see the *`epp(3)`* module or parser (see the *`erl_parse(3)`* module). The returned errors and warnings have the following format:

```
[{FileName2,[ErrorInfo]}]
```

The errors and warnings are listed in the order in which they are encountered in the forms. The errors from one file can therefore be split into different entries in the list of errors.

Error Information

`ErrorInfo` is the standard `ErrorInfo` structure that is returned from all I/O modules. The format is as follows:

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

A string describing the error is obtained with the following call:

```
Module:format_error(ErrorDescriptor)
```

See Also

`epp(3)`, *`erl_parse(3)`*

erl_parse

Erlang module

This module is the basic Erlang parser that converts tokens into the abstract form of either forms (that is, top-level constructs), expressions, or terms. The Abstract Format is described in the *ERTS User's Guide*. Notice that a token list must end with the **dot** token to be acceptable to the parse functions (see the *erl_scan(3)*) module.

Data Types

abstract_clause()

Abstract form of an Erlang clause.

abstract_expr()

Abstract form of an Erlang expression.

abstract_form()

Abstract form of an Erlang form.

abstract_type()

Abstract form of an Erlang type.

```
erl_parse_tree() =
    abstract_clause() |
    abstract_expr() |
    abstract_form() |
    abstract_type()
error_description() = term()
error_info() = {erl_anno:line(), module(), error_description()}
form_info() =
    {eof, erl_anno:line()} |
    {error, erl_scan:error_info() | error_info()} |
    {warning, erl_scan:error_info() | error_info()}
```

Tuples `{error, error_info()}` and `{warning, error_info()}`, denoting syntactically incorrect forms and warnings, and `{eof, line()}`, denoting an end-of-stream encountered before a complete form had been parsed.

token() = *erl_scan:token()*

Exports

abstract(Data) -> AbsTerm

Types:

Data = term()

AbsTerm = *abstract_expr()*

Converts the Erlang data structure `Data` into an abstract form of type `AbsTerm`. This function 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_anno:line()*
AbsTerm = *abstract_expr()*
encoding_func() = fun((integer() >= 0) -> boolean())

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

Option `Line` is the line to be assigned to each node of `AbsTerm`.

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

anno_from_term(Term) -> *erl_parse_tree()* | *form_info()*

Types:

Term = term()

Assumes that `Term` is a term with the same structure as a `erl_parse` tree, but with terms, say `T`, where a `erl_parse` tree has collections of annotations. Returns a `erl_parse` tree where each term `T` is replaced by the value returned by *`erl_anno:from_term(T)`*. The term `Term` is traversed in a depth-first, left-to-right fashion.

anno_to_term(Abstr) -> term()

Types:

Abstr = *erl_parse_tree()* | *form_info()*

Returns a term where each collection of annotations `Anno` of the nodes of the `erl_parse` tree `Abstr` is replaced by the term returned by *`erl_anno:to_term(Anno)`*. The `erl_parse` tree is traversed in a depth-first, left-to-right fashion.

fold_anno(Fun, Acc0, Abstr) -> Acc1

Types:

Fun = fun((Anno, AccIn) -> AccOut)
Anno = *erl_anno:anno()*
Acc0 = Acc1 = AccIn = AccOut = term()
Abstr = *erl_parse_tree()* | *form_info()*

Updates an accumulator by applying `Fun` on each collection of annotations of the `erl_parse` tree `Abstr`. The first call to `Fun` has `AccIn` as argument, the returned accumulator `AccOut` is passed to the next call, and so on. The final value of the accumulator is returned. The `erl_parse` tree is traversed in a depth-first, left-to-right fashion.

format_error(ErrorDescriptor) -> Chars

Types:

ErrorDescriptor = *error_description()*
Chars = [char() | Chars]

Uses an `ErrorDescriptor` and returns a string that describes the error. This function is usually called implicitly when an `ErrorInfo` structure is processed (see section *Error Information*).

`map_anno(Fun, Abstr) -> NewAbstr`

Types:

`Fun = fun((Anno) -> NewAnno)`
`Anno = NewAnno = erl_anno:anno()`
`Abstr = NewAbstr = erl_parse_tree() | form_info()`

Modifies the `erl_parse` tree `Abstr` by applying `Fun` on each collection of annotations of the nodes of the `erl_parse` tree. The `erl_parse` tree is traversed in a depth-first, left-to-right fashion.

`mapfold_anno(Fun, Acc0, Abstr) -> {NewAbstr, Acc1}`

Types:

`Fun = fun((Anno, AccIn) -> {NewAnno, AccOut})`
`Anno = NewAnno = erl_anno:anno()`
`Acc0 = Acc1 = AccIn = AccOut = term()`
`Abstr = NewAbstr = erl_parse_tree() | form_info()`

Modifies the `erl_parse` tree `Abstr` by applying `Fun` on each collection of annotations of the nodes of the `erl_parse` tree, while at the same time updating an accumulator. The first call to `Fun` has `AccIn` as second argument, the returned accumulator `AccOut` is passed to the next call, and so on. The modified `erl_parse` tree and the final value of the accumulator are returned. The `erl_parse` tree is traversed in a depth-first, left-to-right fashion.

`new_anno(Term) -> Abstr`

Types:

`Term = term()`
`Abstr = erl_parse_tree() | form_info()`

Assumes that `Term` is a term with the same structure as a `erl_parse` tree, but with *locations* where a `erl_parse` tree has collections of annotations. Returns a `erl_parse` tree where each location `L` is replaced by the value returned by `erl_anno:new(L)`. The term `Term` is traversed in a depth-first, left-to-right fashion.

`normalise(AbsTerm) -> Data`

Types:

`AbsTerm = abstract_expr()`
`Data = term()`

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

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

Types:

`Tokens = [token()]`
`ExprList = [abstract_expr()]`
`ErrorInfo = error_info()`

Parses `Tokens` as if it was a list of expressions. Returns one of the following:

`{ok, ExprList}`

The parsing was successful. `ExprList` is a list of the abstract forms of the parsed expressions.

`{error, ErrorInfo}`

An error occurred.

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

Types:

`Tokens = [`*`token()`*`]`
`AbsForm =` *`abstract_form()`*
`ErrorInfo =` *`error_info()`*

Parses `Tokens` as if it was a form. Returns one of the following:

`{ok, AbsForm}`

The parsing was successful. `AbsForm` is the abstract form of the parsed form.

`{error, ErrorInfo}`

An error occurred.

`parse_term(Tokens) -> {ok, Term} | {error, ErrorInfo}`

Types:

`Tokens = [`*`token()`*`]`
`Term = term()`
`ErrorInfo =` *`error_info()`*

Parses `Tokens` as if it was a term. Returns one of the following:

`{ok, Term}`

The parsing was successful. `Term` is the Erlang term corresponding to the token list.

`{error, ErrorInfo}`

An error occurred.

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

Types:

`AbsTerm =` *`abstract_expr()`*
`MoreTokens = Tokens = [`*`token()`*`]`

Generates a list of tokens representing the abstract form `AbsTerm` of an expression. Optionally, `MoreTokens` is appended.

Error Information

`ErrorInfo` is the standard `ErrorInfo` structure that is returned from all I/O modules. The format is as follows:

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

A string describing the error is obtained with the following call:

```
Module:format_error(ErrorDescriptor)
```

See Also

`erl_anno(3)`, `erl_scan(3)`, `io(3)`, section *The Abstract Format* in the ERTS User's Guide

erl_pp

Erlang module

The functions in this module are used to generate aesthetically attractive representations of abstract forms, which are suitable for printing. All functions return (possibly deep) lists of characters and generate an error if the form is wrong.

All functions can have an optional argument, which specifies a hook that is called if an attempt is made to print an unknown form.

Data Types

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

Optional argument `HookFunction`, shown in the functions described in this module, defines a function that is called when an unknown form occurs where there is to be a valid expression. If `HookFunction` is equal to `none`, there is no hook function.

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

If `CurrentIndentation` is negative, there are 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

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

Types:

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

Same as *`form/1,2`*, but only for attribute `Attribute`.

```
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()*

Prints one expression. It is useful for implementing hooks (see section *Known Limitations*).

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()*

Same as *form/1,2*, but only for the sequence of expressions in `Expressions`.

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

Types:

Form = *erl_parse:abstract_form()* | *erl_parse:form_info()*
Options = *options()*

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

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

Types:

Function = *erl_parse:abstract_form()*
Options = *options()*

Same as *form/1,2*, but only for function `Function`.

guard(Guard) -> *io_lib:chars()*
guard(Guard, Options) -> *io_lib:chars()*

Types:

Guard = [*erl_parse:abstract_expr()*]
Options = *options()*

Same as *form/1,2*, but only for the guard test `Guard`.

Known Limitations

It is not possible to have hook functions for unknown forms at other places than expressions.

See Also

erl_eval(3), *erl_parse(3)*, *io(3)*

erl_scan

Erlang module

This module contains functions for tokenizing (scanning) characters into Erlang tokens.

Data Types

```
category() = atom()
error_description() = term()
error_info() =
    {erl_anno:location(), module(), error_description()}
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(), Anno :: erl_anno:anno(), symbol()} |
    {category(), Anno :: erl_anno:anno()}
tokens() = [token()]
tokens_result() =
    {ok, Tokens :: tokens(), EndLocation :: erl_anno:location()} |
    {eof, EndLocation :: erl_anno:location()} |
    {error,
     ErrorInfo :: error_info(),
     EndLocation :: erl_anno:location()}
```

Exports

`category(Token) -> category()`

Types:

`Token = token()`

Returns the category of `Token`.

`column(Token) -> erl_anno:column() | undefined`

Types:

`Token = token()`

Returns the column of `Token`'s collection of annotations.

`end_location(Token) -> erl_anno:location() | undefined`

Types:

Token = *token()*

Returns the end location of the text of `Token`'s collection of annotations. If there is no text, `undefined` is returned.

format_error(ErrorDescriptor) -> string()

Types:

ErrorDescriptor = *error_description()*

Uses an `ErrorDescriptor` and returns a string that describes the error or warning. This function is usually called implicitly when an `ErrorInfo` structure is processed (see section *Error Information*).

line(Token) -> *erl_anno:line()*

Types:

Token = *token()*

Returns the line of `Token`'s collection of annotations.

location(Token) -> *erl_anno:location()*

Types:

Token = *token()*

Returns the location of `Token`'s collection of annotations.

reserved_word(Atom :: atom()) -> boolean()

Returns `true` if `Atom` is an Erlang reserved word, otherwise `false`.

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 = *erl_anno:location()*

Takes the list of characters `String` and tries to scan (tokenize) them. Returns one of the following:

`{ok, Tokens, EndLocation}`

`Tokens` are the Erlang tokens from `String`. `EndLocation` is the first location after the last token.

`{error, ErrorInfo, ErrorLocation}`

An error occurred. `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, `Anno`, `EndLocation`, and `ErrorLocation` are lines. If `StartLocation` is a pair of a line and a column, `Anno` takes the form of an opaque compound data type, and `EndLocation` and `ErrorLocation` are pairs of a line and a column. The **token annotations** contain information about the column and the line where the token begins, as well

as the text of the token (if option `text` is specified), all of which can be accessed by calling *`column/1`*, *`line/1`*, *`location/1`*, and *`text/1`*.

A **token** is a tuple containing information about syntactic category, the token annotations, and the terminal symbol. For punctuation characters (such as `;` and `|`) 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, Anno, atom()}`
- `{char, Anno, char()}`
- `{comment, Anno, string()}`
- `{float, Anno, float()}`
- `{integer, Anno, integer()}`
- `{var, Anno, atom()}`
- `{white_space, Anno, string()}`

Valid options:

`{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 becomes the category of the token. If the function returns `false`, `atom` becomes the category of the unquoted atom.

`return_comments`

Return comment tokens.

`return_white_spaces`

Return white space tokens. By convention, a newline character, if present, is always the first character of the text (there cannot be more than one newline in a white space token).

`return`

Short for `[return_comments, return_white_spaces]`.

`text`

Include the token text in the token annotation. The text is the part of the input corresponding to the token.

`symbol(Token) -> *symbol()*`

Types:

`Token = *token()*`

Returns the symbol of `Token`.

`text(Token) -> *erl_anno:text()* | undefined`

Types:

`Token = *token()*`

Returns the text of `Token`'s collection of annotations. If there is no text, `undefined` is returned.

`tokens(Continuation, CharSpec, StartLocation) -> Return`
`tokens(Continuation, CharSpec, StartLocation, Options) -> Return`

Types:

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

An opaque continuation.

This is the re-entrant scanner, which scans characters until either a **dot** ('.' followed by a white space) or `eof` is reached. It returns:

`{done, Result, LeftOverChars}`

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` then takes the value `eof` as well.

`tokens(Continuation, CharSpec, StartLocation)` is equivalent to `tokens(Continuation, CharSpec, StartLocation, [])`.

For a description of the options, see *`string/3`*.

Error Information

`ErrorInfo` is the standard `ErrorInfo` structure that is returned from all I/O modules. The format is as follows:

```
{ErrorLocation, Module, ErrorDescriptor}
```

A string describing the error is obtained with the following call:

```
Module:format_error(ErrorDescriptor)
```

Notes

The continuation of the first call to the re-entrant input functions must be `[]`. For a complete description of how the re-entrant input scheme works, see Armstrong, Virding and Williams: 'Concurrent Programming in Erlang', Chapter 13.

See Also

erl_anno(3), *erl_parse(3)*, *io(3)*

erl_tar

Erlang module

This module archives and extract files to and from a tar file. This module supports reading most common tar formats, namely v7, STAR, USTAR, and PAX, as well as some of GNU tar's extensions to the USTAR format (sparse files most notably). It produces tar archives in USTAR format, unless the files being archived require PAX format due to restrictions in USTAR (such as unicode metadata, filename length, and more). As such, `erl_tar` supports tar archives produced by most all modern tar utilities, and produces tarballs which should be similarly portable.

By convention, the name of a tar file is to end in "`.tar`". To abide to the convention, add "`.tar`" to the name.

Tar files can be created in one operation using function *`create/2`* or *`create/3`*.

Alternatively, for more control, use functions *`open/2`*, *`add/3,4`*, and *`close/1`*.

To extract all files from a tar file, use function *`extract/1`*. To extract only some files or to be able to specify some more options, use function *`extract/2`*.

To return a list of the files in a tar file, use function *`table/1`* or *`table/2`*. To print a list of files to the Erlang shell, use function *`t/1`* or *`tt/1`*.

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

Unicode Support

If *`file:native_name_encoding/0`* returns `utf8`, path names are encoded in UTF-8 when creating tar files, and path names are assumed to be encoded in UTF-8 when extracting tar files.

If *`file:native_name_encoding/0`* returns `latin1`, no translation of path names is done.

Unicode metadata stored in PAX headers is preserved

Other Storage Media

The *`ftp`* module (Inets) normally accesses the tar file on disk using the *`file`* module. When other needs arise, you can define your own low-level Erlang functions to perform the writing and reading on the storage media; use function *`init/3`*.

An example of this is the SFTP support in *`ssh_sftp:open_tar/3`*. This function opens a tar file on a remote machine using an SFTP channel.

Limitations

- If you must remain compatible with the USTAR tar format, you must ensure file paths being stored are less than 255 bytes in total, with a maximum filename component length of 100 bytes. USTAR uses a header field (prefix) in addition to the name field, and splits file paths longer than 100 bytes into two parts. This split is done on a directory boundary, and is done in such a way to make the best use of the space available in those two fields, but in practice this will often mean that you have less than 255 bytes for a path. `erl_tar` will automatically upgrade the format to PAX to handle longer filenames, so this is only an issue if you need to extract the archive with an older implementation of `erl_tar` or `tar` which does not support PAX. In this case, the PAX headers will be extracted as regular files, and you will need to apply them manually.
- Like the above, if you must remain USTAR compatible, you must also ensure than paths for symbolic/hard links are no more than 100 bytes, otherwise PAX headers will be used.

Exports

add(TarDescriptor, Filename, Options) -> RetValue

Types:

```
TarDescriptor = term()
FilenameOrBin = filename()|binary()
NameInArchive = filename()
Filename = filename()|{NameInArchive,FilenameOrBin}
Options = [Option]
Option = dereference|verbose|{chunks,ChunkSize}
ChunkSize = positive_integer()
RetValue = ok|{error,{Filename,Reason}}
Reason = term()
```

Adds a file to a tar file that has been opened for writing by *open/1*.

`NameInArchive` is the name under which the file becomes stored in the tar file. The file gets this name when it is extracted from the tar file.

Options:

`dereference`

By default, symbolic links are stored as symbolic links in the tar file. To override the default and store the file that the symbolic link points to into the tar file, use option `dereference`.

`verbose`

Prints an informational message about the added file.

`{chunks,ChunkSize}`

Reads 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, see *ssh_sftp:open_tar/3*.

add(TarDescriptor, FilenameOrBin, NameInArchive, Options) -> RetValue

Types:

```
TarDescriptor = term()
FilenameOrBin = filename()|binary()
Filename = filename()
NameInArchive = filename()
Options = [Option]
Option = dereference|verbose
RetValue = ok|{error,{Filename,Reason}}
Reason = term()
```

Adds a file to a tar file that has been opened for writing by *open/2*. This function accepts the same options as *add/3*.

close(TarDescriptor)

Types:

```
TarDescriptor = term()
```

Closes a tar file opened by *open/2*.

create(Name, FileList) ->RetValue

Types:

```
Name = filename()
FileList = [Filename|{NameInArchive, FilenameOrBin}]
FilenameOrBin = filename()|binary()
Filename = filename()
NameInArchive = filename()
RetValue = ok|{error,{Name,Reason}}
Reason = term()
```

Creates a tar file and archives the files whose names are specified in `FileList` into it. The files can either be read from disk or be specified as binaries.

create(Name, FileList, OptionList)

Types:

```
Name = filename()
FileList = [Filename|{NameInArchive, FilenameOrBin}]
FilenameOrBin = filename()|binary()
Filename = filename()
NameInArchive = filename()
OptionList = [Option]
Option = compressed|cooked|dereference|verbose
RetValue = ok|{error,{Name,Reason}}
Reason = term()
```

Creates a tar file and archives the files whose names are specified in `FileList` into it. The files can either be read from disk or be specified as binaries.

The options in `OptionList` modify the defaults as follows:

`compressed`

The entire tar file is compressed, as if it has been run through the `gzip` program. To abide to the convention that a compressed tar file is to end in "`.tar.gz`" or "`.tgz`", add the appropriate extension.

`cooked`

By default, function `open/2` opens 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 overrides the default and opens the tar file without option `raw`.

`dereference`

By default, symbolic links are stored as symbolic links in the tar file. To override the default and store the file that the symbolic link points to into the tar file, use option `dereference`.

`verbose`

Prints an informational message about each added file.

extract(Name) -> RetValue

Types:

```
Name = filename() | {binary,binary()} | {file,Fd}
```

```
Fd = file_descriptor()
RetValue = ok|{error,{Name,Reason}}
Reason = term()
```

Extracts all files from a tar archive.

If argument `Name` is specified as `{binary,Binary}`, the contents of the binary is assumed to be a tar archive.

If argument `Name` is specified as `{file,Fd}`, `Fd` is assumed to be a file descriptor returned from function `file:open/2`.

Otherwise, `Name` is to be a filename.

> **Note:**
>
> Leading slashes in tar member names will be removed before writing the file. That is, absolute paths will be turned into relative paths. There will be an info message written to the error logger when paths are changed in this way.

`extract(Name, OptionList)`

Types:

```
Name = filename() | {binary,binary()} | {file,Fd}
Fd = file_descriptor()
OptionList = [Option]
Option = {cwd,Cwd}|{files,FileList}|keep_old_files|verbose|memory
Cwd = [dirname()]
FileList = [filename()]
RetValue = ok|MemoryRetValue|{error,{Name,Reason}}
MemoryRetValue = {ok, [{NameInArchive,binary()}]}
NameInArchive = filename()
Reason = term()
```

Extracts files from a tar archive.

If argument `Name` is specified as `{binary,Binary}`, the contents of the binary is assumed to be a tar archive.

If argument `Name` is specified as `{file,Fd}`, `Fd` is assumed to be a file descriptor returned from function `file:open/2`.

Otherwise, `Name` is to be a filename.

The following options modify the defaults for the extraction as follows:

`{cwd,Cwd}`

Files with relative filenames are by default extracted to the current working directory. With this option, files are instead extracted into directory `Cwd`.

`{files,FileList}`

By default, all files are extracted from the tar file. With this option, only those files are extracted whose names are included in `FileList`.

`compressed`

With this option, the file is uncompressed while extracting. If the tar file is not compressed, this option is ignored.

`cooked`

By default, function `open/2` function opens 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 overrides the default and opens the tar file without option `raw`.

`memory`

Instead of extracting to a directory, this option gives 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 files in the tar file are overwritten. With this option, existing files are not overwriten.

`verbose`

Prints an informational message for each extracted file.

`format_error(Reason) -> string()`

Types:

`Reason = term()`

Cconverts an error reason term to a human-readable error message string.

`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)->... end`

`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 (such as `add/3`, `add/4`, and `close/1`).

The `Fun` is 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.

Parameter `UserPrivate` is typically the result of opening a low-level structure like a file descriptor or an SFTP channel id. The different `Fun` clauses operate on that very term.

The following are the fun clauses parameter lists:

`(write, {UserPrivate,DataToWrite})`

Writes term `DataToWrite` using `UserPrivate`.

`(close, UserPrivate)`

Closes the access.

`(read2, {UserPrivate,Size})`

Reads using `UserPrivate` but only `Size` bytes. Notice that there is only an arity-2 read function, not an arity-1 function.

`(position,{UserPrivate,Position})`

Sets the position of `UserPrivate` as defined for files in *`file:position/2`*

Example:

The following is a complete `Fun` parameter for reading and writing on files using the *`file`* module:

```
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
```

Here `Fd` was specified to function `init/3` as:

```
{ok,Fd} = file:open(Name, ...).
{ok,TarDesc} = erl_tar:init(Fd, [write], ExampleFun),
```

`TarDesc` is then used:

```
erl_tar:add(TarDesc, SomeValueIwantToAdd, FileNameInTarFile),
...,
erl_tar:close(TarDesc)
```

When the `erl_tar` core wants to, for example, write a piece of `Data`, it would call `ExampleFun(write, {UserPrivate,Data})`.

Note:

This example with the `file` module operations is not necessary to use directly, as that is what function *`open/2`* in principle does.

Warning:

The `TarDescriptor` term is not a file descriptor. You are advised not to rely on the specific contents of this term, as it can change in future Erlang/OTP releases when more features are added to this module.

`open(Name, OpenModeList) -> RetValue`

Types:

`Name = filename()`
`OpenModeList = [OpenMode]`
`Mode = write|compressed|cooked`
`RetValue = {ok,TarDescriptor}|{error,{Name,Reason}}`
`TarDescriptor = term()`
`Reason = term()`

Creates a tar file for writing (any existing file with the same name is truncated).

By convention, the name of a tar file is to end in "`.tar`". To abide to the convention, add "`.tar`" to the name.

Except for the `write` atom, the following atoms can be added to `OpenModeList`:

`compressed`

The entire tar file is compressed, as if it has been run through the `gzip` program. To abide to the convention that a compressed tar file is to end in "`.tar.gz`" or "`.tgz`", add the appropriate extension.

`cooked`

By default, the tar file is opened in `raw` mode, which is faster but does not allow a remote (Erlang) file server to be used. Adding `cooked` to the mode list overrides the default and opens the tar file without option `raw`.

To add one file at the time into an opened tar file, use function *`add/3,4`*. When you are finished adding files, use function *`close/1`* to close the tar file.

Warning:

The `TarDescriptor` term is not a file descriptor. You are advised not to rely on the specific contents of this term, as it can change in future Erlang/OTP releases when more features are added to this module..

`table(Name) -> RetValue`

Types:

```
Name = filename()|{binary,binary()}|{file,file_descriptor()}
RetValue = {ok,[string()]}|{error,{Name,Reason}}
Reason = term()
```

Retrieves the names of all files in the tar file `Name`.

`table(Name, Options)`

Types:

```
Name = filename()|{binary,binary()}|{file,file_descriptor()}
```

Retrieves the names of all files in the tar file `Name`.

`t(Name)`

Types:

```
Name = filename()|{binary,binary()}|{file,file_descriptor()}
```

Prints the names of all files in the tar file `Name` to the Erlang shell (similar to "`tar t`").

`tt(Name)`

Types:

```
Name = filename()|{binary,binary()}|{file,file_descriptor()}
```

Prints names and information about all files in the tar file `Name` to the Erlang shell (similar to "`tar tv`").

ets

Erlang module

This module is an interface to the Erlang built-in term storage BIFs. These provide the ability to store very large quantities of data in an Erlang runtime system, and to have constant access time to the data. (In the case of `ordered_set`, see below, access time is proportional to the logarithm of the number of stored objects.)

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` table can have many objects associated with each key.

Note:

The number of tables stored at one Erlang node **used** to be limited. This is no longer the case (except by memory usage). The previous default limit was about 1400 tables and could be increased by setting the environment variable `ERL_MAX_ETS_TABLES` before starting the Erlang runtime system. This hard limit has been removed, but it is currently useful to set the `ERL_MAX_ETS_TABLES` anyway. It should be set to an approximate of the maximum amount of tables used. This since an internal table for named tables is sized using this value. If large amounts of named tables are used and `ERL_MAX_ETS_TABLES` hasn't been increased, the performance of named table lookup will degrade.

Notice that there is no automatic garbage collection for tables. Even if there are no references to a table from any process, it is not automatically destroyed unless the owner process terminates. To destroy a table explicitly, use function *`delete/1`*. The default owner is the process that created the table. To transfer table ownership at process termination, use option *`heir`* or call *`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'` is not to be used as a key, as this atom is used to mark the end of the table when using functions *`first/1`* and *`next/2`*.

Notice the subtle difference between **matching** and **comparing equal**, which is demonstrated by 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 no defined order exists 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

The functions in this module exits with reason `badarg` if any argument has the wrong format, if the table identifier is invalid, or if the operation is denied because of 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 to a single object either succeeds or fails completely without any effect (atomicity) and that no intermediate results of the update can be seen by other processes (isolation). Some functions that update many 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 are carried out serially, one after the other in a strict order.

No other support is available within this module that would guarantee consistency between objects. However, function *`safe_fixtable/2`* can be used to guarantee that a sequence of *`first/1`* and *`next/2`* calls traverse the table without errors and that each existing object in the table is visited exactly once, even if another (or the same) process simultaneously deletes or inserts objects into the table. Nothing else is guaranteed; in particular objects that are inserted or deleted during such a traversal can be visited once or not at all. Functions that internally traverse over a table, like *`select`* and *`match`*, give the same guarantee as *`safe_fixtable`*.

Match Specifications

Some of the functions use a **match specification**, `match_spec`. For a brief explanation, see *`select/2`*. For a detailed description, see section *Match Specifications in Erlang* in ERTS User's Guide.

Data Types

`access() = public | protected | private`

`continuation()`

Opaque continuation used by *`select/1,3`*, *`select_reverse/1,3`*, *`match/1,3`*, and *`match_object/1,3`*.

`match_spec() = [{`*`match_pattern()`*`, [term()], [term()]}]`

A match specification, see above.

`comp_match_spec()`

A compiled match specification.

`match_pattern() = atom() | tuple()`

`tab() = atom() |` ***`tid()`***

`tid()`

A table identifier, as returned by *`new/2`*.

`type() = set | ordered_set | bag | duplicate_bag`

Exports

`all() -> [Tab]`

Types:

`Tab =` ***`tab()`***

Returns a list of all tables at the node. Named tables are specified by their names, unnamed tables are specified by their table identifiers.

There is no guarantee of consistency in the returned list. Tables created or deleted by other processes "during" the `ets:all()` call either are or are not included in the list. Only tables created/deleted **before** `ets:all()` is called are guaranteed to be included/excluded.

delete(Tab) -> true

Types:

Tab = *tab()*

Deletes the entire table `Tab`.

delete(Tab, Key) -> true

Types:

Tab = *tab()*

Key = term()

Deletes all objects with key `Key` from table `Tab`.

delete_all_objects(Tab) -> true

Types:

Tab = *tab()*

Delete all objects in the ETS table `Tab`. The operation is guaranteed to be *atomic and isolated*.

delete_object(Tab, Object) -> true

Types:

Tab = *tab()*

Object = tuple()

Delete the exact object `Object` from the ETS table, leaving objects with the same key but other differences (useful for type `bag`). In a `duplicate_bag` table, all instances of the object are deleted.

file2tab(Filename) -> {ok, Tab} | {error, Reason}

Types:

Filename = *file:name()*

Tab = *tab()*

Reason = term()

Reads a file produced by *`tab2file/2`* or *`tab2file/3`* and creates the corresponding table `Tab`.

Equivalent to `file2tab(Filename, [])`.

file2tab(Filename, Options) -> {ok, Tab} | {error, Reason}

Types:

Filename = *file:name()*

Tab = *tab()*

Options = [Option]

Option = {verify, boolean()}

Reason = term()

Reads a file produced by *`tab2file/2`* or *`tab2file/3`* and creates the corresponding table `Tab`.

The only supported option is `{verify,boolean()}`. If verification is turned on (by specifying `{verify,true}`), the function uses whatever information is present in the file to assert that the information is not damaged. How this is done depends on which `extended_info` was written using *`tab2file/3`*.

If no `extended_info` is present in the file and `{verify,true}` is specified, the number of objects written is compared to the size of the original table when the dump was started. This can make verification fail if the table was `public` and objects were added or removed while the table was dumped to file. To avoid this problem, either do not verify files dumped while updated simultaneously or use option `{extended_info, [object_count]}` to *`tab2file/3`*, which extends the information in the file with the number of objects written.

If verification is turned on and the file was written with option `{extended_info, [md5sum]}`, reading the file is slower and consumes radically more CPU time than otherwise.

`{verify,false}` is the default.

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

Types:

`Tab = tab()`
`Key = term()`

Returns the first key `Key` in table `Tab`. For an `ordered_set` table, the first key in Erlang term order is returned. For other table types, the first key according to the internal order of the table is returned. If the table is empty, `'$end_of_table'` is returned.

To find subsequent keys in the table, use *`next/2`*.

`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 table elements are traversed in an unspecified order, except for `ordered_set` tables, where they are traversed first to last.

If `Function` inserts objects into the table, or another process inserts objects into the table, those objects **can** (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 table elements are traversed in an unspecified order, except for `ordered_set` tables, where they are traversed last to first.

If `Function` inserts objects into the table, or another process inserts objects into the table, those objects **can** (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 `DetsTab`. Existing objects in the ETS table are kept unless overwritten.

If any of the tables does not exist or the Dets table is not open, a `badarg` exception is raised.

fun2ms(LiteralFun) -> MatchSpec

Types:

LiteralFun = function()
MatchSpec = *match_spec()*

Pseudo function that by a `parse_transform` translates `LiteralFun` typed as parameter in the function call to a *match specification*. With "literal" is meant that the fun must textually be written as the parameter of the function, it cannot be held in a variable that in turn is passed to the function.

The parse transform is provided in the `ms_transform` module and the source **must** include file `ms_transform.hrl` in STDLIB for this pseudo function to work. Failing to include the hrl file in the source results in a runtime error, not a compile time error. The include file is easiest included by adding 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 must use the `is_` guard tests. Language constructs that have no representation in a match specification (`if`, `case`, `receive`, and so on) are not allowed.

The return value is the resulting match specification.

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 the following works:

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

The imported variables are replaced by match specification `const` expressions, which is consistent with the static scoping for Erlang funs. However, local or global function calls cannot be in the guard or body of the fun. Calls to built-in match specification functions is of course allowed:

```
4> ets:fun2ms(fun({M,N}) when N > X, my_fun(M) -> M end).
Error: fun containing local Erlang function calls
('my_fun' 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 shown by the example, the function can be called from the shell also. The fun must be literally in the call when used from the shell as well.

> **Warning:**
>
> If the `parse_transform` is not applied to a module that calls this pseudo function, the call fails in runtime (with a `badarg`). The `ets` module exports a function with this name, but it is never to be called except when using the function in the shell. If the `parse_transform` is properly applied by including header file `ms_transform.hrl`, compiled code never calls the function, but the function call is replaced by a literal match specification.

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, message `{'ETS-TRANSFER',Tab,FromPid,GiftData}` is 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.

Notice that this function does not affect option *`heir`* of the table. A table owner can, for example, set `heir` to itself, give the table away, and then get it back if the receiver terminates.

`i() -> ok`

Displays information about all ETS tables on a terminal.

`i(Tab) -> ok`

Types:

```
Tab = tab()
```

Browses table `Tab` on a terminal.

`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} |
    {type, type()} |
```

```
        {write_concurrency, boolean()} |
        {read_concurrency, boolean()}
```

Returns information about 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, a `badarg` exception is raised.

`{compressed, boolean()}`

Indicates if the table is compressed.

`{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 table name.

`{named_table, boolean()}`

Indicates if the table is named.

`{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.

`{read_concurrency, boolean()}`

Indicates whether the table uses `read_concurrency` or not.

`{write_concurrency, boolean()}`

Indicates whether the table uses `write_concurrency`.

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

Types:

```
   Tab = tab()
   Item =
       compressed |
       fixed |
       heir |
       keypos |
```

```
      memory |
      name |
      named_table |
      node |
      owner |
      protection |
      safe_fixed |
      safe_fixed_monotonic_time |
      size |
      stats |
      type |
      write_concurrency |
      read_concurrency
  Value = term()
```

Returns the information associated with `Item` for 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, a `badarg` exception is raised.

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.

- `Item=safe_fixed|safe_fixed_monotonic_time, Value={FixationTime,Info}|false`

  If the table has been fixed using *`safe_fixtable/2`*, the call returns a tuple where `FixationTime` is the time when the table was first fixed by a process, which either is or is not one of the processes it is fixed by now.

  The format and value of `FixationTime` depends on `Item`:

  `safe_fixed`

  `FixationTime` corresponds to the result returned by *`erlang:timestamp/0`* at the time of fixation. Notice that when the system uses single or multi *time warp modes* this can produce strange results, as the use of `safe_fixed` is not *time warp safe*. Time warp safe code must use `safe_fixed_monotonic_time` instead.

  `safe_fixed_monotonic_time`

  `FixationTime` corresponds to the result returned by *`erlang:monotonic_time/0`* at the time of fixation. The use of `safe_fixed_monotonic_time` is *time warp safe*.

  `Info` is a possibly empty lists of tuples `{Pid,RefCount}`, one tuple for every process the table is fixed by now. `RefCount` is the value of the reference counter and it keeps 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.

`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 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 *`insert/2`*.

When called with 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 is called exactly once, and if an error occur, the last function is called with argument `close`, the reply of which is ignored.

If the table type is `set` and more than one object exists 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`.

insert(Tab, ObjectOrObjects) -> true

Types:

```
Tab = tab()
ObjectOrObjects = tuple() | [tuple()]
```

Inserts the object or all of the objects in list `ObjectOrObjects` into table `Tab`.

- If the table type is `set` and the key of the inserted objects **matches** the key of any object in the table, the old object is replaced.
- If the table type is `ordered_set` and the key of the inserted object **compares equal** to the key of any object in the table, the old object is replaced.
- If the list contains more than one object with **matching** keys and the table type is `set`, one is inserted, which one is not defined. The same holds for table type `ordered_set` 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()]
```

Same as *`insert/2`* except that instead of overwriting objects with the same key (for `set` or `ordered_set`) or adding more objects with keys already existing in the table (for `bag` and `duplicate_bag`), `false` is returned.

If `ObjectOrObjects` is a list, the function checks **every** key before inserting anything. Nothing is inserted unless **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()
```

Checks if a term is a valid compiled *match specification*. The compiled match specification is an opaque datatype that **cannot** be sent between Erlang nodes or be stored on disk. Any attempt to create an external representation of a compiled match specification results in an empty binary (<<>>).

Examples:

The following expression yields `true`::

```
ets:is_compiled_ms(ets:match_spec_compile([{'_',[],[true]}])).
```

The following expressions yield `false`, as variable `Broken` contains a compiled match specification that has passed through external representation:

```
MS = ets:match_spec_compile([{'_',[],[true]}]),
Broken = binary_to_term(term_to_binary(MS)),
ets:is_compiled_ms(Broken).
```

Note:

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

`last(Tab) -> Key | '$end_of_table'`

Types:

`Tab = tab()`
`Key = term()`

Returns the last key `Key` according to Erlang term order in table `Tab` of type `ordered_set`. For other table types, the function is synonymous to *`first/1`*. If the table is empty, `'$end_of_table'` is returned.

To find preceding keys in the table, use *`prev/2`*.

`lookup(Tab, Key) -> [Object]`

Types:

`Tab = tab()`
`Key = term()`
`Object = tuple()`

Returns a list of all objects with key `Key` in table `Tab`.

- For tables of type `set`, `bag`, or `duplicate_bag`, an object is returned only if the specified key **matches** the key of the object in the table.
- For tables of type `ordered_set`, an object is returned if the specified key **compares equal** to the key of an object in the table.

The difference is the same as between `=:=` and `==`.

As an example, one can insert an object with `integer()` `1` as a key in an `ordered_set` and get the object returned as a result of doing a `lookup/2` with `float()` `1.0` as the key to search for.

For tables of type `set` or `ordered_set`, the function returns either the empty list or a list with one element, as there cannot be more than one object with the same key. For tables of type `bag` or `duplicate_bag`, the function returns a list of arbitrary length.

Notice that the time order of object insertions is preserved; the first object inserted with the specified key is the first in the resulting list, and so on.

Insert and lookup times in tables of type `set`, `bag`, and `duplicate_bag` are constant, regardless of the table size. For the `ordered_set` datatype, time is proportional to the (binary) logarithm of the number of objects.

lookup_element(Tab, Key, Pos) -> Elem

Types:

Tab = *tab()*
Key = term()
Pos = integer() >= 1
Elem = term() | [term()]

For a table `Tab` of type `set` or `ordered_set`, the function returns the `Pos`:th element of the object with key `Key`.

For tables of type `bag` or `duplicate_bag`, the functions returns a list with the `Pos`:th element of every object with key `Key`.

If no object with key `Key` exists, the function exits with reason `badarg`.

The difference between `set`, `bag`, and `duplicate_bag` on one hand, and `ordered_set` on the other, regarding the fact that `ordered_set` view keys as equal when they **compare equal** whereas the other table types regard them equal only when they **match**, holds for `lookup_element/3`.

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

Types:

Match = [term()]
Continuation = *continuation()*

Continues a match started with *`match/3`*. The next chunk of the size specified in the initial `match/3` call is returned together with a new `Continuation`, which can be used in subsequent calls to this function.

When there are no more objects in the table, `'$end_of_table'` is returned.

match(Tab, Pattern) -> [Match]

Types:

Tab = *tab()*
Pattern = *match_pattern()*
Match = [term()]

Matches the objects in table `Tab` against pattern `Pattern`.

A pattern is a term that can contain:

- Bound parts (Erlang terms)
- `'_'` that matches any Erlang term
- 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, for example:

```
6> ets:match(T, '$1'). % Matches every object in 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, that is, 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.

For tables of type `ordered_set`, 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 *match/2*, but returns only a limited (`Limit`) number of matching objects. Term `Continuation` can then be used in subsequent calls to *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 faster than traversing the table object by object using *first/1* and *next/2*.

If the table is empty, `'$end_of_table'` is returned.

```
match_delete(Tab, Pattern) -> true
```

Types:

```
Tab = tab()
Pattern = match_pattern()
```

Deletes all objects that match pattern `Pattern` from table `Tab`. For a description of patterns, see *match/2*.

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

Types:

```
Object = tuple()
Continuation = continuation()
```

Continues a match started with *match_object/3*. The next chunk of the size specified in the initial `match_object/3` call is returned together with a new `Continuation`, which can be used in subsequent calls to this function.

When there are no more objects in the table, `'$end_of_table'` is returned.

```
match_object(Tab, Pattern) -> [Object]
```

Types:

```
Tab = tab()
Pattern = match_pattern()
Object = tuple()
```

Matches the objects in table `Tab` against pattern `Pattern`. For a description of patterns, see *match/2*. The function returns a list of all objects that match the pattern.

If the key is specified in the pattern, the match is very efficient. If the key is not specified, that is, 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.

For tables of type `ordered_set`, the result is in the same order as in a `first`/`next` traversal.

```
match_object(Tab, Pattern, Limit) ->
                {[Object], Continuation} | '$end_of_table'
```

Types:

```
Tab = tab()
Pattern = match_pattern()
Limit = integer() >= 1
Object = tuple()
Continuation = continuation()
```

Works like *match_object/2*, but only returns a limited (`Limit`) number of matching objects. Term `Continuation` can then be used in subsequent calls to *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 faster than traversing the table object by object using *first/1* and *next/2*.

If the table is empty, `'$end_of_table'` is returned.

match_spec_compile(MatchSpec) -> CompiledMatchSpec

Types:

```
MatchSpec = match_spec()
CompiledMatchSpec = comp_match_spec()
```

Transforms a *match specification* into an internal representation that can be used in subsequent calls to *match_spec_run/2*. The internal representation is opaque and cannot be converted to external term format and then back again without losing its properties (that is, it cannot be sent to a process on another node and still remain a valid compiled match specification, nor can it be stored on disk). To check the validity of a compiled match specification, use *is_compiled_ms/1*.

If term `MatchSpec` cannot be compiled (does not represent a valid match specification), a `badarg` exception is raised.

> **Note:**
>
> This function has limited use in normal code. It is used by the *dets* module to perform the `dets:select()` operations.

match_spec_run(List, CompiledMatchSpec) -> list()

Types:

```
List = [term()]
CompiledMatchSpec = comp_match_spec()
```

Executes the matching specified in a compiled *match specification* on a list of terms. Term `CompiledMatchSpec` is to be the result of a call to *match_spec_compile/1* and is hence the internal representation of the match specification one wants to use.

The matching is executed on each element in `List` and the function returns a list containing all results. If an element in `List` does not match, nothing is returned for that element. The length of the result list is therefore equal or less than the length of parameter `List`.

Example:

The following two calls give the same result (but certainly not the same execution time):

```
Table = ets:new...
MatchSpec = ...
% The following call...
ets:match_spec_run(ets:tab2list(Table),
                   ets:match_spec_compile(MatchSpec)),
% ...gives the same result as the more common (and more efficient)
ets:select(Table, MatchSpec),
```

Note:

This function has limited use in normal code. It is used by the *dets* module to perform the `dets:select()` operations and by Mnesia during transactions.

member(Tab, Key) -> boolean()

Types:

Tab = *tab()*

Key = term()

Works like *lookup/2*, but does not return the objects. Returns `true` if one or more elements in the table has key `Key`, otherwise `false`.

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 that 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.

Parameter `Options` is a list of atoms that specifies table type, access rights, key position, and whether the table is named. Default values are used for omitted options. 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 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` table, `integer() 1` and `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 returned elements, 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 can 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, 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 to use as key. By default, it is the first element, that is, `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.

Notice 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 inherits the table if the owner terminates. Message `{'ETS-TRANSFER',tid(),FromPid,HeirData}` is sent to the heir when that occurs. The heir must be a local process. Default heir is `none`, which destroys the table when the owner terminates.

`{write_concurrency,boolean()}`

Performance tuning. Defaults to `false`, in which case an operation that mutates (writes to) the table obtains exclusive access, blocking any concurrent access of the same table until finished. If set to `true`, the table is optimized to 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.

Option `write_concurrency` can be combined with option *`read_concurrency`*. You typically want to combine these when large concurrent read bursts and large concurrent write bursts are common; for more information, see option *`read_concurrency`*.

Notice that this option does not change any guarantees about *atomicity and isolation*. Functions that makes such promises over many objects (like *`insert/2`*) gain less (or nothing) from this option.

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. Defaults to `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 (that is, many reads not interrupted by writes, and many 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 would get a performance degradation by enabling this option.

Option `read_concurrency` can be combined with option *`write_concurrency`*. 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 is stored in a more compact format to consume less memory. However, it will make table operations slower. Especially operations that need to inspect entire objects, such as `match` and `select`, get much slower. The key element is not compressed.

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

Types:

`Tab = tab()`
`Key1 = Key2 = term()`

Returns the next key `Key2`, following key `Key1` in table `Tab`. For table type `ordered_set`, the next key in Erlang term order is returned. For other table types, the next key according to the internal order of the table is returned. If no next key exists, `'$end_of_table'` is returned.

To find the first key in the table, use *`first/1`*.

Unless a table of type `set`, `bag`, or `duplicate_bag` is protected using *`safe_fixtable/2`*, a traversal can fail if concurrent updates are made to the table. For table type `ordered_set`, the function returns the next key in order, even if the object does no longer exist.

`prev(Tab, Key1) -> Key2 | '$end_of_table'`

Types:

`Tab = tab()`
`Key1 = Key2 = term()`

Returns the previous key `Key2`, preceding key `Key1` according to Erlang term order in table `Tab` of type `ordered_set`. For other table types, the function is synonymous to *`next/2`*. If no previous key exists, `'$end_of_table'` is returned.

To find the last key in the table, use *`last/1`*.

rename(Tab, Name) -> Name

Types:

Tab = *tab()*

Name = atom()

Renames the named table `Tab` to the new name `Name`. Afterwards, the old name cannot be used to access the table. Renaming an unnamed table has no effect.

repair_continuation(Continuation, MatchSpec) -> Continuation

Types:

Continuation = *continuation()*

MatchSpec = *match_spec()*

Restores 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 are 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.

Examples:

The following sequence of calls fails:

```
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).
```

The following sequence works, as the call to `repair_continuation/2` reestablishes the (deliberately) invalidated continuation `Broken`.

```
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)).
```

Note:

This function is rarely needed in application code. It is used by Mnesia to provide 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 specification is performance. It can be subject to change in future releases, while this interface remains for backward compatibility.

safe_fixtable(Tab, Fix) -> true

Types:

Tab = *tab()*

Fix = boolean()

Fixes a table of type `set`, `bag`, or `duplicate_bag` 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 many processes fix a table, the table remains 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 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 is returned only once, even if objects are removed or inserted during the traversal. The keys for new objects inserted during the traversal **can** be returned by `next/2` (it depends on the internal ordering of the keys).

Example:

```
clean_all_with_value(Tab,X) ->
    safe_fixtable(Tab,true),
    clean_all_with_value(Tab,X,ets:first(Tab)),
    safe_fixtable(Tab,false).

clean_all_with_value(Tab,X,'$end_of_table') ->
    true;
clean_all_with_value(Tab,X,Key) ->
    case ets:lookup(Tab,Key) of
        [{Key,X}] ->
            ets:delete(Tab,Key);
        _ ->
            true
    end,
    clean_all_with_value(Tab,X,ets:next(Tab,Key)).
```

Notice that no deleted objects are 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 is never freed. The performance of operations on the table also degrades significantly.

To retrieve information about which processes have fixed which tables, use *`info(Tab, safe_fixed_monotonic_time)`*. A system with many processes fixing tables can need a monitor that sends alarms when tables have been fixed for too long.

Notice that for table type `ordered_set`, `safe_fixtable/2` is not necessary, as calls to `first/1` and `next/2` always succeed.

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

Types:

Match = term()

Continuation = *continuation()*

Continues a match started with *`select/3`*. The next chunk of the size specified in the initial `select/3` call is returned together with a new `Continuation`, which can be used in subsequent calls to this function.

When there are no more objects in the table, `'$end_of_table'` is returned.

select(Tab, MatchSpec) -> [Match]

Types:

Tab = *tab()*
MatchSpec = *match_spec()*
Match = term()

Matches the objects in table `Tab` using a *match specification*. This is a more general call than *`match/2`* and *`match_object/2`* calls. In its simplest form, the match specification is as follows:

```
MatchSpec = [MatchFunction]
MatchFunction = {MatchHead, [Guard], [Result]}
MatchHead = "Pattern as in ets:match"
Guard = {"Guardtest name", ...}
Result = "Term construct"
```

This means that the match specification is always a list of one or more tuples (of arity 3). The first element of the tuple is to be a pattern as described in *`match/2`*. The second element of the tuple is to be a list of 0 or more guard tests (described below). The third element of the tuple is to be a list containing a description of the value to return. In almost all normal cases, the list contains exactly one term that fully describes the value to return for each object.

The return value is constructed using the "match variables" bound in `MatchHead` or using the special match variables `'$_'` (the whole matching object) and `'$$'` (all match variables in a list), so that the following `match/2` expression:

```
ets:match(Tab,{'$1','$2','$3'})
```

is exactly equivalent to:

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

And that the following `match_object/2` call:

```
ets:match_object(Tab,{'$1','$2','$1'})
```

is exactly equivalent to

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

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

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

gives the same output as:

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

That is, 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 where the single element in the tuple is the tuple one wants to construct (as an ordinary tuple can 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 the *`dbg(3)`*) module in Runtime_Tools.

The `Guard`s are constructed as tuples, where the first element is the test name and the remaining elements are the test parameters. 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 does not match and the next `MatchFunction`

(if any) is tried. Most guard tests present in Erlang can be used, but only the new versions prefixed `is_` are allowed (`is_float`, `is_atom`, and so on).

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 the following guard test written in Erlang:

```
is_integer(X), is_integer(Y), X + Y < 4711
```

is expressed as follows (`X` replaced with `'$1'` and `Y` with `'$2'`):

```
[{is_integer, '$1'}, {is_integer, '$2'}, {'<', {'+', '$1', '$2'}, 4711}]
```

For tables of type `ordered_set`, objects are visited in the same order as in a `first`/`next` traversal. This means that the match specification is executed against objects with keys in the `first`/`next` order and the corresponding result list is 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 *`select/2`*, but only returns a limited (`Limit`) number of matching objects. Term `Continuation` can then be used in subsequent calls to *`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 *`first/1`* and *`next/2`*.

If the table is empty, `'$end_of_table'` is returned.

`select_count(Tab, MatchSpec) -> NumMatched`

Types:

`Tab = tab()`
`MatchSpec = match_spec()`
`NumMatched = integer() >= 0`

Matches the objects in table `Tab` using a *match specification*. If the match specification returns `true` for an object, that object considered a match and is counted. For any other result from the match specification the object is not considered a match and is therefore not counted.

This function can be described as a *`match_delete/2`* function that does not 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 table `Tab` using a *match specification*. If the match specification returns `true` for an object, that object is removed from the table. For any other result from the match specification the object is retained. This is a more general call than the *`match_delete/2`* call.

The function returns the number of objects deleted from the table.

> **Note:**
>
> The match specification has to return the atom `true` if the object is to be deleted. No other return value gets the object deleted. So one cannot use the same match specification for looking up elements as for deleting them.

select_replace(Tab, MatchSpec) -> NumReplaced

Types:

Tab = *tab()*
MatchSpec = *match_spec()*
NumReplaced = integer() >= 0

Matches the objects in the table `Tab` using a *match specification*. For each matched object, the existing object is replaced with the match specification result.

The match-and-replace operation for each individual object is guaranteed to be *atomic and isolated*. The `select_replace` table iteration as a whole, like all other select functions, does not give such guarantees.

The match specifiction must be guaranteed to **retain the key** of any matched object. If not, `select_replace` will fail with `badarg` without updating any objects.

For the moment, due to performance and semantic constraints, tables of type `bag` are not yet supported.

The function returns the total number of replaced objects.

Example

For all 2-tuples with a list in second position, add atom `'marker'` first in the list:

```
1> T = ets:new(x,[]), ets:insert(T, {key, [1, 2, 3]}).
true
2> MS = ets:fun2ms(fun({K, L}) when is_list(L) -> {K, [marker | L]} end).
[{{'$1','$2'},[{is_list,'$2'}],[{{'$1',[marker|'$2']}}]}]
3> ets:select_replace(T, MS).
1
4> ets:tab2list(T).
[{key,[marker,1,2,3]}]
```

A generic single object compare-and-swap operation:

```
[Old] = ets:lookup(T, Key),
New = update_object(Old),
Success = (1 =:= ets:select_replace(T, [{Old, [], [{const, New}]}])),
```

select_reverse(Continuation) ->

{[Match], Continuation} | '$end_of_table'

Types:

Continuation = *continuation()*

Match = term()

Continues a match started with *select_reverse/3*. For tables of type `ordered_set`, the traversal of the table continues to objects with keys earlier in the Erlang term order. The returned list also contains objects with keys in reverse order. For all other table types, the behavior is exactly that of *select/1*.

Example:

```
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).
...
```

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 table type `ordered_set`. 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 table type `ordered_set` traversing is done starting at the last object in Erlang term order and moves to the first. For all other table types, the return value is identical to that of `select/3`.

Notice 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.

setopts(Tab, Opts) -> true

Types:

Tab = *tab()*
Opts = Opt | [Opt]
Opt = {heir, pid(), HeirData} | {heir, none}
HeirData = term()

Sets table options. The only allowed option 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 first/next or last/prev are to be used instead.

Returns all objects in slot I of 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. If argument I is out of range, the function fails with reason badarg.

Unless a table of type set, bag, or duplicate_bag is protected using *safe_fixtable/2*, a traversal can fail if concurrent updates are made to the table. For table type ordered_set, the function returns a list containing object I in Erlang term order.

tab2file(Tab, Filename) -> ok | {error, Reason}

Types:

Tab = *tab()*
Filename = *file:name()*
Reason = term()

Dumps table Tab to 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]} | {sync, boolean()}
ExtInfo = md5sum | object_count
Reason = term()

Dumps table Tab to file Filename.

When dumping the table, some 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 is 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 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.

Option `extended_info` specifies what extra information is written to the table dump:

`object_count`

The number of objects written to the file is noted in the file footer, so file truncation can be verified 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 detects the slightest bitflip in the file data. Using this costs a fair amount of CPU time.

Whenever option `extended_info` is used, it results in a file not readable by versions of ETS before that in STDLIB 1.15.1

If option `sync` is set to `true`, it ensures that the content of the file is written to the disk before `tab2file` returns. Defaults to `{sync, false}`.

`tab2list(Tab) -> [Object]`

Types:

```
Tab = tab()
Object = tuple()
```

Returns a list of all objects in 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 when loading the table from file.

`type`

The ETS type of the dumped table (that is, `set`, `bag`, `duplicate_bag`, or `ordered_set`). This type is used when loading the table again.

`protection`

The protection of the dumped table (that is, `private`, `protected`, or `public`). A table loaded from the file gets the same protection.

`named_table`

`true` if the table was a named table when dumped to file, otherwise `false`. Notice 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 is used when loading the table again.

`size`

The number of objects in the table when the table dump to file started. For a `public` table, this number does not need to correspond to the number of objects saved to the file, as objects can 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`*. Notice 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 return `{0,0}` in this field.

An error is returned if the file is inaccessible, badly damaged, or not 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, NObjects} | {traverse, TraverseMethod}
NObjects = default | integer() >= 1
TraverseMethod =
    first_next |
    last_prev |
    select |
```

{select, MatchSpec :: *match_spec()*}

Returns a Query List Comprehension (QLC) query handle. The *qlc* module provides a query language aimed mainly at Mnesia, but ETS tables, Dets tables, and lists are also recognized by QLC as sources of data. Calling `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 *lookup/2* to look up the keys. When that is not possible, the whole table is traversed. Option `traverse` determines how this is done:

`first_next`

The table is traversed one key at a time by calling *first/1* and *next/2*.

`last_prev`

The table is traversed one key at a time by calling *last/1* and *prev/2*.

`select`

The table is traversed by calling *select/3* and *select/1*. 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 specification* (the second argument of `select/3`) is assembled by QLC: simple filters are translated into equivalent match specifications while more complicated filters must be applied to all objects returned by `select/3` given a match specification that matches all objects.

`{select, MatchSpec}`

As for `select`, the table is traversed by calling *select/3* and *select/1*. The difference is that the match specification is explicitly specified. This is how to state match specifications that cannot easily be expressed within the syntax provided by QLC.

Examples:

An explicit match specification is here used to traverse the table:

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

An example with an implicit match specification:

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

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

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

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

take(Tab, Key) -> [Object]

Types:

Tab = *tab()*
Key = term()
Object = tuple()

Returns and removes a list of all objects with key `Key` in table `Tab`.

The specified `Key` is used to identify the object by either **comparing equal** the key of an object in an `ordered_set` table, or **matching** in other types of tables (for details on the difference, see *`lookup/2`* and *`new/2`*).

`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 specification* used in calls to *`select/2`*. The function both tests `MatchSpec` for "syntactic" correctness and runs the match specification against object `Tuple`.

If the match specification is syntactically correct, the function either returns `{ok,Result}`, where `Result` is what would have been the result in a real `select/2` call, or `false` if the match specification does not match object `Tuple`.

If the match specification contains errors, tuple `{error, Errors}` is returned, where `Errors` is a list of natural language descriptions of what was wrong with the match specification.

This is a useful debugging and test tool, especially when writing complicated `select/2` calls.

See also: *erlang:match_spec_test/3*.

`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, Default) -> Result`
`update_counter(Tab, Key, X3 :: [UpdateOp]) -> [Result]`
`update_counter(Tab, Key, X3 :: [UpdateOp], Default) -> [Result]`
`update_counter(Tab, Key, Incr) -> Result`
`update_counter(Tab, Key, Incr, Default) -> Result`

Types:

`Tab = tab()`
`Key = term()`
`UpdateOp = {Pos, Incr} | {Pos, Incr, Threshold, SetValue}`
`Pos = Incr = Threshold = SetValue = Result = integer()`
`Default = tuple()`

This function provides an efficient way to update one or more counters, without the trouble 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, that is, no process can access the ETS table in the middle of the operation.)

This function destructively update the object with key `Key` in table `Tab` by adding `Incr` to the element at position `Pos`. The new counter value is returned. If no position is specified, the element directly following key (`<keypos> +1`) is updated.

If a `Threshold` is specified, the counter is reset to value `SetValue` if the following conditions occur:

- `Incr` is not negative (`>= 0`) and the result would be greater than (`>`) `Threshold`.
- `Incr` is negative (`< 0`) and the result would be less than (`<`) `Threshold`.

A list of `UpdateOp` can be supplied to do many 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 once in the list, the corresponding counter is thus updated many 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 fails, no updates is done.

The specified `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 (for details on the difference, see *`lookup/2`* and *`new/2`*).

If a default object `Default` is specified, it is used as the object to be updated if the key is missing from the table. The value in place of the key is ignored and replaced by the proper key value. The return value is as if the default object had not been used, that is, a single updated element or a list of them.

The function fails with reason `badarg` in the following situations:

- The table type is not `set` or `ordered_set`.
- No object with the correct key exists and no default object was supplied.
- The object has the wrong arity.
- The default object arity is smaller than `<keypos>`.
- Any field from the default object that is updated is not an integer.
- The element to update is not an integer.
- The element to update is also the key.
- 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 trouble of having to look up, update, and write back the entire object.

This function destructively updates the object with key `Key` in table `Tab`. The element at position `Pos` is given the value `Value`.

A list of `{Pos,Value}` can be supplied to update many elements within the same object. If the same position occurs more than once in the list, the last value in the list is written. If the list is empty or the function fails, no updates are done. The function is also atomic in the sense that other processes can never see any intermediate results.

Returns `true` if an object with key `Key` is found, otherwise `false`.

The specified `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 (for details on the difference, see *`lookup/2`* and *`new/2`*).

The function fails with reason `badarg` in the following situations:

- The table type is not `set` or `ordered_set`.

- `Pos` < 1.
- `Pos` > object arity.
- The element to update is also the key.

file_sorter

Erlang module

This module contains functions for sorting terms on files, merging already sorted files, and checking 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. Defaults to 4. The order of the header bytes is defined as follows: if `B` is a binary containing a header only, size `Size` of the binary is calculated as `<<Size:HeaderLength/unit:8>> = B`.

`{format, Format}`

Option `Format` determines the function that is applied to binaries to create the terms to be sorted. Defaults to `binary_term`, which is equivalent to `fun binary_to_term/1`. Value `binary` is equivalent to `fun(X) -> X end`, which means that the binaries are 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 option `header` is allowed.

Option `format` 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. Notice that the binary written is the same binary that was read; the results of applying function `Format` are thrown away when the terms have been sorted. Reading and writing terms using the `io` module is 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 value `descending` or by specifying an ordering function `Fun`. An ordering function is antisymmetric, transitive, and total. `Fun(A, B)` is to return `true` if `A` comes before `B` in the ordering, otherwise `false`. An example of a typical ordering function is less than or equal to, `=</2`. Using an ordering function slows down the sort considerably. Functions `keysort`, `keymerge` and `keycheck` 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`. Defaults to `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 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 unique integer (`erlang:unique_integer([positive])`). A typical name is `fs_mynode@myhost_1763_4711.17`, where `17` is a sequence number. Existing files are overwritten. Temporary files are deleted unless some uncaught `EXIT` signal occurs.

`{compressed, boolean()}`

Temporary files and the output file can be compressed. Defaults `false`, which implies that written files are not compressed. Regardless of the value of option `compressed`, compressed files can always be read. Notice that reading and writing compressed files are significantly slower than reading and writing uncompressed files.

`{size, Size}`

By default about 512*1024 bytes read from files are sorted internally. This option is rarely needed.

`{no_files, NoFiles}`

By default 16 files are merged at a time. This option is rarely needed.

As an alternative to sorting files, a function of one argument can be specified as input. When called with argument `read`, the function is assumed to return either of the following:

- `end_of_input` or `{end_of_input, Value}}` when there is no more input (`Value` is explained below).
- `{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 is called exactly once. If an error occurs, the last function is called with argument `close`, the reply of which is ignored.

A function of one argument can be specified 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 argument `close`, and the reply is returned as value of the current call to the sort or merge function.

If a function is specified as input and the last input function returns `{end_of_input, Value}`, the function specified as output is called with argument `{value, Value}`. This makes it easy to initiate the sequence of output functions with a value calculated by the input functions.

As an example, consider sorting the terms on a disk log file. A function that reads chunks from the disk log and returns a list of binaries is used as input. The results are collected in a list of terms.

```
sort(Log) ->
    {ok, _} = disk_log:open([{name,Log}, {mode,read_only}]),
    Input = input(Log, start),
    Output = output([]),
    Reply = file_sorter:sort(Input, Output, {format,term}),
    ok = disk_log:close(Log),
    Reply.

input(Log, Cont) ->
    fun(close) ->
            ok;
       (read) ->
            case disk_log:chunk(Log, Cont) of
                {error, Reason} ->
                    {error, Reason};
                {Cont2, Terms} ->
                    {Terms, input(Log, Cont2)};
                {Cont2, Terms, _Badbytes} ->
                    {Terms, input(Log, Cont2)};
                eof ->
                    end_of_input
            end
    end.

output(L) ->
    fun(close) ->
            lists:append(lists:reverse(L));
       (Terms) ->
            output([Terms | L])
    end.
```

For more examples of functions as input and output, see the end of the `file_sorter` module; the `term` format is implemented with functions.

The possible values of `Reason` returned when an error occurs are:

- `bad_object`, `{bad_object, FileName}` - Applying the format function failed for some binary, or the key(s) could not be extracted from some term.
- `{bad_term, FileName}` - `io:read/2` failed to read some term.
- `{file_error, FileName, file:posix()}` - For an explanation of `file:posix()`, see *`file(3)`*.
- `{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

```
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 = *key_pos()*
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.

`keycheck(KeyPos, FileName)` is equivalent to `keycheck(KeyPos, [FileName], [])`.

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, [])`.

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, the next element according to `KeyPos` is compared. The sort is stable.

`keysort(N, Input, Output)` is equivalent to `keysort(N, Input, Output, [])`.

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, [])`.

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, [])`.

filelib

Erlang module

This module contains utilities on a higher level than the *file* module.

This module does not support "raw" filenames (that is, files whose names do not comply with the expected encoding). Such files are ignored by the functions in this module.

For more information about raw filenames, see the *file* module.

Data Types

```
filename() = file:name()
dirname() = filename()
dirname_all() = filename_all()
filename_all() = file:name_all()
find_file_rule() =
    {ObjDirSuffix :: string(), SrcDirSuffix :: string()}
find_source_rule() =
    {ObjExtension :: string(),
     SrcExtension :: string(),
     [find_file_rule()]}
```

Exports

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

Types:

```
Name = filename_all() | dirname_all()
Reason = file:posix()
```

Ensures that all parent directories for the specified file or directory name `Name` exist, trying to create them if necessary.

Returns `ok` if all parent directories already exist or can be created. Returns `{error, Reason}` if some parent directory does not exist and cannot be created.

file_size(Filename) -> integer() >= 0

Types:

```
Filename = filename_all()
```

Returns the size of the specified file.

fold_files(Dir, RegExp, Recursive, Fun, AccIn) -> AccOut

Types:

```
Dir = dirname()
RegExp = string()
Recursive = boolean()
Fun = fun((F :: file:filename(), AccIn) -> AccOut)
AccIn = AccOut = term()
```

Folds function `Fun` over all (regular) files `F` in directory `Dir` that match the regular expression `RegExp` (for a description of the allowed regular expressions, see the *re* module). If `Recursive` is `true`, all subdirectories to `Dir` are processed. The regular expression matching is only done on the filename without the directory part.

If Unicode filename translation is in effect and the file system is transparent, filenames that cannot be interpreted as Unicode can be encountered, in which case the `fun()` must be prepared to handle raw filenames (that is, binaries). If the regular expression contains codepoints > 255, it does not match filenames that do not conform to the expected character encoding (that is, are not encoded in valid UTF-8).

For more information about raw filenames, see the *file* module.

is_dir(Name) -> boolean()

Types:

```
Name = filename_all() | dirname_all()
```

Returns `true` if `Name` refers to a directory, otherwise `false`.

is_file(Name) -> boolean()

Types:

```
Name = filename_all() | dirname_all()
```

Returns `true` if `Name` refers to a file or a directory, otherwise `false`.

is_regular(Name) -> boolean()

Types:

```
Name = filename_all()
```

Returns `true` if `Name` refers to a (regular) file, otherwise `false`.

last_modified(Name) -> file:date_time() | 0

Types:

```
Name = filename_all() | dirname_all()
```

Returns the date and time the specified file or directory was last modified, or `0` if the file does not exist.

wildcard(Wildcard) -> [file:filename()]

Types:

```
Wildcard = filename() | dirname()
```

Returns a list of all files that match Unix-style wildcard string `Wildcard`.

The wildcard string looks like an ordinary filename, except that the following "wildcard characters" are interpreted in a special way:

?

Matches one character.

*

Matches any number of characters up to the end of the filename, the next dot, or the next slash.

**

Two adjacent `*` used as a single pattern match all files and zero or more directories and subdirectories.

[Character1,Character2,...]

Matches any of the characters listed. Two characters separated by a hyphen match a range of characters. Example: `[A-Z]` matches any uppercase letter.

{Item,...}

Alternation. Matches one of the alternatives.

Other characters represent themselves. Only filenames that have exactly the same character in the same position match. Matching is case-sensitive, for example, "a" does not match "A".

Notice 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, use the following line:

```
filelib:wildcard("lib/*/ebin/*.beam").
```

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

```
filelib:wildcard("lib/*/src/*.?rl")
```

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

To find all `.hrl` files in `src` or `include` directories:

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

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

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

To find all `.erl` or `.hrl` files in any subdirectory:

```
filelib:wildcard("lib/**/*.{erl,hrl}")
```

`wildcard(Wildcard, Cwd) -> [file:filename()]`

Types:

`Wildcard = filename() | dirname()`

`Cwd = dirname()`

Same as *`wildcard/1`*, except that `Cwd` is used instead of the working directory.

```
find_file(Filename :: filename(), Dir :: filename()) ->
             {ok, filename()} | {error, not_found}
find_file(Filename :: filename(),
          Dir :: filename(),
          Rules :: [find_file_rule()]) ->
```

{ok, *filename()*} | {error, not_found}

Looks for a file of the given name by applying suffix rules to the given directory path. For example, a rule `{"ebin", "src"}` means that if the directory path ends with `"ebin"`, the corresponding path ending in `"src"` should be searched.

If `Rules` is left out or is an empty list, the default system rules are used. See also the Kernel application parameter *`source_search_rules`*.

find_source(FilePath :: *filename()*) ->
{ok, *filename()*} | {error, not_found}

Equivalent to `find_source(Base, Dir)`, where `Dir` is `filename:dirname(FilePath)` and `Base` is `filename:basename(FilePath)`.

find_source(Filename :: *filename()*, Dir :: *filename()*) ->
{ok, *filename()*} | {error, not_found}
find_source(Filename :: *filename()*,
Dir :: *filename()*,
Rules :: [*find_source_rule()*]) ->
{ok, *filename()*} | {error, not_found}

Applies file extension specific rules to find the source file for a given object file relative to the object directory. For example, for a file with the extension `.beam`, the default rule is to look for a file with a corresponding extension `.erl` by replacing the suffix `"ebin"` of the object directory path with `"src"` or `"src/*"`. The file search is done through *`find_file/3`*. The directory of the object file is always tried before any other directory specified by the rules.

If `Rules` is left out or is an empty list, the default system rules are used. See also the Kernel application parameter *`source_search_rules`*.

filename

Erlang module

This module provides functions for analyzing and manipulating filenames. These functions are designed so that the Erlang code can work on many different platforms with different filename formats. With filename is meant all strings that can be used to denote a file. The filename can be a short relative name like `foo.erl`, a long absolute name including a drive designator, a directory name like `D:\usr/local\bin\erl/lib\tools\foo.erl`, or any variations in between.

In Windows, all functions return filenames with forward slashes only, even if the arguments contain backslashes. To normalize a filename by removing redundant directory separators, use *`join/1`*.

The module supports raw filenames in the way that if a binary is present, or the filename cannot be interpreted according to the return value of *`file:native_name_encoding/0`*, a raw filename is also returned. For example, `join/1` provided with a path component that is a binary (and cannot be interpreted under the current native filename encoding) results in a raw filename that is returned (the join operation is performed of course). For more information about raw filenames, see the *`file`* module.

Data Types

```
basedir_type() =
    user_cache |
    user_config |
    user_data |
    user_log |
    site_config |
    site_data
```

Exports

`absname(Filename) -> file:filename_all()`

Types:

`Filename = file:name_all()`

Converts a relative `Filename` and returns an absolute name. No attempt is made to create the shortest absolute name, as this can give incorrect results on file systems that allow links.

Unix examples:

```
1> pwd().
"/usr/local"
2> filename:absname("foo").
"/usr/local/foo"
3> filename:absname("../x").
"/usr/local/../x"
4> filename:absname("/").
"/"
```

Windows examples:

```
1> pwd().
"D:/usr/local"
2> filename:absname("foo").
"D:/usr/local/foo"
3> filename:absname("../x").
"D:/usr/local/../x"
4> filename:absname("/").
"D:/"
```

absname(Filename, Dir) -> *file:filename_all()*

Types:

Filename = Dir = *file:name_all()*

Same as *absname/1*, except that the directory to which the filename is to be made relative is specified in argument `Dir`.

absname_join(Dir, Filename) -> *file:filename_all()*

Types:

Dir = Filename = *file:name_all()*

Joins an absolute directory with a relative filename. Similar to *join/2*, but on platforms with tight restrictions on raw filename length and no support for symbolic links (read: VxWorks), leading parent directory components in `Filename` are matched against trailing directory components in `Dir` so they can be removed from the result - minimizing its length.

basedir(Type, Application) -> *file:filename_all()*

Types:

Type = *basedir_type()*
Application = string() | binary()

Equivalent to *basedir(Type, Application, #{})*.

basedir(Type, Application, Opts) -> *file:filename_all()*

Types:

Type = *basedir_type()*
Application = string() | binary()
Opts =
#{author => string() | binary(),
os => windows | darwin | linux,
version => string() | binary()}

Returns a suitable path, or paths, for a given type. If `os` is not set in `Opts` the function will default to the native option, that is `'linux'`, `'darwin'` or `'windows'`, as understood by `os:type/0`. Anything not recognized as `'darwin'` or `'windows'` is interpreted as `'linux'`.

The options `'author'` and `'version'` are only used with `'windows'` option mode.

- `user_cache`

  The path location is intended for transient data files on a local machine.

  On Linux: Respects the os environment variable `XDG_CACHE_HOME`.

```
1> filename:basedir(user_cache, "my_application", #{os=>linux}).
"/home/otptest/.cache/my_application"
```

On Darwin:

```
1> filename:basedir(user_cache, "my_application", #{os=>darwin}).
"/home/otptest/Library/Caches/my_application"
```

On Windows:

```
1> filename:basedir(user_cache, "My App").
"c:/Users/otptest/AppData/Local/My App/Cache"
2> filename:basedir(user_cache, "My App").
"c:/Users/otptest/AppData/Local/My App/Cache"
3> filename:basedir(user_cache, "My App", #{author=>"Erlang"}).
"c:/Users/otptest/AppData/Local/Erlang/My App/Cache"
4> filename:basedir(user_cache, "My App", #{version=>"1.2"}).
"c:/Users/otptest/AppData/Local/My App/1.2/Cache"
5> filename:basedir(user_cache, "My App", #{author=>"Erlang",version=>"1.2"}).
"c:/Users/otptest/AppData/Local/Erlang/My App/1.2/Cache"
```

- `user_config`

  The path location is intended for persistent configuration files.

  On Linux: Respects the os environment variable `XDG_CONFIG_HOME`.

  ```
  2> filename:basedir(user_config, "my_application", #{os=>linux}).
  "/home/otptest/.config/my_application"
  ```

  On Darwin:

  ```
  2> filename:basedir(user_config, "my_application", #{os=>darwin}).
  "/home/otptest/Library/Application Support/my_application"
  ```

  On Windows:

  ```
  1> filename:basedir(user_config, "My App").
  "c:/Users/otptest/AppData/Roaming/My App"
  2> filename:basedir(user_config, "My App", #{author=>"Erlang", version=>"1.2"}).
  "c:/Users/otptest/AppData/Roaming/Erlang/My App/1.2"
  ```

- `user_data`

  The path location is intended for persistent data files.

  On Linux: Respects the os environment variable `XDG_DATA_HOME`.

  ```
  3> filename:basedir(user_data, "my_application", #{os=>linux}).
  "/home/otptest/.local/my_application"
  ```

  On Darwin:

  ```
  3> filename:basedir(user_data, "my_application", #{os=>darwin}).
  "/home/otptest/Library/Application Support/my_application"
  ```

  On Windows:

```
8> filename:basedir(user_data, "My App").
"c:/Users/otptest/AppData/Local/My App"
9> filename:basedir(user_data, "My App",#{author=>"Erlang",version=>"1.2"}).
"c:/Users/otptest/AppData/Local/Erlang/My App/1.2"
```

- `user_log`

  The path location is intended for transient log files on a local machine.

  On Linux: Respects the os environment variable `XDG_CACHE_HOME`.

```
4> filename:basedir(user_log, "my_application", #{os=>linux}).
"/home/otptest/.cache/my_application/log"
```

  On Darwin:

```
4> filename:basedir(user_log, "my_application", #{os=>darwin}).
"/home/otptest/Library/Caches/my_application"
```

  On Windows:

```
12> filename:basedir(user_log, "My App").
"c:/Users/otptest/AppData/Local/My App/Logs"
13> filename:basedir(user_log, "My App",#{author=>"Erlang",version=>"1.2"}).
"c:/Users/otptest/AppData/Local/Erlang/My App/1.2/Logs"
```

- `site_config`

  On Linux: Respects the os environment variable `XDG_CONFIG_DIRS`.

```
5> filename:basedir(site_data, "my_application", #{os=>linux}).
["/usr/local/share/my_application",
 "/usr/share/my_application"]
6> os:getenv("XDG_CONFIG_DIRS").
"/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg"
7> filename:basedir(site_config, "my_application", #{os=>linux}).
["/etc/xdg/xdg-ubuntu/my_application",
 "/usr/share/upstart/xdg/my_application",
 "/etc/xdg/my_application"]
8> os:unsetenv("XDG_CONFIG_DIRS").
true
9> filename:basedir(site_config, "my_application", #{os=>linux}).
["/etc/xdg/my_application"]
```

  On Darwin:

```
5> filename:basedir(site_config, "my_application", #{os=>darwin}).
["/Library/Application Support/my_application"]
```

- `site_data`

  On Linux: Respects the os environment variable `XDG_DATA_DIRS`.

```
10> os:getenv("XDG_DATA_DIRS").
"/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/usr/share/"
11> filename:basedir(site_data, "my_application", #{os=>linux}).
["/usr/share/ubuntu/my_application",
 "/usr/share/gnome/my_application",
 "/usr/local/share/my_application",
 "/usr/share/my_application"]
12> os:unsetenv("XDG_DATA_DIRS").
true
13> filename:basedir(site_data, "my_application", #{os=>linux}).
["/usr/local/share/my_application",
 "/usr/share/my_application"]
```

On Darwin:

```
5> filename:basedir(site_data, "my_application", #{os=>darwin}).
["/Library/Application Support/my_application"]
```

basename(Filename) -> *file:filename_all()*

Types:

Filename = *file:name_all()*

Returns the last component of `Filename`, or `Filename` itself if it does not contain any directory separators.

Examples:

```
5> filename:basename("foo").
"foo"
6> filename:basename("/usr/foo").
"foo"
7> filename:basename("/").
[]
```

basename(Filename, Ext) -> *file:filename_all()*

Types:

Filename = Ext = *file:name_all()*

Returns the last component of `Filename` with extension `Ext` stripped. This function is to be used to remove a (possible) specific extension. To remove an existing extension when you are unsure which one it is, use `rootname(basename(Filename))`.

Examples:

```
8> filename:basename("~/src/kalle.erl", ".erl").
"kalle"
9> filename:basename("~/src/kalle.beam", ".erl").
"kalle.beam"
10> filename:basename("~/src/kalle.old.erl", ".erl").
"kalle.old"
11> filename:rootname(filename:basename("~/src/kalle.erl")).
"kalle"
12> filename:rootname(filename:basename("~/src/kalle.beam")).
"kalle"
```

dirname(Filename) -> *file:filename_all()*

Types:

Filename = *file:name_all()*

Returns the directory part of `Filename`.

Examples:

```
13> filename:dirname("/usr/src/kalle.erl").
"/usr/src"
14> filename:dirname("kalle.erl").
"."
```

```
5> filename:dirname("\\usr\\src/kalle.erl"). % Windows
"/usr/src"
```

extension(Filename) -> *file:filename_all()*

Types:

Filename = *file:name_all()*

Returns the file extension of `Filename`, including the period. Returns an empty string if no extension exists.

Examples:

```
15> filename:extension("foo.erl").
".erl"
16> filename:extension("beam.src/kalle").
[]
```

```
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`* to compile the file again.

Warning:

This function is deprecated. Use *filelib:find_source/1* instead for finding source files.

If possible, use the *beam_lib(3)* module to extract the compiler options and the abstract code format from the Beam file and compile that instead.

Argument `Beam`, which can be a string or an atom, specifies either the module name or the path to the source code, with or without extension `".erl"`. In either case, the module must be known by the code server, that is, `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 name created by replacing `BinSuffix` with `SourceSuffix` is expanded by calling *filelib:wildcard/1*. If a regular file is found among the matches, the function returns that location together with `Options`. Otherwise the next rule is tried, and so on.

`Rules` defaults to:

```
[{"", ""}, {"ebin", "src"}, {"ebin", "esrc"},
 {"ebin", "src/*"}, {"ebin", "esrc/*"}]
```

The function returns `{SourceFile, Options}` if it succeeds. `SourceFile` is the absolute path to the source file without extension `".erl"`. `Options` includes the options that are necessary to recompile the file with `compile:file/2`, but excludes options such as `report` and `verbose`, which do not change the way code is generated. The paths in options `{outdir, Path}` and `{i, Path}` are guaranteed to be absolute.

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 filename `Components` with directory separators. If one of the elements of `Components` includes an absolute path, such as `"/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.

Examples:

```
17> filename:join(["/usr", "local", "bin"]).
"/usr/local/bin"
18> filename:join(["a/b///c/"]).
"a/b/c"
```

```
6> filename:join(["B:a\\b///c/"]). % Windows
"b:a/b/c"
```

join(Name1, Name2) -> *file:filename_all()*

Types:

Name1 = Name2 = *file:name_all()*

Joins two filename components with directory separators. Equivalent to `join([Name1, Name2])`.

nativename(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 are converted to backward slashes. On all platforms, the name is normalized as done by *`join/1`*.

Examples:

```
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 path type, which is one of the following:

`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()*

Removes a filename extension. `rootname/2` works as `rootname/1`, except that the extension is removed only if it is `Ext`.

Examples:

```
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"
```

safe_relative_path(Filename) -> unsafe | SafeFilename

Types:

Filename = SafeFilename = *file:name_all()*

Sanitizes the relative path by eliminating ".." and "." components to protect against directory traversal attacks. Either returns the sanitized path name, or the atom `unsafe` if the path is unsafe. The path is considered unsafe in the following circumstances:

- The path is not relative.
- A ".." component would climb up above the root of the relative path.

Examples:

```
1> filename:safe_relative_path("dir/sub_dir/..").
"dir"
2> filename:safe_relative_path("dir/..").
[]
3> filename:safe_relative_path("dir/../..").
unsafe
4> filename:safe_relative_path("/abs/path").
unsafe
```

split(Filename) -> Components

Types:

Filename = *file:name_all()*

Components = [*file:name_all()*]

Returns a list whose elements are the path components of `Filename`.

Examples:

```
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"]
```

gb_sets

Erlang module

This module provides ordered sets using Prof. Arne Andersson's General Balanced Trees. Ordered sets 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| * 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 10-100 times. This implementation is particularly suited for accumulating elements a few at a time, building up a large set (> 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

The following functions in this module also exist and provides the same functionality in the *sets(3)* and *ordsets(3)* 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 general balanced set.

set() = *set*(term())
iter(Element)

A general balanced set iterator.

iter() = *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`. Notice that this is rarely necessary, but can be motivated when a large number of elements have been deleted from the tree without further insertions. Rebalancing can then be forced to minimise lookup times, as deletion does not rebalance the tree.

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.

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

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

Types:

Set1 = Set2 = Set3 = *set*(Element)

Returns only the elements of `Set1` that are not also elements of `Set2`.

empty() -> Set

Types:

Set = *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` can 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(SetList) -> Set

Types:

SetList = [*set*(Element), ...]
Set = *set*(Element)

Returns the intersection of the non-empty list of sets.

intersection(Set1, Set2) -> Set3

Types:

Set1 = Set2 = Set3 = *set*(Element)

Returns the intersection of `Set1` and `Set2`.

is_disjoint(Set1, Set2) -> boolean()

Types:

Set1 = Set2 = *set*(Element)

Returns `true` if `Set1` and `Set2` are disjoint (have no elements in common), otherwise `false`.

is_element(Element, Set) -> boolean()

Types:

Set = *set*(Element)

Returns `true` if `Element` is an element of `Set`, otherwise `false`.

is_empty(Set) -> boolean()

Types:

Set = *set()*

Returns `true` if `Set` is an empty set, otherwise `false`.

is_member(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.

iterator_from(Element, 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 difference as compared to the iterator returned by *iterator/1* is that the first element greater than or equal to `Element` is returned.

largest(Set) -> Element

Types:

Set = *set*(Element)

Returns the largest element in `Set`. Assumes that `Set` is not empty.

new() -> Set

Types:

Set = *set()*

Returns a new empty set.

next(Iter1) -> {Element, Iter2} | none

Types:

Iter1 = Iter2 = *iter*(Element)

Returns `{Element, Iter2}`, where `Element` is the smallest element referred to by 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 element `Element`.

size(Set) -> integer() >= 0

Types:

Set = *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 not empty.

subtract(Set1, Set2) -> Set3

Types:

Set1 = Set2 = Set3 = *set*(Element)

Returns only the elements of `Set1` that are not also elements of `Set2`.

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 not empty.

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 not empty.

to_list(Set) -> List

Types:

Set = *set*(Element)
List = [Element]

Returns the elements of `Set` as a list.

union(SetList) -> Set

Types:

SetList = [*set*(Element), ...]
Set = *set*(Element)

Returns the merged (union) set of the list of sets.

union(Set1, Set2) -> Set3

Types:

Set1 = Set2 = Set3 = *set*(Element)

Returns the merged (union) set of `Set1` and `Set2`.

See Also

gb_trees(3), *ordsets(3)*, *sets(3)*

gb_trees

Erlang module

This module provides Prof. Arne Andersson's General Balanced Trees. These have no storage overhead compared to unbalanced binary trees, and their performance is better than AVL trees.

This module considers two keys as different if and only if they do not compare equal (==).

Data Structure

```
{Size, Tree}
```

`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. As deletions do not increase the height of a tree, this should be OK.

The original balance condition **h(T) <= ceil(c * log(|T|))** has been changed to the similar (but not quite equivalent) condition **2 ^ h(T) <= |T| ^ c**. This should also be OK.

Data Types

`tree(Key, Value)`

A general balanced tree.

`tree() = tree(term(), term())`

`iter(Key, Value)`

A general balanced tree iterator.

`iter() = iter(term(), term())`

Exports

`balance(Tree1) -> Tree2`

Types:

`Tree1 = Tree2 = tree(Key, Value)`

Rebalances `Tree1`. Notice that this is rarely necessary, but can be motivated when many nodes have been deleted from the tree without further insertions. Rebalancing can then be forced to minimize lookup times, as deletion does not rebalance the tree.

`delete(Key, Tree1) -> Tree2`

Types:

`Tree1 = Tree2 = tree(Key, Value)`

Removes the node with key `Key` from `Tree1` and returns the 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 the new tree.

take(Key, Tree1) -> {Value, Tree2}

Types:

Tree1 = Tree2 = *tree*(Key, term())
Key = Value = term()

Returns a value `Value` from node with key `Key` and new `Tree2` without the node with this value. Assumes that the node with key is present in the tree, crashes otherwise.

take_any(Key, Tree1) -> {Value, Tree2} | error

Types:

Tree1 = Tree2 = *tree*(Key, term())
Key = Value = term()

Returns a value `Value` from node with key `Key` and new `Tree2` without the node with this value. Returns `error` if the node with the key is not present in the 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` and 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, othwewise `false`.

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.

iterator_from(Key, 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 difference as compared to the iterator returned by *`iterator/1`* is that the first key greater than or equal to `Key` is returned.

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 not empty.

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 function F(K, V1) -> V2 to all key-value pairs of tree `Tree1`. 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 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 not empty.

`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 not empty.

`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 not empty.

`to_list(Tree) -> [{Key, Value}]`

Types:

```
Tree = tree(Key, 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` and 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

dict(3), *gb_sets(3)*

gen_event

Erlang module

This behavior module provides event handling functionality. It consists of a generic event manager process with any number of event handlers that are added and deleted dynamically.

An event manager implemented using this module has a standard set of interface functions and includes functionality for tracing and error reporting. It also fits into an OTP supervision tree. For more information, see *OTP Design Principles*.

Each event handler is implemented as a callback module exporting a predefined set of functions. The relationship between the behavior functions and the callback functions is as follows:

```
gen_event module                    Callback module
----------------                    ---------------
gen_event:start
gen_event:start_link       ----->   -

gen_event:add_handler
gen_event:add_sup_handler  ----->   Module:init/1

gen_event:notify
gen_event:sync_notify      ----->   Module:handle_event/2

gen_event:call             ----->   Module:handle_call/2

-                          ----->   Module:handle_info/2

gen_event:delete_handler   ----->   Module:terminate/2

gen_event:swap_handler
gen_event:swap_sup_handler ----->   Module1:terminate/2
                                    Module2:init/1

gen_event:which_handlers   ----->   -

gen_event:stop             ----->   Module:terminate/2

-                          ----->   Module:code_change/3
```

As each event handler is one callback module, an event manager has many callback modules that are added and deleted dynamically. `gen_event` is therefore more tolerant of callback module errors than the other behaviors. If a callback function for an installed event handler fails with `Reason`, or returns a bad value `Term`, the event manager does not fail. It deletes the event handler by calling callback function *`Module:terminate/2`*, giving as argument `{error, {'EXIT',Reason}}` or `{error,Term}`, respectively. No other event handler is affected.

A `gen_event` process handles system messages as described in *`sys(3)`*. The `sys` module can be used for debugging an event manager.

Notice that an event manager **does** trap exit signals automatically.

The `gen_event` process can go into hibernation (see *`erlang:hibernate/3`*) if a callback function in a handler module specifies `hibernate` in its return value. This can be useful if the server is expected to be idle for a long time. However, use this feature with care, as hibernation implies at least two garbage collections (when hibernating and shortly after waking up) and is not something you want to do between each event handled by a busy event manager.

Notice that when multiple event handlers are invoked, it is 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 specified.

Data Types

```
handler() = atom() | {atom(), term()}
handler_args() = term()
add_handler_ret() = ok | term() | {'EXIT', term()}
del_handler_ret() = ok | term() | {'EXIT', term()}
```

Exports

`add_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 = ok | {'EXIT',Reason} | term()
 Reason = term()
```

Adds a new event handler to event manager `EventMgrRef`. The event manager calls *`Module:init/1`* to initiate the event handler and its internal state.

`EventMgrRef` can be any of the following:

- The pid
- `Name`, if the event manager is locally registered
- `{Name,Node}`, if the event manager is locally registered at another node
- `{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 many event handlers use the same callback module.

`Args` is any term that 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:

```
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 = ok | {'EXIT',Reason} | term()
 Reason = term()
```

Adds a new event handler in the same way as *add_handler/3*, but also supervises the connection between the event handler and the calling process.

- If the calling process later terminates with `Reason`, the event manager deletes the event handler by calling *Module:terminate/2* with `{stop,Reason}` as argument.
- If the event handler is deleted later, 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 because of 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 because of an error. Which term depends on the error.

For a description of the arguments and return values, see *add_handler/3*.

```
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 event handler `Handler` installed in event manager `EventMgrRef` by sending a request and waiting until a reply arrives or a time-out occurs. The event manager calls *Module:handle_call/2* to handle the request.

For a description of `EventMgrRef` and `Handler`, see *add_handler/3*.

`Request` is any term that is passed as one of the arguments to `Module:handle_call/2`.

`Timeout` is an integer greater than zero that specifies how many milliseconds to wait for a reply, or the atom `infinity` to wait indefinitely. Defaults to 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 event manager `EventMgrRef`. The event manager calls *`Module:terminate/2`* to terminate the event handler.

For a description of `EventMgrRef` and `Handler`, see *`add_handler/3`*.

`Args` is any term that 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}`.

`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 event manager `EventMgrRef`. The event manager calls *`Module:handle_event/2`* for each installed event handler to handle the event.

`notify/2` is asynchronous and returns immediately after the event notification has been sent. `sync_notify/2` is synchronous in the sense that it returns `ok` after the event has been handled by all event handlers.

For a description of `EventMgrRef`, see *`add_handler/3`*.

`Event` is any term that is passed as one of the arguments to *`Module:handle_event/2`*.

`notify/1` does not fail even if the specified event manager does not exist, unless it is specified as `Name`.

```
start() -> Result
start(EventMgrName | Options) -> Result
start(EventMgrName, Options) -> Result
```

Types:

```
   EventMgrName = {local,Name} | {global,GlobalName} | {via,Module,ViaName}
    Name = atom()
    GlobalName = ViaName = term()
   Options = [Option]
    Option = {debug,Dbgs} | {timeout,Time} |
   {hibernate_after,HibernateAfterTimeout} | {spawn_opt,SOpts}
     Dbgs = [Dbg]
      Dbg = trace | log | statistics | {log_to_file,FileName} | {install,
   {Func,FuncState}}
     SOpts = [term()]
   Result = {ok,Pid} | {error,{already_started,Pid}}
    Pid = pid()
```

Creates a stand-alone event manager process, that is, an event manager that is not part of a supervision tree and thus has no supervisor.

For a description of the arguments and return values, see *`start_link/0,1`*.

```
start_link() -> Result
start_link(EventMgrName | Options) -> Result
start_link(EventMgrName, Options) -> Result
```

Types:

```
   EventMgrName = {local,Name} | {global,GlobalName} | {via,Module,ViaName}
    Name = atom()
    GlobalName = ViaName = term()
   Options = [Option]
    Option = {debug,Dbgs} | {timeout,Time} |
   {hibernate_after,HibernateAfterTimeout} | {spawn_opt,SOpts}
     Dbgs = [Dbg]
      Dbg = trace | log | statistics | {log_to_file,FileName} | {install,
   {Func,FuncState}}
     SOpts = [term()]
   Result = {ok,Pid} | {error,{already_started,Pid}}
    Pid = pid()
```

Creates an event manager process as part of a supervision tree. The function is to be called, directly or indirectly, by the supervisor. For example, it ensures 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 registers with the registry represented by `Module`. The `Module` callback is to export the functions `register_name/2`, `unregister_name/1`,

whereis_name/1, and send/2, which are to behave as the corresponding functions in *global*. Thus, {via,global,GlobalName} is a valid reference.

- If option {hibernate_after,HibernateAfterTimeout} is present, the gen_event process awaits any message for HibernateAfterTimeout milliseconds and if no message is received, the process goes into hibernation automatically (by calling *proc_lib:hibernate/3*).

If the event manager is successfully created, the function returns {ok,Pid}, where Pid is the pid of the event manager. If a process with the specified EventMgrName exists already, the function returns {error, {already_started,Pid}}, where Pid is the pid of that process.

stop(EventMgrRef) -> ok

stop(EventMgrRef, Reason, Timeout) -> ok

Types:

```
EventMgrRef = Name | {Name,Node} | {global,GlobalName} |
{via,Module,ViaName} | pid()
Name = Node = atom()
GlobalName = ViaName = term()
Reason = term()
Timeout = int()>0 | infinity
```

Orders event manager EventMgrRef to exit with the specifies Reason and waits for it to terminate. Before terminating, gen_event calls *Module:terminate(stop,...)* for each installed event handler.

The function returns ok if the event manager terminates with the expected reason. Any other reason than normal, shutdown, or {shutdown,Term} causes an error report to be issued using *error_logger:format/2*. The default Reason is normal.

Timeout is an integer greater than zero that specifies how many milliseconds to wait for the event manager to terminate, or the atom infinity to wait indefinitely. Defaults to infinity. If the event manager has not terminated within the specified time, a timeout exception is raised.

If the process does not exist, a noproc exception is raised.

For a description of EventMgrRef, see *add_handler/3*.

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 event manager EventMgrRef.

For a description of the arguments, see *add_handler/3*.

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` is the return value of `Module1:terminate/2`. This makes it possible to transfer information from `Handler1` to `Handler2`.

The new handler is 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 is deleted even if `Module2:init/1` fails.

If there was a supervised connection between `Handler1` and a process `Pid`, there is 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 event manager `EventMgrRef` in the same way as `swap_handler/3`, but also supervises the connection between `Handler2` and the calling process.

For a description of the arguments and return values, see *`swap_handler/3`*.

`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 event manager `EventMgrRef`.

For a description of `EventMgrRef` and `Handler`, see *`add_handler/3`*.

Callback Functions

The following functions are to be exported from a `gen_event` callback module.

Exports

`Module:code_change(OldVsn, State, Extra) -> {ok, NewState}`

Types:

```
OldVsn = Vsn | {down, Vsn}
  Vsn = term()
State = NewState = term()
Extra = term()
```

> **Note:**
>
> This callback is optional, so callback modules need not export it. If a release upgrade/downgrade with `Change={advanced,Extra}` specified in the `.appup` file is made when `code_change/3` isn't implemented the event handler will crash with an `undef` error reason.

This function is called for an installed event handler that is to update its internal state during a release upgrade/downgrade, that is, when the instruction `{update,Module,Change,...}`, where `Change={advanced,Extra}`, is specified in the `.appup` file. For more information, see *OTP Design Principles*.

For an upgrade, `OldVsn` is `Vsn`, and for a downgrade, `OldVsn` is `{down,Vsn}`. `Vsn` is defined by the `vsn` attribute(s) of the old version of the callback module `Module`. If no such attribute is defined, the version is the checksum of the Beam file.

`State` is the internal state of the event handler.

`Extra` is passed "as is" from the `{advanced,Extra}` part of the update instruction.

The function is to return the updated internal state.

`Module:format_status(Opt, [PDict, State]) -> Status`

Types:

```
Opt = normal | terminate
PDict = [{Key, Value}]
State = term()
Status = term()
```

> **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 in the following situations:

- One of *`sys:get_status/1,2`* is invoked to get the `gen_event` status. `Opt` is set to the atom `normal` for this case.
- The event handler terminates abnormally and `gen_event` logs an error. `Opt` is set to the atom `terminate` for this case.

This function is useful for changing the form and appearance of the event handler state for these cases. An event handler callback module wishing to change the the `sys:get_status/1,2` return value as well as how its state appears in termination error logs, exports an instance of `format_status/2` that returns a term describing the current state of the event handler.

`PDict` is the current value of the process dictionary of `gen_event`.

`State` is the internal state of the event handler.

The function is to return `Status`, a term that change the details of the current state of the event handler. Any term is allowed for `Status`. The `gen_event` module uses `Status` as follows:

- When `sys:get_status/1,2` is called, `gen_event` ensures that its return value contains `Status` in place of the state term of the event handler.
- When an event handler terminates abnormally, `gen_event` logs `Status` in place of the state term of the event handler.

One use for this function is to return compact alternative state representations to avoid that large state terms are printed in log files.

`Module:handle_call(Request, State) -> Result`

Types:

```
Request = term()
State = term()
Result = {ok,Reply,NewState} | {ok,Reply,NewState,hibernate}
 | {swap_handler,Reply,Args1,NewState,Handler2,Args2}
 | {remove_handler, Reply}
 Reply = term()
 NewState = term()
 Args1 = Args2 = term()
 Handler2 = Module2 | {Module2,Id}
  Module2 = atom()
  Id = term()
```

Whenever an event manager receives a request sent using *`call/3,4`*, this function is called for the specified event handler to handle the request.

`Request` is the `Request` argument of `call/3,4`.

`State` is the internal state of the event handler.

The return values are the same as for *`Module:handle_event/2`* except that they also contain a term `Reply`, which is the reply to the client as the return value of `call/3,4`.

`Module:handle_event(Event, State) -> Result`

Types:

```
Event = term()
State = term()
Result = {ok,NewState} | {ok,NewState,hibernate}
  | {swap_handler,Args1,NewState,Handler2,Args2} | remove_handler
 NewState = term()
 Args1 = Args2 = term()
```

```
Handler2 = Module2 | {Module2,Id}
 Module2 = atom()
 Id = term()
```

Whenever an event manager receives an event sent using *notify/2* or *sync_notify/2*, this function is called for each installed event handler to handle the event.

`Event` is the `Event` argument of `notify/2`/`sync_notify/2`.

`State` is the internal state of the event handler.

- If `{ok,NewState}` or `{ok,NewState,hibernate}` is returned, the event handler remains in the event manager with the possible updated internal state `NewState`.
- If `{ok,NewState,hibernate}` is returned, the event manager also goes into hibernation (by calling *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 `{swap_handler,Args1,NewState,Handler2,Args2}` is returned, the event handler is replaced by `Handler2` by first calling `Module:terminate(Args1,NewState)` and then `Module2:init({Args2,Term})`, where `Term` is the return value of `Module:terminate/2`. For more information, see *swap_handler/3*.
- If `remove_handler` is returned, the event handler is deleted by calling `Module:terminate(remove_handler,State)`.

`Module:handle_info(Info, State) -> Result`

Types:

```
Info = term()
State = term()
Result = {ok,NewState} | {ok,NewState,hibernate}
 | {swap_handler,Args1,NewState,Handler2,Args2} | remove_handler
 NewState = term()
 Args1 = Args2 = term()
 Handler2 = Module2 | {Module2,Id}
  Module2 = atom()
  Id = term()
```

> **Note:**
>
> This callback is optional, so callback modules need not export it. The `gen_event` module provides a default implementation of this function that logs about the unexpected `Info` message, drops it and returns `{noreply, State}`.

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.

For a description of `State` and possible return values, see *Module:handle_event/2*.

`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 because of a call to *add_handler/3* or *add_sup_handler/3*, `InitArgs` is the `Args` argument of these functions.

If the event handler replaces another event handler because of a call to *swap_handler/3* or *swap_sup_handler/3*, or because of 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 *swap_handler/3*.

If successful, the function returns `{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 goes into hibernation (by calling *proc_lib:hibernate/3*), waiting for the next event to occur.

`Module:terminate(Arg, State) -> term()`

Types:

```
Arg = Args | {stop,Reason} | stop | remove_handler
 | {error,{'EXIT',Reason}} | {error,Term}
 Args = Reason = Term = term()
```

Note:

This callback is optional, so callback modules need not export it. The `gen_event` module provides a default implementation without cleanup.

Whenever an event handler is deleted from an event manager, this function is called. It is to be the opposite of *Module:init/1* and do any necessary cleaning up.

If the event handler is deleted because of a call to *delete_handler/3*, *swap_handler/3*, or *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 that has terminated with reason `Reason`.

`Arg=stop` if the event handler is deleted because the event manager is terminating.

The event manager terminates 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 terminates 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 can return any term. If the event handler is deleted because of a call to `gen_event:delete_handler/3`, the return value of that function becomes the return value of this function. If the event handler is to be replaced with another event handler because of a swap, the return value is passed to the `init` function of the new event handler. Otherwise the return value is ignored.

See Also

supervisor(3), *sys(3)*

gen_fsm

Erlang module

Deprecated and replaced by *gen_statem*

Migration to gen_statem

Here follows a simple example of turning a gen_fsm into a *gen_statem*. The example comes from the previous Users Guide for gen_fsm

```
-module(code_lock).
-define(NAME, code_lock).
%-define(BEFORE_REWRITE, true).

-ifdef(BEFORE_REWRITE).
-behaviour(gen_fsm).
-else.
-behaviour(gen_statem).
-endif.

-export([start_link/1, button/1, stop/0]).

-ifdef(BEFORE_REWRITE).
-export([init/1, locked/2, open/2, handle_sync_event/4, handle_event/3,
  handle_info/3, terminate/3, code_change/4]).
-else.
-export([init/1, callback_mode/0, locked/3, open/3, terminate/3, code_change/4]).
%% Add callback__mode/0
%% Change arity of the state functions
%% Remove handle_info/3
-endif.

-ifdef(BEFORE_REWRITE).
start_link(Code) ->
    gen_fsm:start_link({local, ?NAME}, ?MODULE, Code, []).
-else.
start_link(Code) ->
    gen_statem:start_link({local,?NAME}, ?MODULE, Code, []).
-endif.

-ifdef(BEFORE_REWRITE).
button(Digit) ->
    gen_fsm:send_event(?NAME, {button, Digit}).
-else.
button(Digit) ->
    gen_statem:cast(?NAME, {button,Digit}).
    %% send_event is asynchronous and becomes a cast
-endif.

-ifdef(BEFORE_REWRITE).
stop() ->
    gen_fsm:sync_send_all_state_event(?NAME, stop).
-else.
stop() ->
    gen_statem:call(?NAME, stop).
    %% sync_send is synchronous and becomes call
    %% all_state is handled by callback code in gen_statem
-endif.

init(Code) ->
    do_lock(),
    Data = #{code => Code, remaining => Code},
    {ok, locked, Data}.

-ifdef(BEFORE_REWRITE).
-else.
callback_mode() ->
    state_functions.
%% state_functions mode is the mode most similar to
%% gen_fsm. There is also handle_event mode which is
%% a fairly different concept.
-endif.

-ifdef(BEFORE_REWRITE).
locked({button, Digit}, Data0) ->
```

```
    case analyze_lock(Digit, Data0) of
 {open = StateName, Data} ->
     {next_state, StateName, Data, 10000};
 {StateName, Data} ->
     {next_state, StateName, Data}
    end.
-else.
locked(cast, {button,Digit}, Data0) ->
    case analyze_lock(Digit, Data0) of
 {open = StateName, Data} ->
     {next_state, StateName, Data, 10000};
 {StateName, Data} ->
     {next_state, StateName, Data}
    end;
locked({call, From}, Msg, Data) ->
    handle_call(From, Msg, Data);
locked({info, Msg}, StateName, Data) ->
    handle_info(Msg, StateName, Data).
%% Arity differs
%% All state events are dispatched to handle_call and handle_info help
%% functions. If you want to handle a call or cast event specifically
%% for this state you would add a special clause for it above.
-endif.

-ifdef(BEFORE_REWRITE).
open(timeout, State) ->
     do_lock(),
    {next_state, locked, State};
open({button,_}, Data) ->
    {next_state, locked, Data}.
-else.
open(timeout, _, Data) ->
    do_lock(),
    {next_state, locked, Data};
open(cast, {button,_}, Data) ->
    {next_state, locked, Data};
open({call, From}, Msg, Data) ->
    handle_call(From, Msg, Data);
open(info, Msg, Data) ->
    handle_info(Msg, open, Data).
%% Arity differs
%% All state events are dispatched to handle_call and handle_info help
%% functions. If you want to handle a call or cast event specifically
%% for this state you would add a special clause for it above.
-endif.

-ifdef(BEFORE_REWRITE).
handle_sync_event(stop, _From, _StateName, Data) ->
    {stop, normal, ok, Data}.

handle_event(Event, StateName, Data) ->
    {stop, {shutdown, {unexpected, Event, StateName}}, Data}.

handle_info(Info, StateName, Data) ->
    {stop, {shutdown, {unexpected, Info, StateName}}, StateName, Data}.
-else.
-endif.

terminate(_Reason, State, _Data) ->
    State =/= locked andalso do_lock(),
    ok.
code_change(_Vsn, State, Data, _Extra) ->
    {ok, State, Data}.

%% Internal functions
```

```
-ifdef(BEFORE_REWRITE).
-else.
handle_call(From, stop, Data) ->
     {stop_and_reply, normal,  {reply, From, ok}, Data}.

handle_info(Info, StateName, Data) ->
    {stop, {shutdown, {unexpected, Info, StateName}}, StateName, Data}.
%% These are internal functions for handling all state events
%% and not behaviour callbacks as in gen_fsm
-endif.

analyze_lock(Digit, #{code := Code, remaining := Remaining} = Data) ->
     case Remaining of
         [Digit] ->
      do_unlock(),
      {open,  Data#{remaining := Code}};
         [Digit|Rest] -> % Incomplete
             {locked, Data#{remaining := Rest}};
         _Wrong ->
             {locked, Data#{remaining := Code}}
     end.

do_lock() ->
    io:format("Lock~n", []).
do_unlock() ->
    io:format("Unlock~n", []).
```

gen_server

Erlang module

This behavior module provides the server of a client-server relation. A generic server process (`gen_server`) implemented using this module has a standard set of interface functions and includes functionality for tracing and error reporting. It also fits into an OTP supervision tree. For more information, see section *gen_server Behaviour* in OTP Design Principles.

A `gen_server` process assumes all specific parts to be located in a callback module exporting a predefined set of functions. The relationship between the behavior functions and the callback functions is as follows:

```
gen_server module            Callback module
-----------------            ---------------
gen_server:start
gen_server:start_link -----> Module:init/1

gen_server:stop       -----> Module:terminate/2

gen_server:call
gen_server:multi_call -----> Module:handle_call/3

gen_server:cast
gen_server:abcast     -----> Module:handle_cast/2

-                     -----> Module:handle_info/2

-                     -----> Module:terminate/2

-                     -----> Module:code_change/3
```

If a callback function fails or returns a bad value, the `gen_server` process terminates.

A `gen_server` process handles system messages as described in *`sys(3)`*. The `sys` module can be used for debugging a `gen_server` process.

Notice that a `gen_server` process 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` process does not exist or if bad arguments are specified.

The `gen_server` process can go into hibernation (see *`erlang:hibernate/3`*) if a callback function specifies `'hibernate'` instead of a time-out value. This can be useful if the server is expected to be idle for a long time. However, use this feature with care, as hibernation implies at least two garbage collections (when hibernating and shortly after waking up) and is not something you want to do between each call to a busy server.

Exports

`abcast(Name, Request) -> abcast`

`abcast(Nodes, Name, Request) -> abcast`

Types:

`Nodes = [Node]`

`Node = atom()`

`Name = atom()`

```
Request = term()
```

Sends an asynchronous request to the `gen_server` processes 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_server` processes call *`Module:handle_cast/2`* to handle the request.

For a description of the arguments, see *`multi_call/2,3,4`*.

```
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 `ServerRef` of the `gen_server` process by sending a request and waiting until a reply arrives or a time-out occurs. The `gen_server` process calls *`Module:handle_call/3`* to handle the request.

`ServerRef` can be any of the following:

- The pid
- `Name`, if the `gen_server` process is locally registered
- `{Name,Node}`, if the `gen_server` process is locally registered at another node
- `{global,GlobalName}`, if the `gen_server` process is globally registered
- `{via,Module,ViaName}`, if the `gen_server` process is registered through an alternative process registry

`Request` is any term that is passed as one of the arguments to `Module:handle_call/3`.

`Timeout` is an integer greater than zero that specifies how many milliseconds to wait for a reply, or the atom `infinity` to wait indefinitely. Defaults to 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 can arrive at any time later into the message queue of the caller. 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 can fail for many reasons, including time-out and the called `gen_server` process dying before or during the call.

```
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 `ServerRef` of the `gen_server` process and returns `ok` immediately, ignoring if the destination node or `gen_server` process does not exist. The `gen_server` process calls *`Module:handle_cast/2`* to handle the request.

For a description of `ServerRef`, see *`call/2,3`*.

`Request` is any term that is passed as one of the arguments to `Module:handle_cast/2`.

```
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} | {hibernate_after,HibernateAfterTimeout}
  Dbgs = [Dbg]
   Dbg = trace | log | statistics
    | {log_to_file,FileName} | {install,{Func,FuncState}}
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` process. Does not return, instead the calling process enters the `gen_server` process receive loop and becomes a `gen_server` process. The process **must** have been started using one of the start functions in *`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` process behavior provides.

`Module`, `Options`, and `ServerName` have the same meanings as when calling *`start[_link]/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`*. The callback module `Module` does not need to export an `init/1` function.

The function fails if the calling process was not started by a `proc_lib` start function, or if it is not registered according to `ServerName`.

```
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_server` processes locally registered as `Name` at the specified nodes by first sending a request to every node and then waits for the replies. The `gen_server` process calls *`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 is to be sent. Default value is the list of all known nodes `[node()|nodes()]`.

`Name` is the locally registered name of each `gen_server` process.

`Request` is any term that is passed as one of the arguments to `Module:handle_call/3`.

`Timeout` is an integer greater than zero that specifies how many milliseconds to wait for each reply, or the atom `infinity` to wait indefinitely. Defaults to `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` process 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 cannot process monitors, for example, C or Java nodes, and the `gen_server` process 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 prevent late answers (after the time-out) from polluting the message queue of the caller, a middleman process is used to do the calls. Late answers are then discarded when they arrive to a terminated process.

```
reply(Client, Reply) -> Result
```

Types:

```
Client - see below
Reply = term()
Result = term()
```

This function can be used by a `gen_server` process 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 any term 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 is always to be ignored.

```
start(Module, Args, Options) -> Result
start(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} |
{hibernate_after,HibernateAfterTimeout} | {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 standalone `gen_server` process, that is, a `gen_server` process that is not part of a supervision tree and thus has no supervisor.

For a description of arguments and return values, see *`start_link/3,4`*.

```
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} |
{hibernate_after,HibernateAfterTimeout} | {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. This function is to be called, directly or indirectly, by the supervisor. For example, it ensures that the `gen_server` process is linked to the supervisor.

The `gen_server` process calls *`Module:init/1`* to initialize. To ensure a synchronized startup procedure, `start_link/3,4` does not return until `Module:init/1` has returned.

- If `ServerName={local,Name}`, the `gen_server` process is registered locally as `Name` using `register/2`.

- If `ServerName={global,GlobalName}`, the `gen_server` process id registered globally as `GlobalName` using *`global:register_name/2`* If no name is provided, the `gen_server` process is not registered.
- If `ServerName={via,Module,ViaName}`, the `gen_server` process registers with the registry represented by `Module`. The `Module` callback is to export the functions `register_name/2`, `unregister_name/1`, `whereis_name/1`, and `send/2`, which are to 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 any term that is passed as the argument to *`Module:init/1`*.

- If option `{timeout,Time}` is present, the `gen_server` process is allowed to spend `Time` milliseconds initializing or it is terminated and the start function returns `{error,timeout}`.
- If option `{hibernate_after,HibernateAfterTimeout}` is present, the `gen_server` process awaits any message for `HibernateAfterTimeout` milliseconds and if no message is received, the process goes into hibernation automatically (by calling *`proc_lib:hibernate/3`*).
- If option `{debug,Dbgs}` is present, the corresponding `sys` function is called for each item in `Dbgs`; see *`sys(3)`*.
- If option `{spawn_opt,SOpts}` is present, `SOpts` is passed as option list to the `spawn_opt` BIF, which is used to spawn the `gen_server` process; see *`spawn_opt/2`*.

Note:

Using spawn option `monitor` is not allowed, it causes the function to fail with reason `badarg`.

If the `gen_server` process is successfully created and initialized, the function returns `{ok,Pid}`, where `Pid` is the pid of the `gen_server` process. If a process with the specified `ServerName` exists already, 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.

```
stop(ServerRef) -> ok
stop(ServerRef, Reason, Timeout) -> ok
```

Types:

```
ServerRef = Name | {Name,Node} | {global,GlobalName}
  | {via,Module,ViaName} | pid()
 Node = atom()
 GlobalName = ViaName = term()
Reason = term()
Timeout = int()>0 | infinity
```

Orders a generic server to exit with the specified `Reason` and waits for it to terminate. The `gen_server` process calls *`Module:terminate/2`* before exiting.

The function returns `ok` if the server terminates with the expected reason. Any other reason than `normal`, `shutdown`, or `{shutdown,Term}` causes an error report to be issued using *`error_logger:format/2`*. The default `Reason` is `normal`.

`Timeout` is an integer greater than zero that specifies how many milliseconds to wait for the server to terminate, or the atom `infinity` to wait indefinitely. Defaults to `infinity`. If the server has not terminated within the specified time, a `timeout` exception is raised.

If the process does not exist, a `noproc` exception is raised.

Callback Functions

The following functions are to be exported from a `gen_server` callback module.

Exports

`Module:code_change(OldVsn, State, Extra) -> {ok, NewState} | {error, Reason}`

Types:

```
OldVsn = Vsn | {down, Vsn}
  Vsn = term()
State = NewState = term()
Extra = term()
Reason = term()
```

> **Note:**
>
> This callback is optional, so callback modules need not export it. If a release upgrade/downgrade with `Change={advanced,Extra}` specified in the `appup` file is made when `code_change/3` isn't implemented the process will crash with an `undef` exit reason.

This function is called by a `gen_server` process when it is to update its internal state during a release upgrade/downgrade, that is, when the instruction `{update,Module,Change,...}`, where `Change={advanced,Extra}`, is specifed in the `appup` file. For more information, see section *Release Handling Instructions* in OTP Design Principles.

For an upgrade, `OldVsn` is `Vsn`, and for 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` process.

`Extra` is passed "as is" from the `{advanced,Extra}` part of the update instruction.

If successful, the function must return the updated internal state.

If the function returns `{error,Reason}`, the ongoing upgrade fails and rolls 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 in the following situations:

- One of *`sys:get_status/1,2`* is invoked to get the `gen_server` status. `Opt` is set to the atom `normal`.
- The `gen_server` process terminates abnormally and logs an error. `Opt` is set to the atom `terminate`.

This function is useful for changing the form and appearance of the `gen_server` status for these cases. A callback module wishing to change 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` process.

`PDict` is the current value of the process dictionary of the `gen_server` process..

`State` is the internal state of the `gen_server` process.

The function is to return `Status`, a term that changes the details of the current state and status of the `gen_server` process. 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 is not required, but it makes 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 that large state terms are printed in log files.

`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` process receives a request sent using *`call/2,3`* or *`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` process.

- If `{reply,Reply,NewState}` is returned, `{reply,Reply,NewState,Timeout}` or `{reply,Reply,NewState,hibernate}`, `Reply` is 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` process then continues executing with the possibly updated internal state `NewState`.

For a description of `Timeout` and `hibernate`, see *`Module:init/1`*.

- If `{noreply,NewState}` is returned, `{noreply,NewState,Timeout}`, or `{noreply,NewState,hibernate}`, the `gen_server` process continues executing with `NewState`. Any reply to `From` must be specified explicitly using *`reply/2`*.
- If `{stop,Reason,Reply,NewState}` is returned, `Reply` is given back to `From`.
- If `{stop,Reason,NewState}` is returned, any reply to `From` must be specified explicitly using *`reply/2`*. The `gen_server` process then calls `Module:terminate(Reason,NewState)` and terminates.

`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` process receives a request sent using *`cast/2`* or *`abcast/2,3`*, this function is called to handle the request.

For a description of the arguments and possible return values, see *`Module:handle_call/3`*.

`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()
```

Note:

This callback is optional, so callback modules need not export it. The `gen_server` module provides a default implementation of this function that logs about the unexpected `Info` message, drops it and returns `{noreply, State}`.

This function is called by a `gen_server` process when a time-out 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 time-out has occurred, or the received message.

For a description of the other arguments and possible return values, see *`Module:handle_call/3`*.

`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` process is started using *`start/3,4`* or *`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 is to return `{ok,State}`, `{ok,State,Timeout}`, or `{ok,State,hibernate}`, where `State` is the internal state of the `gen_server` process.

If an integer time-out value is provided, a time-out occurs unless a request or a message is received within `Timeout` milliseconds. A time-out is represented by the atom `timeout`, which is to be handled by the *`Module: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 time-out value, the process goes into hibernation when waiting for the next message to arrive (by calling *`proc_lib:hibernate/3`*).

If the initialization fails, the function is to return `{stop,Reason}`, where `Reason` is any term, or `ignore`.

`Module:terminate(Reason, State)`

Types:

```
Reason = normal | shutdown | {shutdown,term()} | term()
State = term()
```

> **Note:**
>
> This callback is optional, so callback modules need not export it. The `gen_server` module provides a default implementation without cleanup.

This function is called by a `gen_server` process when it is about to terminate. It is to be the opposite of *`Module:init/1`* and do any necessary cleaning up. When it returns, the `gen_server` process 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` process.

`Reason` depends on why the `gen_server` process is terminating. If it is because another callback function has returned a stop tuple `{stop,..}`, `Reason` has the value specified in that tuple. If it is because of a failure, `Reason` is the error reason.

If the `gen_server` process is part of a supervision tree and is ordered by its supervisor to terminate, this function is called with `Reason=shutdown` if the following conditions apply:

- The `gen_server` process has been set to trap exit signals.
- The shutdown strategy as defined in the child specification of the supervisor is an integer time-out value, not `brutal_kill`.

Even if the `gen_server` process is **not** part of a supervision tree, this function is called if it receives an `'EXIT'` message from its parent. `Reason` is the same as in the `'EXIT'` message.

Otherwise, the `gen_server` process terminates immediately.

Notice that for any other reason than `normal`, `shutdown`, or `{shutdown,Term}`, the `gen_server` process is assumed to terminate because of an error and an error report is issued using *`error_logger:format/2`*.

See Also

`gen_event(3)`, *`gen_statem(3)`*, *`proc_lib(3)`*, *`supervisor(3)`*, *`sys(3)`*

gen_statem

Erlang module

This behavior module provides a state machine. Two ***callback modes*** are supported:

- One for finite-state machines (`gen_fsm` like), which requires the state to be an atom and uses that state as the name of the current callback function
- One without restriction on the state data type that uses one callback function for all states

> **Note:**
>
> This is a new behavior in Erlang/OTP 19.0. It has been thoroughly reviewed, is stable enough to be used by at least two heavy OTP applications, and is here to stay. Depending on user feedback, we do not expect but can find it necessary to make minor not backward compatible changes into Erlang/OTP 20.0.

The `gen_statem` behavior replaces `gen_fsm` in Erlang/OTP 20.0. It has the same features and adds some really useful:

- Gathered state code.
- Arbitrary term state.
- Event postponing.
- Self-generated events.
- State time-out.
- Multiple generic named time-outs.
- Absolute time-out time.
- Automatic state enter calls.
- Reply from other state than the request.
- Multiple `sys` traceable replies.

The callback model(s) for `gen_statem` differs from the one for `gen_fsm`, but it is still fairly easy to *rewrite from* `gen_fsm` to `gen_statem`.

A generic state machine process (`gen_statem`) implemented using this module has a standard set of interface functions and includes functionality for tracing and error reporting. It also fits into an OTP supervision tree. For more information, see *OTP Design Principles*.

A `gen_statem` assumes all specific parts to be located in a callback module exporting a predefined set of functions. The relationship between the behavior functions and the callback functions is as follows:

```
gen_statem module            Callback module
-----------------            ---------------
gen_statem:start
gen_statem:start_link -----> Module:init/1

Server start or code change
                      -----> Module:callback_mode/0

gen_statem:stop       -----> Module:terminate/3

gen_statem:call
gen_statem:cast
erlang:send
erlang:'!'            -----> Module:StateName/3
                             Module:handle_event/4

-                     -----> Module:terminate/3

-                     -----> Module:code_change/4
```

Events are of different *types*, so the callback functions can know the origin of an event and how to respond.

If a callback function fails or returns a bad value, the `gen_statem` terminates, unless otherwise stated. However, an exception of class `throw` is not regarded as an error but as a valid return from all callback functions.

The "**state callback**" for a specific *state* in a `gen_statem` is the callback function that is called for all events in this state. It is selected depending on which ***callback mode*** that the callback module defines with the callback function `Module:callback_mode/0`.

When the ***callback mode*** is `state_functions`, the state must be an atom and is used as the state callback name; see `Module:StateName/3`. This gathers all code for a specific state in one function as the `gen_statem` engine branches depending on state name. Note the fact that the callback function `Module:terminate/3` makes the state name `terminate` unusable in this mode.

When the ***callback mode*** is `handle_event_function`, the state can be any term and the state callback name is `Module:handle_event/4`. This makes it easy to branch depending on state or event as you desire. Be careful about which events you handle in which states so that you do not accidentally postpone an event forever creating an infinite busy loop.

The `gen_statem` enqueues incoming events in order of arrival and presents these to the *state callback* in that order. The state callback can postpone an event so it is not retried in the current state. After a state change the queue restarts with the postponed events.

The `gen_statem` event queue model is sufficient to emulate the normal process message queue with selective receive. Postponing an event corresponds to not matching it in a receive statement, and changing states corresponds to entering a new receive statement.

The *state callback* can insert events using the `action()` `next_event` and such an event is inserted as the next to present to the state callback. That is, as if it is the oldest incoming event. A dedicated `event_type()` `internal` can be used for such events making them impossible to mistake for external events.

Inserting an event replaces the trick of calling your own state handling functions that you often would have to resort to in, for example, `gen_fsm` to force processing an inserted event before others.

The `gen_statem` engine can automatically make a specialized call to the *state callback* whenever a new state is entered; see `state_enter()`. This is for writing code common to all state entries. Another way to do it is to insert events at state transitions, but you have to do so everywhere it is needed.

> **Note:**
>
> If you in `gen_statem`, for example, postpone an event in one state and then call another state callback of yours, you have not changed states and hence the postponed event is not retried, which is logical but can be confusing.

For the details of a state transition, see type *`transition_option()`*.

A `gen_statem` handles system messages as described in *`sys`*. The `sys` module can be used for debugging a `gen_statem`.

Notice that a `gen_statem` does not trap exit signals automatically, this must be explicitly initiated in the callback module (by calling *`process_flag(trap_exit, true)`*.

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

The `gen_statem` process can go into hibernation; see *`proc_lib:hibernate/3`*. It is done when a *state callback* or *`Module:init/1`* specifies `hibernate` in the returned *`Actions`* list. This feature can be useful to reclaim process heap memory while the server is expected to be idle for a long time. However, use this feature with care, as hibernation can be too costly to use after every event; see *`erlang:hibernate/3`*.

Example

The following example shows a simple pushbutton model for a toggling pushbutton implemented with ***callback mode*** `state_functions`. You can push the button and it replies if it went on or off, and you can ask for a count of how many times it has been pushed to switch on.

The following is the complete callback module file `pushbutton.erl`:

```
-module(pushbutton).
-behaviour(gen_statem).

-export([start/0,push/0,get_count/0,stop/0]).
-export([terminate/3,code_change/4,init/1,callback_mode/0]).
-export([on/3,off/3]).

name() -> pushbutton_statem. % The registered server name

%% API.  This example uses a registered name name()
%% and does not link to the caller.
start() ->
    gen_statem:start({local,name()}, ?MODULE, [], []).
push() ->
    gen_statem:call(name(), push).
get_count() ->
    gen_statem:call(name(), get_count).
stop() ->
    gen_statem:stop(name()).

%% Mandatory callback functions
terminate(_Reason, _State, _Data) ->
    void.
code_change(_Vsn, State, Data, _Extra) ->
    {ok,State,Data}.
init([]) ->
    %% Set the initial state + data.  Data is used only as a counter.
    State = off, Data = 0,
    {ok,State,Data}.
callback_mode() -> state_functions.

%%% state callback(s)

off({call,From}, push, Data) ->
    %% Go to 'on', increment count and reply
    %% that the resulting status is 'on'
    {next_state,on,Data+1,[{reply,From,on}]};
off(EventType, EventContent, Data) ->
    handle_event(EventType, EventContent, Data).

on({call,From}, push, Data) ->
    %% Go to 'off' and reply that the resulting status is 'off'
    {next_state,off,Data,[{reply,From,off}]};
on(EventType, EventContent, Data) ->
    handle_event(EventType, EventContent, Data).

%% Handle events common to all states
handle_event({call,From}, get_count, Data) ->
    %% Reply with the current count
    {keep_state,Data,[{reply,From,Data}]};
handle_event(_, _, Data) ->
    %% Ignore all other events
    {keep_state,Data}.
```

The following is a shell session when running it:

```
1> pushbutton:start().
{ok,<0.36.0>}
2> pushbutton:get_count().
0
3> pushbutton:push().
on
4> pushbutton:get_count().
1
5> pushbutton:push().
off
6> pushbutton:get_count().
1
7> pushbutton:stop().
ok
8> pushbutton:push().
** exception exit: {noproc,{gen_statem,call,[pushbutton_statem,push,infinity]}}
     in function  gen:do_for_proc/2 (gen.erl, line 261)
     in call from gen_statem:call/3 (gen_statem.erl, line 386)
```

To compare styles, here follows the same example using ***callback mode*** `handle_event_function`, or rather the code to replace after function `init/1` of the `pushbutton.erl` example file above:

```
callback_mode() -> handle_event_function.

%%% state callback(s)

handle_event({call,From}, push, off, Data) ->
    %% Go to 'on', increment count and reply
    %% that the resulting status is 'on'
    {next_state,on,Data+1,[{reply,From,on}]};
handle_event({call,From}, push, on, Data) ->
    %% Go to 'off' and reply that the resulting status is 'off'
    {next_state,off,Data,[{reply,From,off}]};
%%
%% Event handling common to all states
handle_event({call,From}, get_count, State, Data) ->
    %% Reply with the current count
    {next_state,State,Data,[{reply,From,Data}]};
handle_event(_, _, State, Data) ->
    %% Ignore all other events
    {next_state,State,Data}.
```

Data Types

```
server_name() =
    {global, GlobalName :: term()} |
    {via, RegMod :: module(), Name :: term()} |
    {local, atom()}
```

Name specification to use when starting a `gen_statem` server. See *`start_link/3`* and *`server_ref()`* below.

```
server_ref() =
    pid() |
    (LocalName :: atom()) |
    {Name :: atom(), Node :: atom()} |
    {global, GlobalName :: term()} |
    {via, RegMod :: module(), ViaName :: term()}
```

Server specification to use when addressing a `gen_statem` server. See *`call/2`* and *`server_name()`* above.

It can be:

`pid() | LocalName`

The `gen_statem` is locally registered.

`{Name,Node}`

The `gen_statem` is locally registered on another node.

`{global,GlobalName}`

The `gen_statem` is globally registered in *`global`*.

`{via,RegMod,ViaName}`

The `gen_statem` is registered in an alternative process registry. The registry callback module `RegMod` is to export functions `register_name/2`, `unregister_name/1`, `whereis_name/1`, and `send/2`, which are to behave like the corresponding functions in *`global`*. Thus, `{via,global,GlobalName}` is the same as `{global,GlobalName}`.

```
debug_opt() =
    {debug,
     Dbgs ::
         [trace | log | statistics | debug | {logfile, string()}]}
```

Debug option that can be used when starting a `gen_statem` server through, *`enter_loop/4-6`*.

For every entry in `Dbgs`, the corresponding function in *`sys`* is called.

```
hibernate_after_opt() =
    {hibernate_after, HibernateAfterTimeout :: timeout()}
```

hibernate_after option that can be used when starting a `gen_statem` server through, *`enter_loop/4-6`*.

If option *`{hibernate_after,HibernateAfterTimeout}`* is present, the `gen_statem` process awaits any message for `HibernateAfterTimeout` milliseconds and if no message is received, the process goes into hibernation automatically (by calling *`proc_lib:hibernate/3`*).

```
start_opt() =
    debug_opt() |
    {timeout, Time :: timeout()} |
    hibernate_after_opt() |
    {spawn_opt, [proc_lib:spawn_option()]}
```

Options that can be used when starting a `gen_statem` server through, for example, *`start_link/3`*.

```
start_ret() = {ok, pid()} | ignore | {error, term()}
```

Return value from the start functions, for example, *`start_link/3`*.

```
from() = {To :: pid(), Tag :: term()}
```

Destination to use when replying through, for example, the *`action()`* `{reply,From,Reply}` to a process that has called the `gen_statem` server using *`call/2`*.

```
state() = state_name() | term()
```

If the ***callback mode*** is `handle_event_function`, the state can be any term. After a state change (`NextState =/= State`), all postponed events are retried.

```
state_name() = atom()
```

If the ***callback mode*** is `state_functions`, the state must be of this type. After a state change (`NextState =/= State`), all postponed events are retried.

data() = term()

A term in which the state machine implementation is to store any server data it needs. The difference between this and the *state()* itself is that a change in this data does not cause postponed events to be retried. Hence, if a change in this data would change the set of events that are handled, then that data item is to be made a part of the state.

event_type() =
**   *external_event_type()* | *timeout_event_type()* | internal**

There are 3 categories of events: *external*, *timeout*, and `internal`.

`internal` events can only be generated by the state machine itself through the state transition action *next_event*.

external_event_type() = {call, From :: *from()*} | cast | info

External events are of 3 types: `{call,From}`, `cast`, or `info`. *Calls* (synchronous) and *casts* originate from the corresponding API functions. For calls, the event contains whom to reply to. Type `info` originates from regular process messages sent to the `gen_statem`.

timeout_event_type() =
**   timeout | {timeout, Name :: term()} | state_timeout**

There are 3 types of timeout events that the state machine can generate for itself with the corresponding *timeout_action()*s.

callback_mode_result() =
**   *callback_mode()* | [*callback_mode()* | *state_enter()*]**

This is the return type from *Module:callback_mode/0* and selects *callback mode* and whether to do *state enter calls*, or not.

callback_mode() = state_functions | handle_event_function

The **callback mode** is selected when starting the `gen_statem` and after code change using the return value from *Module:callback_mode/0*.

`state_functions`

The state must be of type *state_name()* and one callback function per state, that is, *Module:StateName/3*, is used.

`handle_event_function`

The state can be any term and the callback function *Module:handle_event/4* is used for all states.

state_enter() = state_enter

Whether the state machine should use **state enter calls** or not is selected when starting the `gen_statem` and after code change using the return value from *Module:callback_mode/0*.

If *Module:callback_mode/0* returns a list containing `state_enter`, the `gen_statem` engine will, at every state change, call the *state callback* with arguments `(enter, OldState, Data)`. This may look like an event but is really a call performed after the previous state callback returned and before any event is delivered to the new state callback. See *Module:StateName/3* and *Module:handle_event/4*. Such a call can be repeated by returning a *repeat_state* or *repeat_state_and_data* tuple from the state callback.

If *Module:callback_mode/0* does not return such a list, no state enter calls are done.

If *Module:code_change/4* should transform the state to a state with a different name it is still regarded as the same state so this does not cause a state enter call.

Note that a state enter call **will** be done right before entering the initial state even though this formally is not a state change. In this case `OldState` will be the same as `State`, which can not happen for a subsequent state change, but will happen when repeating the state enter call.

```
transition_option() =
    postpone() |
    hibernate() |
    event_timeout() |
    generic_timeout() |
    state_timeout()
```

Transition options can be set by *actions* and they modify how the state transition is done:

- If the state changes, is the initial state, `repeat_state` or `repeat_state_and_data` is used, and also ***state enter calls*** are used, the `gen_statem` calls the new state callback with arguments *(enter, OldState, Data)*. Any `actions` returned from this call are handled as if they were appended to the actions returned by the state callback that changed states.
- All *actions* are processed in order of appearance.
- If `postpone()` is `true`, the current event is postponed.
- If the state changes, the queue of incoming events is reset to start with the oldest postponed.
- All events stored with `action()` `next_event` are inserted to be processed before the other queued events.
- Time-out timers `event_timeout()`, `generic_timeout()` and `state_timeout()` are handled. Time-outs with zero time are guaranteed to be delivered to the state machine before any external not yet received event so if there is such a time-out requested, the corresponding time-out zero event is enqueued as the newest event.

  Any event cancels an `event_timeout()` so a zero time event time-out is only generated if the event queue is empty.

  A state change cancels a `state_timeout()` and any new transition option of this type belongs to the new state.
- If there are enqueued events the *state callback* for the possibly new state is called with the oldest enqueued event, and we start again from the top of this list.
- Otherwise the `gen_statem` goes into `receive` or hibernation (if `hibernate()` is `true`) to wait for the next message. In hibernation the next non-system event awakens the `gen_statem`, or rather the next incoming message awakens the `gen_statem`, but if it is a system event it goes right back into hibernation. When a new message arrives the *state callback* is called with the corresponding event, and we start again from the top of this list.

`postpone() = boolean()`

If `true`, postpones the current event and retries it when the state changes (`NextState =/= State`).

`hibernate() = boolean()`

If `true`, hibernates the `gen_statem` by calling `proc_lib:hibernate/3` before going into `receive` to wait for a new external event. If there are enqueued events, to prevent receiving any new event, an `erlang:garbage_collect/0` is done instead to simulate that the `gen_statem` entered hibernation and immediately got awakened by the oldest enqueued event.

`event_timeout() = timeout() | integer()`

Starts a timer set by `enter_action()` `timeout`. When the timer expires an event of `event_type()` `timeout` will be generated. See `erlang:start_timer/4` for how `Time` and `Options` are interpreted. Future `erlang:start_timer/4` `Options` will not necessarily be supported.

Any event that arrives cancels this time-out. Note that a retried or inserted event counts as arrived. So does a state time-out zero event, if it was generated before this time-out is requested.

If `Time` is `infinity`, no timer is started, as it never would expire anyway.

If `Time` is relative and `0` no timer is actually started, instead the the time-out event is enqueued to ensure that it gets processed before any not yet received external event.

Note that it is not possible nor needed to cancel this time-out, as it is cancelled automatically by any other event.

`generic_timeout() = timeout() | integer()`

Starts a timer set by *`enter_action()`* `{timeout,Name}`. When the timer expires an event of *`event_type()`* `{timeout,Name}` will be generated. See *`erlang:start_timer/4`* for how `Time` and *`Options`* are interpreted. Future `erlang:start_timer/4` `Options` will not necessarily be supported.

If `Time` is `infinity`, no timer is started, as it never would expire anyway.

If `Time` is relative and `0` no timer is actually started, instead the the time-out event is enqueued to ensure that it gets processed before any not yet received external event.

Setting a timer with the same `Name` while it is running will restart it with the new time-out value. Therefore it is possible to cancel a specific time-out by setting it to `infinity`.

`state_timeout() = timeout() | integer()`

Starts a timer set by *`enter_action()`* `state_timeout`. When the timer expires an event of *`event_type()`* `state_timeout` will be generated. See *`erlang:start_timer/4`* for how `Time` and *`Options`* are interpreted. Future `erlang:start_timer/4` `Options` will not necessarily be supported.

If `Time` is `infinity`, no timer is started, as it never would expire anyway.

If `Time` is relative and `0` no timer is actually started, instead the the time-out event is enqueued to ensure that it gets processed before any not yet received external event.

Setting this timer while it is running will restart it with the new time-out value. Therefore it is possible to cancel this time-out by setting it to `infinity`.

`timeout_option() = {abs, Abs :: boolean()}`

If `Abs` is `true` an absolute timer is started, and if it is `false` a relative, which is the default. See *`erlang:start_timer/4`* for details.

```
action() =
    postpone |
    {postpone, Postpone :: postpone()} |
    {next_event,
     EventType :: event_type(),
     EventContent :: term()} |
    enter_action()
```

These state transition actions can be invoked by returning them from the *state callback* when it is called with an *event*, from *`Module:init/1`* or by giving them to *`enter_loop/5,6`*.

Actions are executed in the containing list order.

Actions that set *transition options* override any previous of the same type, so the last in the containing list wins. For example, the last *`postpone()`* overrides any previous `postpone()` in the list.

`postpone`

Sets the *`transition_option()`* *`postpone()`* for this state transition. This action is ignored when returned from *`Module:init/1`* or given to *`enter_loop/5,6`*, as there is no event to postpone in those cases.

`next_event`

Stores the specified `EventType` and `EventContent` for insertion after all actions have been executed.

The stored events are inserted in the queue as the next to process before any already queued events. The order of these stored events is preserved, so the first `next_event` in the containing list becomes the first to process.

An event of type *`internal`* is to be used when you want to reliably distinguish an event inserted this way from any external event.

```
enter_action() =
    hibernate |
    {hibernate, Hibernate :: hibernate()} |
    timeout_action() |
    reply_action()
```

These state transition actions can be invoked by returning them from the *state callback*, from `Module:init/1` or by giving them to `enter_loop/5,6`.

Actions are executed in the containing list order.

Actions that set *transition options* override any previous of the same type, so the last in the containing list wins. For example, the last `event_timeout()` overrides any previous `event_timeout()` in the list.

`hibernate`

Sets the `transition_option()` `hibernate()` for this state transition.

```
timeout_action() =
    (Timeout :: event_timeout()) |
    {timeout, Time :: event_timeout(), EventContent :: term()} |
    {timeout,
     Time :: event_timeout(),
     EventContent :: term(),
     Options :: timeout_option() | [timeout_option()]} |
    {{timeout, Name :: term()},
     Time :: generic_timeout(),
     EventContent :: term()} |
    {{timeout, Name :: term()},
     Time :: generic_timeout(),
     EventContent :: term(),
     Options :: timeout_option() | [timeout_option()]} |
    {state_timeout,
     Time :: state_timeout(),
     EventContent :: term()} |
    {state_timeout,
     Time :: state_timeout(),
     EventContent :: term(),
     Options :: timeout_option() | [timeout_option()]}
```

These state transition actions can be invoked by returning them from the *state callback*, from `Module:init/1` or by giving them to `enter_loop/5,6`.

These timeout actions sets timeout *transition options*.

`Timeout`

Short for `{timeout,Timeout,Timeout}`, that is, the time-out message is the time-out time. This form exists to make the *state callback* return value `{next_state,NextState,NewData,Timeout}` allowed like for `gen_fsm`'s

`timeout`

Sets the `transition_option()` `event_timeout()` to `Time` with `EventContent` and time-out options `Options`.

`{timeout,Name}`

Sets the `transition_option()` `generic_timeout()` to `Time` for `Name` with `EventContent` and time-out options `Options`.

`state_timeout`

Sets the *`transition_option() state_timeout()`* to `Time` with `EventContent` and time-out options *`Options`*.

`reply_action() = {reply, From :: from(), Reply :: term()}`

This state transition action can be invoked by returning it from the *state callback*, from *`Module:init/1`* or by giving it to *`enter_loop/5,6`*.

It replies to a caller waiting for a reply in *`call/2`*. `From` must be the term from argument *`{call,From}`* in a call to a *state callback*.

Note that using this action from *`Module:init/1`* or *`enter_loop/5,6`* would be weird on the border of witchcraft since there has been no earlier call to a *state callback* in this server.

```
init_result(StateType) =
    {ok, State :: StateType, Data :: data()} |
    {ok,
     State :: StateType,
     Data :: data(),
     Actions :: [action()] | action()} |
    ignore |
    {stop, Reason :: term()}
```

For a succesful initialization, `State` is the initial *`state()`* and `Data` the initial server *`data()`* of the `gen_statem`.

The *`Actions`* are executed when entering the first *state* just as for a *state callback*, except that the action `postpone` is forced to `false` since there is no event to postpone.

For an unsuccesful initialization, `{stop,Reason}` or `ignore` should be used; see *`start_link/3,4`*.

```
state_enter_result(State) =
    {next_state, State, NewData :: data()} |
    {next_state,
     State,
     NewData :: data(),
     Actions :: [enter_action()] | enter_action()} |
    state_callback_result(enter_action())
```

`State` is the current state and it can not be changed since the state callback was called with a ***state enter call***.

`next_state`

The `gen_statem` does a state transition to `State`, which has to be the current state, sets `NewData`, and executes all `Actions`.

```
event_handler_result(StateType) =
    {next_state, NextState :: StateType, NewData :: data()} |
    {next_state,
     NextState :: StateType,
     NewData :: data(),
     Actions :: [action()] | action()} |
    state_callback_result(action())
```

`StateType` is *`state_name()`* if ***callback mode*** is `state_functions`, or *`state()`* if ***callback mode*** is `handle_event_function`.

`next_state`

The `gen_statem` does a state transition to `NextState` (which can be the same as the current state), sets `NewData`, and executes all `Actions`.

```
state_callback_result(ActionType) =
    {keep_state, NewData :: data()} |
    {keep_state,
     NewData :: data(),
     Actions :: [ActionType] | ActionType} |
    keep_state_and_data |
    {keep_state_and_data, Actions :: [ActionType] | ActionType} |
    {repeat_state, NewData :: data()} |
    {repeat_state,
     NewData :: data(),
     Actions :: [ActionType] | ActionType} |
    repeat_state_and_data |
    {repeat_state_and_data, Actions :: [ActionType] | ActionType} |
    stop |
    {stop, Reason :: term()} |
    {stop, Reason :: term(), NewData :: data()} |
    {stop_and_reply,
     Reason :: term(),
     Replies :: [reply_action()] | reply_action()} |
    {stop_and_reply,
     Reason :: term(),
     Replies :: [reply_action()] | reply_action(),
     NewData :: data()}
```

`ActionType` is *`enter_action()`* if the state callback was called with a ***state enter call*** and *`action()`* if the state callback was called with an event.

`keep_state`

The `gen_statem` keeps the current state, or does a state transition to the current state if you like, sets `NewData`, and executes all `Actions`. This is the same as `{next_state,CurrentState,NewData,Actions}`.

`keep_state_and_data`

The `gen_statem` keeps the current state or does a state transition to the current state if you like, keeps the current server data, and executes all `Actions`. This is the same as `{next_state,CurrentState,CurrentData,Actions}`.

`repeat_state`

The `gen_statem` keeps the current state, or does a state transition to the current state if you like, sets `NewData`, and executes all `Actions`. If the `gen_statem` runs with ***state enter calls***, the state enter call is repeated, see type *`transition_option()`*, otherwise `repeat_state` is the same as `keep_state`.

`repeat_state_and_data`

The `gen_statem` keeps the current state and data, or does a state transition to the current state if you like, and executes all `Actions`. This is the same as `{repeat_state,CurrentData,Actions}`. If the `gen_statem` runs with ***state enter calls***, the state enter call is repeated, see type *`transition_option()`*, otherwise `repeat_state_and_data` is the same as `keep_state_and_data`.

`stop`

Terminates the `gen_statem` by calling *`Module:terminate/3`* with `Reason` and `NewData`, if specified.

`stop_and_reply`

Sends all `Replies`, then terminates the `gen_statem` by calling *`Module:terminate/3`* with `Reason` and `NewData`, if specified.

All these terms are tuples or atoms and this property will hold in any future version of `gen_statem`.

Exports

```
call(ServerRef :: server_ref(), Request :: term()) ->
        Reply :: term()
call(ServerRef :: server_ref(),
     Request :: term(),
     Timeout ::
         timeout() |
         {clean_timeout, T :: timeout()} |
         {dirty_timeout, T :: timeout()}) ->
        Reply :: term()
```

Makes a synchronous call to the `gen_statem` *`ServerRef`* by sending a request and waiting until its reply arrives. The `gen_statem` calls the *state callback* with *`event_type()`* `{call,From}` and event content `Request`.

A `Reply` is generated when a *state callback* returns with `{reply,From,Reply}` as one *`action()`*, and that `Reply` becomes the return value of this function.

`Timeout` is an integer > 0, which specifies how many milliseconds to wait for a reply, or the atom `infinity` to wait indefinitely, which is the default. If no reply is received within the specified time, the function call fails.

> **Note:**
>
> For `Timeout < infinity`, to avoid getting a late reply in the caller's inbox if the caller should catch exceptions, this function spawns a proxy process that does the call. A late reply gets delivered to the dead proxy process, hence gets discarded. This is less efficient than using `Timeout == infinity`.

`Timeout` can also be a tuple `{clean_timeout,T}` or `{dirty_timeout,T}`, where `T` is the time-out time. `{clean_timeout,T}` works like just `T` described in the note above and uses a proxy process for `T < infinity`, while `{dirty_timeout,T}` bypasses the proxy process which is more lightweight.

> **Note:**
>
> If you combine catching exceptions from this function with `{dirty_timeout,T}` to avoid that the calling process dies when the call times out, you will have to be prepared to handle a late reply. So why not just let the calling process die?

The call can also fail, for example, if the `gen_statem` dies before or during this function call.

```
cast(ServerRef :: server_ref(), Msg :: term()) -> ok
```

Sends an asynchronous event to the `gen_statem` *`ServerRef`* and returns `ok` immediately, ignoring if the destination node or `gen_statem` does not exist. The `gen_statem` calls the *state callback* with *`event_type()`* `cast` and event content `Msg`.

```
enter_loop(Module :: module(),
           Opts :: [debug_opt() | hibernate_after_opt()],
```

```
          State :: state(),
          Data :: data()) ->
             no_return()
```

The same as *enter_loop/6* with `Actions = []` except that no *server_name()* must have been registered. This creates an anonymous server.

```
enter_loop(Module :: module(),
          Opts :: [debug_opt() | hibernate_after_opt()],
          State :: state(),
          Data :: data(),
          Server_or_Actions :: server_name() | pid() | [action()]) ->
             no_return()
```

If `Server_or_Actions` is a `list()`, the same as *enter_loop/6* except that no *server_name()* must have been registered and `Actions = Server_or_Actions`. This creates an anonymous server.

Otherwise the same as *enter_loop/6* with `Server = Server_or_Actions` and `Actions = []`.

```
enter_loop(Module :: module(),
          Opts :: [debug_opt() | hibernate_after_opt()],
          State :: state(),
          Data :: data(),
          Server :: server_name() | pid(),
          Actions :: [action()] | action()) ->
             no_return()
```

Makes the calling process become a `gen_statem`. Does not return, instead the calling process enters the `gen_statem` receive loop and becomes a `gen_statem` server. The process **must** have been started using one of the start functions in *proc_lib*. 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_statem` behavior provides.

`Module`, `Opts` have the same meaning as when calling *start[_link]/3,4*.

If `Server` is `self()` an anonymous server is created just as when using *start[_link]/3*. If `Server` is a *server_name()* a named server is created just as when using *start[_link]/4*. However, the *server_name()* name must have been registered accordingly **before** this function is called.

`State`, `Data`, and `Actions` have the same meanings as in the return value of *Module:init/1*. Also, the callback module does not need to export a *Module:init/1* function.

The function fails if the calling process was not started by a *proc_lib* start function, or if it is not registered according to *server_name()*.

```
reply(Replies :: [reply_action()] | reply_action()) -> ok
reply(From :: from(), Reply :: term()) -> ok
```

This function can be used by a `gen_statem` to explicitly send a reply to a process that waits in *call/2* when the reply cannot be defined in the return value of a *state callback*.

`From` must be the term from argument *{call,From}* to the *state callback*. A reply or multiple replies canalso be sent using one or several *reply_action()*s from a *state callback*.

> **Note:**
>
> A reply sent with this function is not visible in *sys* debug output.

```
start(Module :: module(), Args :: term(), Opts :: [start_opt()]) ->
         start_ret()
start(ServerName :: server_name(),
      Module :: module(),
      Args :: term(),
      Opts :: [start_opt()]) ->
         start_ret()
```

Creates a standalone gen_statem process according to OTP design principles (using *proc_lib* primitives). As it does not get linked to the calling process, this start function cannot be used by a supervisor to start a child.

For a description of arguments and return values, see *start_link/3,4*.

```
start_link(Module :: module(),
           Args :: term(),
           Opts :: [start_opt()]) ->
              start_ret()
start_link(ServerName :: server_name(),
           Module :: module(),
           Args :: term(),
           Opts :: [start_opt()]) ->
              start_ret()
```

Creates a gen_statem process according to OTP design principles (using *proc_lib* primitives) that is linked to the calling process. This is essential when the gen_statem must be part of a supervision tree so it gets linked to its supervisor.

The gen_statem process calls *Module:init/1* to initialize the server. To ensure a synchronized startup procedure, start_link/3,4 does not return until *Module:init/1* has returned.

ServerName specifies the *server_name()* to register for the gen_statem. If the gen_statem is started with start_link/3, no ServerName is provided and the gen_statem is not registered.

Module is the name of the callback module.

Args is an arbitrary term that is passed as the argument to *Module:init/1*.

- If option {timeout,Time} is present in Opts, the gen_statem is allowed to spend Time milliseconds initializing or it terminates and the start function returns *{error,timeout}*.
- If option*{hibernate_after,HibernateAfterTimeout}* is present, the gen_statem process awaits any message for HibernateAfterTimeout milliseconds and if no message is received, the process goes into hibernation automatically (by calling *proc_lib:hibernate/3*).
- If option *{debug,Dbgs}* is present in Opts, debugging through *sys* is activated.
- If option {spawn_opt,SpawnOpts} is present in Opts, SpawnOpts is passed as option list to *erlang:spawn_opt/2*, which is used to spawn the gen_statem process.

> **Note:**
>
> Using spawn option monitor is not allowed, it causes this function to fail with reason badarg.

If the `gen_statem` is successfully created and initialized, this function returns *`{ok,Pid}`*, where `Pid` is the `pid()` of the `gen_statem`. If a process with the specified `ServerName` exists already, this function returns *`{error,{already_started,Pid}}`*, where `Pid` is the `pid()` of that process.

If `Module:init/1` fails with `Reason`, this function returns *`{error,Reason}`*. If `Module:init/1` returns *`{stop,Reason}`* or *`ignore`*, the process is terminated and this function returns *`{error,Reason}`* or *`ignore`*, respectively.

`stop(ServerRef :: server_ref()) -> ok`

The same as *`stop(ServerRef, normal, infinity)`*.

```
stop(ServerRef :: server_ref(),
     Reason :: term(),
     Timeout :: timeout()) ->
        ok
```

Orders the `gen_statem` *`ServerRef`* to exit with the specified `Reason` and waits for it to terminate. The `gen_statem` calls *`Module:terminate/3`* before exiting.

This function returns `ok` if the server terminates with the expected reason. Any other reason than `normal`, `shutdown`, or `{shutdown,Term}` causes an error report to be issued through *`error_logger:format/2`*. The default `Reason` is `normal`.

`Timeout` is an integer > 0, which specifies how many milliseconds to wait for the server to terminate, or the atom `infinity` to wait indefinitely. Defaults to `infinity`. If the server does not terminate within the specified time, a `timeout` exception is raised.

If the process does not exist, a `noproc` exception is raised.

Callback Functions

The following functions are to be exported from a `gen_statem` callback module.

Exports

`Module:callback_mode() -> CallbackMode`

Types:

`CallbackMode = callback_mode() | [ callback_mode() | state_enter() ]`

This function is called by a `gen_statem` when it needs to find out the ***callback mode*** of the callback module. The value is cached by `gen_statem` for efficiency reasons, so this function is only called once after server start and after code change, but before the first *state callback* in the current code version is called. More occasions may be added in future versions of `gen_statem`.

Server start happens either when *`Module:init/1`* returns or when *`enter_loop/4-6`* is called. Code change happens when *`Module:code_change/4`* returns.

The `CallbackMode` is either just *`callback_mode()`* or a list containing *`callback_mode()`* and possibly the atom *`state_enter`*.

Note:

If this function's body does not return an inline constant value the callback module is doing something strange.

`Module:code_change(OldVsn, OldState, OldData, Extra) -> Result`

Types:

`OldVsn = Vsn | {down,Vsn}`
`Vsn = term()`
`OldState = NewState = term()`
`Extra = term()`
`Result = {ok,NewState,NewData} | Reason`
`OldState = NewState = state()`
`OldData = NewData = data()`
`Reason = term()`

> **Note:**
>
> This callback is optional, so callback modules need not export it. If a release upgrade/downgrade with `Change={advanced,Extra}` specified in the `.appup` file is made when `code_change/4` is not implemented the process will crash with exit reason `undef`.

This function is called by a `gen_statem` when it is to update its internal state during a release upgrade/downgrade, that is, when the instruction `{update,Module,Change,...}`, where `Change={advanced,Extra}`, is specified in the *appup* file. For more information, see *OTP Design Principles*.

For an upgrade, `OldVsn` is `Vsn`, and for 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.

`OldState` and `OldData` is the internal state of the `gen_statem`.

`Extra` is passed "as is" from the `{advanced,Extra}` part of the update instruction.

If successful, the function must return the updated internal state in an `{ok,NewState,NewData}` tuple.

If the function returns a failure `Reason`, the ongoing upgrade fails and rolls back to the old release. Note that `Reason` can not be an `{ok,_,_}` tuple since that will be regarded as a `{ok,NewState,NewData}` tuple, and that a tuple matching `{ok,_}` is an also invalid failure `Reason`. It is recommended to use an atom as `Reason` since it will be wrapped in an `{error,Reason}` tuple.

Also note when upgrading a `gen_statem`, this function and hence the `Change={advanced,Extra}` parameter in the *appup* file is not only needed to update the internal state or to act on the `Extra` argument. It is also needed if an upgrade or downgrade should change ***callback mode***, or else the callback mode after the code change will not be honoured, most probably causing a server crash.

`Module:init(Args) -> Result(StateType)`

Types:

`Args = term()`
`Result(StateType) = init_result(StateType)`

Whenever a `gen_statem` is started using *`start_link/3,4`* or *`start/3,4`*, this function is called by the new process to initialize the implementation state and server data.

`Args` is the `Args` argument provided to that start function.

Note:

Note that if the `gen_statem` is started trough *`proc_lib`* and *`enter_loop/4-6`*, this callback will never be called. Since this callback is not optional it can in that case be implemented as:

```
init(Args) -> erlang:error(not_implemented, [Args]).
```

`Module:format_status(Opt, [PDict,State,Data]) -> Status`

Types:

`Opt = normal | terminate`
`PDict = [{Key, Value}]`
`State = state()`
`Data = data()`
`Key = term()`
`Value = term()`
`Status = term()`

Note:

This callback is optional, so a callback module does not need to export it. The `gen_statem` module provides a default implementation of this function that returns `{State,Data}`.

If this callback is exported but fails, to hide possibly sensitive data, the default function will instead return `{State,Info}`, where `Info` says nothing but the fact that `format_status/2` has crashed.

This function is called by a `gen_statem` process when any of the following apply:

- One of *`sys:get_status/1,2`* is invoked to get the `gen_statem` status. `Opt` is set to the atom `normal` for this case.
- The `gen_statem` terminates abnormally and logs an error. `Opt` is set to the atom `terminate` for this case.

This function is useful for changing the form and appearance of the `gen_statem` status for these cases. A callback module wishing to change the *`sys:get_status/1,2`* return value and how its status appears in termination error logs exports an instance of `format_status/2`, which returns a term describing the current status of the `gen_statem`.

`PDict` is the current value of the process dictionary of the `gen_statem`.

`State` is the internal state of the `gen_statem`.

`Data` is the internal server data of the `gen_statem`.

The function is to return `Status`, a term that contains the appropriate details of the current state and status of the `gen_statem`. 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_statem` state. Following this recommendation is not required, but it makes 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 log files. Another use is to hide sensitive data from being written to the error log.

```
Module:StateName(enter, OldState, Data) -> StateEnterResult(StateName)
Module:StateName(EventType, EventContent, Data) -> StateFunctionResult
Module:handle_event(enter, OldState, State, Data) -> StateEnterResult(State)
Module:handle_event(EventType, EventContent, State, Data) ->
HandleEventResult
```

Types:

```
EventType = event_type()
EventContent = term()
State = state()
Data = NewData = data()
StateEnterResult(StateName) = state_enter_result(StateName)
StateFunctionResult = event_handler_result(state_name())
StateEnterResult(State) = state_enter_result(State)
HandleEventResult = event_handler_result(state())
```

Whenever a `gen_statem` receives an event from *`call/2`*, *`cast/2`*, or as a normal process message, one of these functions is called. If ***callback mode*** is `state_functions`, `Module:StateName/3` is called, and if it is `handle_event_function`, `Module:handle_event/4` is called.

If `EventType` is *`{call,From}`*, the caller waits for a reply. The reply can be sent from this or from any other *state callback* by returning with `{reply,From,Reply}` in *`Actions`*, in *`Replies`*, or by calling *`reply(From, Reply)`*.

If this function returns with a next state that does not match equal (`=/=`) to the current state, all postponed events are retried in the next state.

The only difference between `StateFunctionResult` and `HandleEventResult` is that for `StateFunctionResult` the next state must be an atom, but for `HandleEventResult` there is no restriction on the next state.

For options that can be set and actions that can be done by `gen_statem` after returning from this function, see *`action()`*.

When the `gen_statem` runs with *state enter calls*, these functions are also called with arguments `(enter, OldState, ...)` whenever the state changes. In this case there are some restrictions on the *actions* that may be returned: *`postpone()`* is not allowed since a **state enter call** is not an event so there is no event to postpone, and *`{next_event,_,_}`* is not allowed since using **state enter calls** should not affect how events are consumed and produced. You may also not change states from this call. Should you return `{next_state,NextState, ...}` with `NextState =/= State` the `gen_statem` crashes. It is possible to use `{repeat_state, ...}`, `{repeat_state_and_data,_}` or `repeat_state_and_data` but all of them makes little sense since you immediately will be called again with a new **state enter call** making this just a weird way of looping, and there are better ways to loop in Erlang. You are advised to use `{keep_state,...}`, `{keep_state_and_data,_}` or `keep_state_and_data` since you can not change states from a **state enter call** anyway.

Note the fact that you can use *`throw`* to return the result, which can be useful. For example to bail out with `throw(keep_state_and_data)` from deep within complex code that can not return `{next_state,State,Data}` because `State` or `Data` is no longer in scope.

```
Module:terminate(Reason, State, Data) -> Ignored
```

Types:

```
Reason = normal | shutdown | {shutdown,term()} | term()
State = state()
```

```
Data = data()
Ignored = term()
```

Note:

This callback is optional, so callback modules need not export it. The `gen_statem` module provides a default implementation without cleanup.

This function is called by a `gen_statem` when it is about to terminate. It is to be the opposite of *`Module:init/1`* and do any necessary cleaning up. When it returns, the `gen_statem` 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_statem`.

`Reason` depends on why the `gen_statem` is terminating. If it is because another callback function has returned, a stop tuple `{stop,Reason}` in *`Actions`*, `Reason` has the value specified in that tuple. If it is because of a failure, `Reason` is the error reason.

If the `gen_statem` is part of a supervision tree and is ordered by its supervisor to terminate, this function is called with `Reason = shutdown` if both the following conditions apply:

- The `gen_statem` has been set to trap exit signals.
- The shutdown strategy as defined in the supervisor's child specification is an integer time-out value, not `brutal_kill`.

Even if the `gen_statem` is **not** part of a supervision tree, this function is called if it receives an `'EXIT'` message from its parent. `Reason` is the same as in the `'EXIT'` message.

Otherwise, the `gen_statem` is immediately terminated.

Notice that for any other reason than `normal`, `shutdown`, or `{shutdown,Term}`, the `gen_statem` is assumed to terminate because of an error and an error report is issued using *`error_logger:format/2`*.

See Also

`gen_event(3)`, *`gen_fsm(3)`*, *`gen_server(3)`*, *`proc_lib(3)`*, *`supervisor(3)`*, *`sys(3)`*.

io

Erlang module

This module provides an interface to standard Erlang I/O servers. The output functions all return `ok` if they are successful, or exit if they are not.

All functions in this module have an optional parameter `IoDevice`. If included, it must be the pid of a process that handles the I/O protocols. Normally, it is the `IoDevice` returned by *`file:open/2`*.

For a description of the I/O protocols, see section *The Erlang I/O Protocol* in the User's Guide.

Warning:

As from Erlang/OTP R13A, data supplied to function *`put_chars/2`* is to be in the *`unicode:chardata()`* format. This means that programs supplying binaries to this function must convert them to UTF-8 before trying to output the data on an I/O device.

If an I/O device is set in binary mode, functions *`get_chars/2,3`* and *`get_line/1,2`* can return binaries instead of lists. The binaries are, as from Erlang/OTP R13A, encoded in UTF-8.

To work with binaries in ISO Latin-1 encoding, use the *`file`* module instead.

For conversion functions between character encodings, see the *`unicode`* module.

Data Types

```
device() = atom() | pid()
```

An I/O device, either `standard_io`, `standard_error`, a registered name, or a pid handling I/O protocols (returned from *`file:open/2`*).

```
opt_pair() =
    {binary, boolean()} |
    {echo, boolean()} |
    {expand_fun, expand_fun()} |
    {encoding, encoding()}
expand_fun() =
    fun((term()) -> {yes | no, string(), [string(), ...]})
encoding() =
    latin1 |
    unicode |
    utf8 |
    utf16 |
    utf32 |
    {utf16, big | little} |
    {utf32, big | little}
setopt() = binary | list | opt_pair()
format() = atom() | string() | binary()
location() = erl_anno:location()
prompt() = atom() | unicode:chardata()
server_no_data() = {error, ErrorDescription :: term()} | eof
```

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` (that is, the width of a terminal). The function succeeds for terminal devices and returns `{error, enotsup}` for all other I/O devices.

format(Format) -> ok
format(Format, Data) -> ok
format(IoDevice, Format, Data) -> ok
fwrite(Format) -> ok
fwrite(Format, Data) -> ok
fwrite(IoDevice, Format, Data) -> ok

Types:

IoDevice = *device()*
Format = *format()*
Data = [term()]

Writes the items in `Data` (`[]`) on the standard output (`IoDevice`) in accordance with `Format`. `Format` contains plain characters that are copied to the output device, and control sequences for formatting, see below. If `Format` is an atom or a binary, it is first converted to a list with the aid of `atom_to_list/1` or `binary_to_list/1`. Example:

```
1> io:fwrite("Hello world!~n", []).
Hello world!
ok
```

The general format of a control sequence is `~F.P.PadModC`.

Character `C` determines the type of control sequence to be used, `F` and `P` are optional numeric arguments. If `F`, `P`, or `Pad` is `*`, the next argument in `Data` is used as the numeric value of `F` or `P`.

- `F` is the `field width` of the printed argument. A negative value means that the argument is left-justified within the field, otherwise right-justified. If no field width is specified, the required print width is used. If the field width specified is too small, the whole field is filled with `*` characters.
- `P` is the `precision` of the printed argument. A default value is used if no precision is specified. The interpretation of precision depends on the control sequences. Unless otherwise specified, argument `within` is used to determine print width.
- `Pad` is the padding character. This is the character used to pad the printed representation of the argument so that it conforms to the specified field width and precision. Only one padding character can be specified and, whenever applicable, it is used for both the field width and precision. The default padding character is `' '` (space).
- `Mod` is the control sequence modifier. It is either a single character (`t`, for Unicode translation, and `l`, for stopping `p` and `P` from detecting printable characters) that changes the interpretation of `Data`.

Available control sequences:

~

Character ~ is written.

c

The argument is a number that is interpreted as an ASCII code. The precision is the number of times the character is printed and defaults to the field width, which in turn defaults to 1. Example:

```
1> io:fwrite("|~10.5c|~-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 is to 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 that 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 < 1.

e

The argument is a float that is written as `[-]d.ddde+-ddd`, where the precision is the number of digits written. The default precision is 6 and it cannot be < 2.

g

The argument is a float that is written as `f`, if it is >= 0.1 and < 10000.0. Otherwise, it is written in the `e` format. The precision is the number of significant digits. It defaults to 6 and is not to be < 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 specified precision and then padded and justified to the specified 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|
3> io:fwrite("|~-10.8s|~n", [io_lib:write({hey, hey, hey})]).
|{hey,hey  |
ok
```

A list with integers > 255 is considered an error if the Unicode translation modifier is not specified:

```
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. Atom characters > 255 are escaped unless the Unicode translation modifier (`t`) is used. 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. Left-justification is not supported. 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]).
[{attributes,[[{id,age,1.5},{mode,explicit},{typename,
[73,78,84,69,71,69,82]}],[{id,cho},{mode,explicit},{typena
me,'Cho'}]]},{typename,'Person'},{tag,{'PRIVATE',3}},{mode
,implicit}]
ok
3> io:fwrite("~62p~n", [T]).
[{attributes,[[{id,age,1.5},
               {mode,explicit},
               {typename,"INTEGER"}],
              [{id,cho},{mode,explicit},{typename,'Cho'}]]},
 {typename,'Person'},
 {tag,{'PRIVATE',3}},
 {mode,implicit}]
ok
```

The field width specifies the maximum line length. 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 *`write/1`* or *`format/1,2,3`*. 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 `l` is specified, no detection of printable character lists takes place, for example:

```
5> S = [{a,"a"}, {b, "b"}].
6> io:fwrite("~15p~n", [S]).
[{a,"a"},
 {b,"b"}]
ok
7> io:fwrite("~15lp~n", [S]).
[{a,[97]},
 {b,[98]}]
ok
```

Binaries that look like UTF-8 encoded strings are output with the string syntax if the Unicode translation modifier is specified:

```
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 that 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 is reached, it cannot be 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 that 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
ok
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. Example:

```
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. Example:

```
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.

The function returns:

ok

The formatting succeeded.

If an error occurs, there is no output. 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]).
** exception exit: {badarg,[{io,format,[<0.22.0>,"~s","A"]},
                            {erl_eval,do_apply,5},
                            {shell,exprs,6},
                            {shell,eval_exprs,6},
                            {shell,eval_loop,3}]}
     in function  io:o_request/2
```

In this example, an attempt was made to output the single character 65 with the aid of the string formatting directive `"~s"`.

`fread(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 that directs the interpretation of the input.

`Format` can contain the following:

- Whitespace characters (**Space**, **Tab**, and **Newline**) that cause input to be read to the next non-whitespace character.
- Ordinary characters that must match the next input character.
- Control sequences, which have the general format `~*FMC`, where:
  - Character `*` is an optional return suppression character. It provides a method to specify a field that 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 supported, meaning Unicode translation).
  - `C` determines the type of control sequence.

  Unless otherwise specified, leading whitespace is ignored for all control sequences. An input field cannot be more than one line wide.

  Available control sequences:

  `~`

  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 whitespace characters are not skipped.

`-`

An optional sign character is expected. A sign character `-` gives return value `-1`. Sign character `+` or none gives `1`. The field width parameter is ignored. Leading whitespace 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-whitespace characters is read. If a field width has been specified, this number of characters are read and all trailing whitespace characters are stripped. An Erlang string (list of characters) is returned.

If Unicode translation is in effect (`~ts`), characters > 255 are accepted, otherwise not. With the translation modifier, the returned list can as a consequence also contain integers > 255:

```
1> io:fread("Prompt> ","~s").
Prompt> <Characters beyond latin1 range not printable in this medium>
{error,{fread,string}}
2> io:fread("Prompt> ","~ts").
Prompt> <Characters beyond latin1 range not printable in this medium>
{ok,[[1091,1085,1080,1094,1086,1076,1077]]}
```

`a`

Similar to `s`, but the resulting string is converted into an atom.

`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 whitespace 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 that have been scanned up to that point, including whitespace characters.

The function 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 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    "]}
```

`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`.

The function returns:

`Data`

The input characters. If the I/O device supports Unicode, the data can represent codepoints > 255 (the `latin1` range). If the I/O server is set to deliver binaries, they are encoded in UTF-8 (regardless of whether the I/O device supports Unicode).

`eof`

End of file was encountered.

`{error, ErrorDescription}`

Other (rare) error condition, such as `{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`.

The function returns:

`Data`

The characters in the line terminated by a line feed (or end of file). If the I/O device supports Unicode, the data can represent codepoints > 255 (the `latin1` range). If the I/O server is set to deliver binaries, they are encoded in UTF-8 (regardless of if the I/O device supports Unicode).

`eof`

End of file was encountered.

`{error, ErrorDescription}`

Other (rare) error condition, such as `{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()`

Requests all available options and their current values for a specific I/O device, for 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`. However, the standard shell 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.

`nl() -> ok`
`nl(IoDevice) -> ok`

Types:

`IoDevice = device()`

Writes new line to the standard output (`IoDevice`).

`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()} |
    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`). Argument `Options` is passed on as argument `Options` of function *`erl_scan:tokens/4`*. The data is tokenized and parsed as if it was a sequence of Erlang expressions until a final dot (`.`) is reached.

The function returns:

`{ok, ExprList, 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, such as `{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()
server_no_data() = {error, ErrorDescription :: term()} | eof
```

Reads data from the standard input (`IoDevice`), prompting it with `Prompt`. Starts reading at location `StartLocation` (`1`). Argument `Options` is passed on as argument `Options` of function *`erl_scan:tokens/4`*. The data is tokenized and parsed as if it was an Erlang form (one of the valid Erlang expressions in an Erlang source file) until a final dot (`.`) is reached.

The function 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, such as `{error, estale}` if reading from an NFS file system.

printable_range() -> unicode | latin1

Returns 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.

The only valid values for `<range>` are `latin1` and `unicode`. `latin1` means that only code points < 256 (except control characters, and so on) are considered printable. `unicode` means that all printable characters in all Unicode character ranges are considered printable by the I/O functions.

By default, Erlang is started so that only the `latin1` range of characters indicate that a list of integers is a string.

The simplest way to use the setting is to call *`io_lib:printable_list/1`*, which uses the return value of this function to decide if a list is a string of printable characters.

> **Note:**
>
> In a future release, this function may return more values and ranges. To avoid compatibility problems, it is recommended to use function *`io_lib:printable_list/1`*.

```
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`).

```
read(Prompt) -> Result
read(IoDevice, Prompt) -> Result
```

Types:

```
IoDevice = device()
Prompt = prompt()
Result =
    {ok, Term :: term()} | server_no_data() | {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`.

The function returns:

`{ok, Term}`

The parsing was successful.

`eof`

End of file was encountered.

`{error, ErrorInfo}`

The parsing failed.

`{error, ErrorDescription}`

Other (rare) error condition, such as `{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`. Argument `Options` is passed on as argument `Options` of function *`erl_scan:tokens/4`*.

The function 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, such as `{error, estale}` if reading from an NFS file system.

```
rows() -> {ok, integer() >= 1} | {error, enotsup}
rows(IoDevice) -> {ok, integer() >= 1} | {error, enotsup}
```

Types:

```
  IoDevice = device()
```

Retrieves the number of rows of `IoDevice` (that is, the height of a terminal). The function only succeeds for terminal devices, for all other I/O 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`). Argument `Options` is passed on as argument `Options` of function *`erl_scan:tokens/4`*. The data is tokenized as if it were a sequence of Erlang expressions until a final dot (`.`) is reached. This token is also returned.

The function 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, such as `{error, estale}` if reading from an NFS file system.

Example:

```
23> io:scan_erl_exprs('enter>').
enter>abc(), "hey".
{ok,[{atom,1,abc},{'(',1},{')',1},{',',1},{string,1,"hey"},{dot,1}],2}
24> io:scan_erl_exprs('enter>').
enter>1.0er.
{error,{1,erl_scan,{illegal,float}},2}
```

`scan_erl_form(Prompt) -> 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`). Argument `Options` is passed on as argument `Options` of function *`erl_scan:tokens/4`*. The data is tokenized as if it was 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/1,2,3,4`*.

`setopts(Opts) -> ok | {error, Reason}`
`setopts(IoDevice, Opts) -> ok | {error, Reason}`

Types:

`IoDevice = device()`
`Opts = [setopt()]`
`Reason = term()`

Set options for the standard I/O device (`IoDevice`).

Possible options and values vary depending on the I/O device. For a list of supported options and their current values on a specific I/O device, use function *`getopts/1`*.

The options and values supported by the OTP I/O devices are as follows:

`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 (for details, see section *The Erlang I/O Protocol*) in the User's Guide). The immediate effect is that *`get_chars/2,3`* and *`get_line/1,2`* return UTF-8 binaries instead of lists of characters for the affected I/O device.

By default, all I/O devices in OTP are set in `list` mode. However, 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 is to echo input. Only supported for the standard shell I/O server (`group.erl`)

`{expand_fun, expand_fun()}`

Provides 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 is to 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; the third is a list of possible expansions. If this list is not empty, it is printed and the current input line is written once again.

Trivial example (beep on anything except empty line, which is expanded to `"quit"`):

```
fun("") -> {yes, "quit", []};
   (_) -> {no, "", ["quit"]} end
```

This option is only supported by the standard shell (`group.erl`).

`{encoding, latin1 | unicode}`

Specifies how characters are input or output from or to the I/O device, implying that, for example, 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 I/O device is to handle Unicode characters to the "physical" device.

The standard shell is set for `unicode` or `latin1` encoding when the system is started. The 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. So, the user can input Unicode characters and the I/O device is in `{encoding, unicode}` mode if the I/O device supports it. The mode can be changed, if the assumption of the runtime system is wrong, by setting this option.

The I/O device used when Erlang is started with the "-oldshell" or "-noshell" flags is by default set to `latin1` encoding, meaning that any characters > codepoint 255 are 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 are 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 has the effect that data is expected to be read as the specified encoding from the file, and the data is written in the specified encoding to the disk file.

`{encoding, utf8}` has the same effect as `{encoding, unicode}` on files.

The extended encodings are only supported on disk files (opened by function *`file:open/2`*).

`write(Term) -> ok`

`write(IoDevice, Term) -> ok`

Types:

`IoDevice = `*`device()`*

`Term = term()`

Writes term `Term` to the standard output (`IoDevice`).

Standard Input/Output

All Erlang processes have a default standard I/O device. This device is used when no `IoDevice` argument is specified in the function calls in this module. However, it is sometimes desirable to use an explicit `IoDevice` argument that refers to the default I/O device. This is the case with functions that can access either a file or the default I/O 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 can be convenient. The I/O device `standard_error` can be used to direct output to whatever the current operating system considers a suitable I/O 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 in this module is the standard `ErrorInfo` structure that is returned from all I/O modules. It has the following format:

```
{ErrorLocation, Module, ErrorDescriptor}
```

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

```
Module:format_error(ErrorDescriptor)
```

io_lib

Erlang module

This module contains functions for converting to and from strings (lists of characters). They are used for implementing the functions in the *io* module. There is no guarantee that the character lists returned from some of the functions are flat, they can be deep lists. Function *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()]
```

```
format_spec() =
    #{control_char := char(),
      args := [any()],
      width := none | integer(),
      adjust := left | right,
      precision := none | integer(),
      pad_char := char(),
      encoding := unicode | latin1,
      strings := boolean()}
```

Where:

- `control_char` is the type of control sequence: `$P`, `$w`, and so on.
- `args` is a list of the arguments used by the control sequence, or an empty list if the control sequence does not take any arguments.
- `width` is the field width.
- `adjust` is the adjustment.
- `precision` is the precision of the printed argument.
- `pad_char` is the padding character.
- `encoding` is set to `true` if translation modifier `t` is present.
- `strings` is set to `false` if modifier `l` is present.

Exports

build_text(FormatList) -> *chars()*

Types:

FormatList = [char() | *format_spec()*]

For details, see *scan_format/2*.

char_list(Term) -> boolean()

Types:

Term = term()

Returns `true` if `Term` is a flat list of characters in the Unicode range, otherwise `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 `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 `false`.

format(Format, Data) -> *chars()*
fwrite(Format, Data) -> *chars()*

Types:

Format = *io:format()*
Data = [term()]

Returns a character list that represents `Data` formatted in accordance with `Format`. For a detailed description of the available formatting options, see *io:fwrite/1,2,3*. If the format string or argument list contains an error, a fault is generated.

If and only if the Unicode translation modifier is used in the format string (that is, `~ts` or `~tc`), the resulting list can contain characters beyond the ISO Latin-1 character range (that is, numbers > 255). If so, the result is still an ordinary Erlang `string()`, and can well be used in any context where Unicode data is allowed.

fread(Format, String) -> Result

Types:

```
Format = String = string()
Result =
    {ok, InputList :: [fread_item()], LeftOverChars :: string()} |
    {more,
     RestFormat :: string(),
     Nchars :: integer() >= 0,
     InputStack :: chars()} |
```

`{error, {fread, What :: fread_error()}}`

Tries to read `String` in accordance with the control sequences in `Format`. For a detailed description of the available formatting options, see *`io:fread/3`*. It is assumed that `String` contains whole lines.

The function returns:

`{ok, InputList, LeftOverChars}`

The string was read. `InputList` is the list of successfully matched and read items, and `LeftOverChars` are the input characters not used.

`{more, RestFormat, Nchars, InputStack}`

The string was read, but more input is needed to complete the original format string. `RestFormat` is the remaining format string, `Nchars` is the number of characters scanned, and `InputStack` is the reversed list of inputs matched up to that point.

`{error, What}`

The read operation failed and parameter `What` gives a hint about the error.

Example:

```
3> io_lib:fread("~f~f~f", "15.6 17.3e-6 24.5").
{ok,[15.6,1.73e-5,24.5],[]}
```

`fread(Continuation, 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 `[]`. For a complete description of how the re-entrant input scheme works, see Armstrong, Virding, Williams: 'Concurrent Programming in Erlang', Chapter 13.

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 was encountered. `LeftOverChars` are the input characters not used.

`{error, What}`

An error occurred and 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.

`indentation(String, StartIndent) -> integer()`

Types:

`String = string()`
`StartIndent = integer()`

Returns the indentation if `String` has been printed, starting at `StartIndent`.

`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 `false`.

`nl() -> string()`

Returns a character list that represents a new line character.

`print(Term) -> *chars()*`
`print(Term, Column, LineLength, Depth) -> *chars()*`

Types:

`Term = term()`
`Column = LineLength = integer() >= 0`
`Depth = *depth()*`

Returns a list of characters that represents `Term`, but breaks representations longer than one line into many lines and indents each line sensibly. Also tries to detect and output lists of printable characters as strings.

- `Column` is the starting column; defaults to 1.
- `LineLength` is the maximum line length; defaults to 80.
- `Depth` is the maximum print depth; defaults to -1, which means no limitation.

`printable_latin1_list(Term) -> boolean()`

Types:

`Term = term()`

Returns `true` if `Term` is a flat list of printable ISO Latin-1 characters, otherwise `false`.

`printable_list(Term) -> boolean()`

Types:

`Term = term()`

Returns `true` if `Term` is a flat list of printable characters, otherwise `false`.

What is a printable character in this case is determined by startup flag `+pc` to the Erlang VM; see *`io:printable_range/0`* and *`erl(1)`*.

printable_unicode_list(Term) -> boolean()

Types:

Term = term()

Returns `true` if `Term` is a flat list of printable Unicode characters, otherwise `false`.

scan_format(Format, Data) -> FormatList

Types:

Format = *io:format()*
Data = [term()]
FormatList = [char() | *format_spec()*]

Returns a list corresponding to the specified format string, where control sequences have been replaced with corresponding tuples. This list can be passed to:

- *`build_text/1`* to have the same effect as `format(Format, Args)`
- *`unscan_format/1`* to get the corresponding pair of `Format` and `Args` (with every `*` and corresponding argument expanded to numeric values)

A typical use of this function is to replace unbounded-size control sequences like `~w` and `~p` with the depth-limited variants `~W` and `~P` before formatting to text in, for example, a logger.

unscan_format(FormatList) -> {Format, Data}

Types:

FormatList = [char() | *format_spec()*]
Format = *io:format()*
Data = [term()]

For details, see *`scan_format/2`*.

write(Term) -> *chars()*
write(Term, Depth) -> *chars()*
write(Term, Options) -> *chars()*

Types:

Term = term()
Options = [Option]
Option = {depth, Depth} | {encoding, latin1 | utf8 | unicode}
Depth = *depth()*

Returns a character list that represents `Term`. Argument `Depth` controls the depth of the structures written. When the specified depth is reached, everything below this level is replaced by "`...`". `Depth` defaults to -1, which means no limitation.

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],[...],...}"
```

`write_atom(Atom) -> chars()`

Types:

`Atom = atom()`

Returns the list of characters needed to print atom `Atom`.

`write_atom_as_latin1(Atom) -> latin1_string()`

Types:

`Atom = atom()`

Returns the list of characters needed to print atom `Atom`. Non-Latin-1 characters are escaped.

`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.

`write_latin1_string(Latin1String) -> latin1_string()`

Types:

`Latin1String = latin1_string()`

Returns the list of characters needed to print `Latin1String` as a string.

`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.

lib

Erlang module

> **Warning:**
>
> This module is retained for backward compatibility. It can disappear without warning in a future Erlang/OTP release.

Exports

`error_message(Format, Args) -> ok`

Types:

`Format = io:format()`
`Args = [term()]`

Prints error message `Args` in accordance with `Format`. Similar to *`io:format/2`*.

`flush_receive() -> ok`

Flushes the message buffer of the current process.

`nonl(String1) -> String2`

Types:

`String1 = String2 = string()`

Removes the last newline character, if any, in `String1`.

`progname() -> atom()`

Returns the name of the script that started the current Erlang session.

`send(To, Msg) -> Msg`

Types:

`To = pid() | atom() | {atom(), node()}`
`Msg = term()`

Makes it possible to send a message using the `apply/3` BIF.

`sendw(To, Msg) -> term()`

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},
    receive
        Reply -> Reply
    end.
```

The returned message is not necessarily a reply to the sent message.

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 sublists of `ListOfLists` have been appended.

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`.

Example:

```
> lists:append("abc", "def").
"abcdef"
```

`lists:append(A, B)` is equivalent to `A ++ B`.

concat(Things) -> string()

Types:

```
Things = [Thing]
Thing = atom() | integer() | float() | string()
```

Concatenates the text representation of the elements of `Things`. The elements of `Things` can be atoms, integers, floats, or strings.

Example:

```
> 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 is to be non-empty, otherwise the function crashes 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 containing `N` copies of term `Elem`.

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/1` 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]).
[1,2]
```

flatlength(DeepList) -> integer() >= 0

Types:

DeepList = [term() | DeepList]

Equivalent to `length(flatten(DeepList))`, but more efficient.

flatmap(Fun, List1) -> List2

Types:

Fun = fun((A) -> [B])
List1 = [A]
List2 = [B]
A = B = term()

Takes a function from 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:

```
flatmap(Fun, List1) ->
    append(map(Fun, List1)).
```

Example:

```
> lists:flatmap(fun(X)->[X,X] end, [a,b,c]).
[a,a,b,b,c,c]
```

flatten(DeepList) -> List

Types:

DeepList = [term() | DeepList]
List = [term()]

Returns a flattened version of `DeepList`.

flatten(DeepList, Tail) -> List

Types:

DeepList = [term() | DeepList]
Tail = List = [term()]

Returns a flattened version of `DeepList` with tail `Tail` appended.

foldl(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.

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.

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 is usually preferred to `foldr/3`.

join(Sep, List1) -> List2

Types:

```
Sep = T
List1 = List2 = [T]
T = term()
```

Inserts `Sep` between each element in `List1`. Has no effect on the empty list and on a singleton list. For example:

```
> lists:join(x, [a,b,c]).
[a,x,b,x,c]
> lists:join(x, [a]).
[a]
> lists:join(x, []).
[]
```

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 `N`th 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 `N`th 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 `N`th 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 `N`th 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 `N`th element of each tuple. Both `TupleList1` and `TupleList2` must be key-sorted before 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 `N`th 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 `N`th element compares equal to `Key`. Returns `{value, Tuple}` if such a tuple is found, otherwise `false`.

Note:

This function is retained for backward compatibility. Function *`keyfind/3`* is usually 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 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() >= 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() >= 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 `A`s to `B`s, and a list of `A`s and produces a list of `B`s by applying the function to every element in the list. This function is used to obtain the return values. The evaluation order depends on the implementation.

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()

Combines the operations of *map/2* and *foldl/3* into one pass.

Example:

Summing the elements in a list and double them at the same time:

```
> lists:mapfoldl(fun(X, Sum) -> {2*X, X+Sum} end,
0, [1,2,3,4,5]).
{[2,4,6,8,10],15}
```

mapfoldr(Fun, Acc0, List1) -> {List2, Acc1}

Types:

Fun = fun((A, AccIn) -> {B, AccOut})
Acc0 = Acc1 = AccIn = AccOut = term()
List1 = [A]
List2 = [B]
A = B = term()

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 sublists of `ListOfLists`. All sublists must be sorted before evaluating this function. When two elements compare equal, the element from the sublist 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 before 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` before evaluating this function. `Fun(A, B)` is to return `true` if `A` compares less than or equal to `B` in the ordering, otherwise `false`. 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 before 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 `N`th element of `List`.

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 `N`th tail of `List`, that is, the sublist of `List` starting at `N+1` and continuing up to the end of the list.

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:

```
> lists:partition(fun(A) -> A rem 2 == 1 end, [1,2,3,4,5,6,7]).
{[1,3,5,7],[2,4,6]}
> lists:partition(fun(A) -> is_atom(A) end, [a,b,1,c,d,2,3,4,e]).
{[a,b,c,d,e],[1,2,3,4]}
```

For a different way to partition a list, see *splitwith/2*.

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 tail `Tail` appended.

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 that starts with `From` and contains the successive results of adding `Incr` to the previous element, until `To` is reached or passed (in the latter case, `To` is not an element of the sequence). `Incr` defaults to 1.

Failures:

- If `To < From - Incr` and `Incr > 0`.
- If `To > From - Incr` and `Incr < 0`.
- 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)` is to return `true` if `A` compares less than or equal to `B` in the ordering, otherwise `false`.

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 remaining elements (the `N`th 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]}
```

For a different way to partition a list, see *`partition/2`*.

sublist(List1, Len) -> List2

Types:

List1 = List2 = [T]
Len = integer() >= 0
T = term()

Returns the sublist of `List1` starting at position 1 and with (maximum) `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

1..(length(List1)+1)

Len = integer() >= 0
T = term()

Returns the sublist of `List1` starting at `Start` and with (maximum) `Len` elements. It is not an error for `Start+Len` to exceed the length of the list.

Examples:

```
> 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` that is a copy of `List1`, subjected to the following procedure: for each element in `List2`, its first occurrence in `List1` is deleted.

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 is very slow if both `A` and `B` are long lists. (If both lists are long, it is a much better choice to use ordered lists and *`ordsets:subtract/2`*.

`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 `N`th element of each tuple. Both `TupleList1` and `TupleList2` must be key-sorted without duplicates before evaluating this function. When two tuples compare equal, the tuple from `TupleList1` is picked and the one from `TupleList2` is 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 list `TupleList1` where all except the first tuple of the tuples comparing equal have been deleted. Sorting is performed on the `N`th element of the tuples.

umerge(ListOfLists) -> List1

Types:

ListOfLists = [List]

List = List1 = [T]

T = term()

Returns the sorted list formed by merging all the sublists of `ListOfLists`. All sublists must be sorted and contain no duplicates before evaluating this function. When two elements compare equal, the element from the sublist with the lowest position in `ListOfLists` is picked and the other is 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 before evaluating this function. When two elements compare equal, the element from `List1` is picked and the one from `List2` is 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 before evaluating this function. `Fun(A, B)` is to return `true` if `A` compares less than or equal to `B` in the ordering, otherwise `false`. When two elements compare equal, the element from `List1` is picked and the one from `List2` is 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 before 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 is 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 except 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 containing the sorted elements of `List1` where all except the first element of the elements comparing equal according to the *ordering function* `Fun` have been deleted. `Fun(A, B)` is to return `true` if `A` compares less than or equal to `B` in the ordering, otherwise `false`.

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 the 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 the 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()
```

Combines 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 is `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()
```

Combines 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 is `Combine(X, Y, Z)`.

`zipwith3(fun(X, Y, Z) -> {X,Y,Z} end, List1, List2, List3)` is equivalent to `zip3(List1, List2, List3)`.

Examples:

```
> lists:zipwith3(fun(X, Y, Z) -> X+Y+Z end, [1,2,3], [4,5,6], [7,8,9]).
[12,15,18]
> lists:zipwith3(fun(X, Y, Z) -> [X,Y,Z] end, [a,b,c], [x,y,z], [1,2,3]).
[[a,x,1],[b,y,2],[c,z,3]]
```

log_mf_h

Erlang module

This module is a `gen_event` handler module that can be installed in any `gen_event` process. It logs onto disk all events that 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(3)`*) must be used to read the files. The events are written to multiple files. When all files have been used, the first one is reused 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. Returns `Args`, which is to 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)`*

maps

Erlang module

This module contains functions for maps processing.

Exports

`filter(Pred, Map1) -> Map2`

Types:

`Pred = fun((Key, Value) -> boolean())`
`Key = Value = term()`
`Map1 = Map2 = map()`

Returns a map `Map2` for which predicate `Pred` holds true in `Map1`.

The call fails with a `{badmap,Map}` exception if `Map1` is not a map, or with `badarg` if `Pred` is not a function of arity 2.

Example:

```
> M = #{a => 2, b => 3, c=> 4, "a" => 1, "b" => 2, "c" => 4},
  Pred = fun(K,V) -> is_atom(K) andalso (V rem 2) =:= 0 end,
  maps:filter(Pred,M).
#{a => 2,c => 4}
```

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

Types:

`Key = term()`
`Map = map()`
`Value = term()`

Returns a tuple `{ok, Value}`, where `Value` is the value associated with `Key`, or `error` if no value is associated with `Key` in `Map`.

The call fails with a `{badmap,Map}` exception if `Map` is not a map.

Example:

```
> Map = #{"hi" => 42},
  Key = "hi",
  maps:find(Key,Map).
{ok,42}
```

`fold(Fun, Init, Map) -> Acc`

Types:

```
Fun = fun((K, V, AccIn) -> AccOut)
Init = Acc = AccIn = AccOut = term()
Map = map()
K = V = term()
```

Calls `F(K, V, AccIn)` for every `K` to value `V` association in `Map` in any order. Function `fun F/3` must return a new accumulator, which is passed to the next successive call. This function returns the final value of the accumulator. The initial accumulator value `Init` is returned if the map is empty.

Example:

```
> 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) -> Map

Types:

```
List = [{Key, Value}]
Key = Value = term()
Map = map()
```

Takes a list of key-value tuples elements and builds a map. The associations can be in any order, and both keys and values in the association can be of any term. If the same key appears more than once, the latter (right-most) value is used and the previous values are ignored.

Example:

```
> 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 = map()
Value = term()
```

Returns value `Value` associated with `Key` if `Map` contains `Key`.

The call fails with a `{badmap,Map}` exception if `Map` is not a map, or with a `{badkey,Key}` exception if no value is associated with `Key`.

Example:

```
> 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 = map()
Value = Default = term()

Returns value `Value` associated with `Key` if `Map` contains `Key`. If no value is associated with `Key`, `Default` is returned.

The call fails with a `{badmap,Map}` exception if `Map` is not a map.

Example:

```
> 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 = map()

Returns `true` if map `Map` contains `Key` and returns `false` if it does not contain the `Key`.

The call fails with a `{badmap,Map}` 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 = map()
Keys = [Key]
Key = term()

Returns a complete list of keys, in any order, which resides within `Map`.

The call fails with a `{badmap,Map}` exception if `Map` is not a 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 = map()
K = V1 = V2 = term()
```

Produces a new map `Map2` by calling function `fun F(K, V1)` for every `K` to value `V1` association in `Map1` in any order. Function `fun F/2` must return 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 = map()
```

Merges two maps into a single map `Map3`. If two keys exist in both maps, the value in `Map1` is superseded by the value in `Map2`.

The call fails with a `{badmap,Map}` exception if `Map1` or `Map2` is not a map.

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 = map()
```

Returns a new empty map.

Example:

```
> maps:new().
#{}
```

put(Key, Value, Map1) -> Map2

Types:

```
Key = Value = term()
Map1 = Map2 = map()
```

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`.

The call fails with a `{badmap,Map}` exception if `Map1` is not a map.

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 = map()

Removes the `Key`, if it exists, and its associated value from `Map1` and returns a new map `Map2` without key `Key`.

The call fails with a `{badmap,Map}` exception if `Map1` is not a map.

Example:

```
> Map = #{"a" => 1}.
#{"a" => 1}
> maps:remove("a",Map).
#{}
> maps:remove("b",Map).
#{"a" => 1}
```

size(Map) -> integer() >= 0

Types:

Map = map()

Returns the number of key-value associations in `Map`. This operation occurs in constant time.

Example:

```
> Map = #{42 => value_two,1337 => "value one","a" => 1},
  maps:size(Map).
3
```

take(Key, Map1) -> {Value, Map2} | error

Types:

Key = term()

Map1 = map()

Value = term()

Map2 = map()

The function removes the `Key`, if it exists, and its associated value from `Map1` and returns a tuple with the removed `Value` and the new map `Map2` without key `Key`. If the key does not exist `error` is returned.

The call will fail with a `{badmap,Map}` exception if `Map1` is not a map.

Example:

```
> Map = #{"a" => "hello", "b" => "world"}.
#{"a" => "hello", "b" => "world"}
> maps:take("a",Map).
{"hello",#{"b" => "world"}}
> maps:take("does not exist",Map).
error
```

to_list(Map) -> [{Key, Value}]

Types:

Map = map()
Key = Value = term()

Returns a list of pairs representing the key-value associations of `Map`, where the pairs `[{K1,V1}, ..., {Kn,Vn}]` are returned in arbitrary order.

The call fails with a `{badmap,Map}` exception if `Map` is not a map.

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 = map()

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.

The call fails with a `{badmap,Map}` exception if `Map1` is not a map, or with a `{badkey,Key}` exception if no value is associated with `Key`.

Example:

```
> Map = #{"a" => 1}.
#{"a" => 1}
> maps:update("a", 42, Map).
#{"a" => 42}
```

update_with(Key, Fun, Map1) -> Map2

Types:

Key = term()
Map1 = Map2 = map()
Fun = fun((Value1 :: term()) -> Value2 :: term())

Update a value in a `Map1` associated with `Key` by calling `Fun` on the old value to get a new value. An exception `{badkey,Key}` is generated if `Key` is not present in the map.

Example:

```
> Map = #{"counter" => 1},
  Fun = fun(V) -> V + 1 end,
  maps:update_with("counter",Fun,Map).
#{"counter" => 2}
```

update_with(Key, Fun, Init, Map1) -> Map2

Types:

Key = term()
Map1 = Map1
Map2 = Map2
Fun = fun((Value1 :: term()) -> Value2 :: term())
Init = term()

Update a value in a `Map1` associated with `Key` by calling `Fun` on the old value to get a new value. If `Key` is not present in `Map1` then `Init` will be associated with `Key`.

Example:

```
> Map = #{"counter" => 1},
  Fun = fun(V) -> V + 1 end,
  maps:update_with("new counter",Fun,42,Map).
#{"counter" => 1,"new counter" => 42}
```

values(Map) -> Values

Types:

Map = map()
Values = [Value]
Value = term()

Returns a complete list of values, in arbitrary order, contained in map `Map`.

The call fails with a `{badmap,Map}` exception if `Map` is not a map.

Example:

```
> 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 = map()
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` is ignored.

Example:

```
> 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 = map()
K = term()
```

Returns a new map `Map2` without keys `K1` through `Kn` and their associated values from map `Map1`. Any key in `Ks` that does not exist in `Map1` is ignored

Example:

```
> Map = #{42 => value_three,1337 => "value two","a" => 1},
  Ks = ["a",42,"other key"],
  maps:without(Ks,Map).
#{1337 => "value two"}
```

math

Erlang module

This module provides an interface to a number of mathematical functions.

> **Note:**
>
> Not all functions are provided on all platforms. In particular, the *erf/1* and *erfc/1* functions are not provided on Windows.

Exports

```
acos(X) -> float()
acosh(X) -> float()
asin(X) -> float()
asinh(X) -> float()
atan(X) -> float()
atan2(Y, X) -> float()
atanh(X) -> float()
ceil(X) -> float()
cos(X) -> float()
cosh(X) -> float()
exp(X) -> float()
floor(X) -> float()
fmod(X, Y) -> float()
log(X) -> float()
log10(X) -> float()
log2(X) -> float()
pow(X, Y) -> float()
sin(X) -> float()
sinh(X) -> float()
sqrt(X) -> float()
tan(X) -> float()
tanh(X) -> float()
```

Types:

```
Y = X = number()
```

A collection of mathematical functions that return floats. Arguments are numbers.

```
erf(X) -> float()
```

Types:

```
X = number()
```

Returns the error function of X, where:

```
erf(X) = 2/sqrt(pi)*integral from 0 to X of exp(-t*t) dt.
```

`erfc(X) -> float()`

Types:

`X = number()`

`erfc(X)` returns `1.0 - erf(X)`, computed by methods that avoid cancellation for large `X`.

`pi() -> float()`

A useful number.

Limitations

As these are the C library, the same limitations apply.

ms_transform

Erlang module

This module provides the parse transformation that makes calls to *ets* and *dbg:fun2ms/1* translate into literal match specifications. It also provides the back end for the same functions when called from the Erlang shell.

The translation from funs to match specifications is accessed through the two "pseudo functions" *ets:fun2ms/1* and *dbg:fun2ms/1*.

As everyone trying to use *ets:select/2* or *dbg* seems to end up reading this manual page, this description is an introduction to the concept of match specifications.

Read the whole manual page if it is the first time you are 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 with *lists:foldl/3*, and so on. However, the syntax of pure match specifications is awkward, as they are made up purely by Erlang terms, and the language has no syntax to make the match specifications more readable.

As the execution and structure of the match specifications are like that of a fun, it is more straightforward to write it using the familiar fun syntax and to have that translated into a match specification automatically. A real fun is clearly more powerful than the match specifications allow, but bearing the match specifications in mind, and what they can do, it is still more convenient to write it all as a fun. This module contains the code that translates the fun syntax into match specification terms.

Example 1

Using *ets:select/2* 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. One can use *ets:foldl/3* instead, but the `ets:select/2` call is far more efficient. Without the translation provided by `ms_transform`, one must struggle with writing match specifications terms to accommodate this.

Consider a simple table of employees:

```
-record(emp, {empno,     %Employee number as a string, the key
              surname,   %Surname of the employee
              givenname, %Given name of employee
              dept,      %Department, one of {dev,sales,prod,adm}
              empyear}). %Year the employee was employed
```

We create the table using:

```
ets:new(emp_tab, [{keypos,#emp.empno},named_table,ordered_set]).
```

We fill the table with randomly chosen data:

```
[{emp,"011103","Black","Alfred",sales,2000},
 {emp,"041231","Doe","John",prod,2001},
 {emp,"052341","Smith","John",dev,1997},
 {emp,"076324","Smith","Ella",sales,1995},
 {emp,"122334","Weston","Anna",prod,2002},
 {emp,"535216","Chalker","Samuel",adm,1998},
 {emp,"789789","Harrysson","Joe",adm,1996},
 {emp,"963721","Scott","Juliana",dev,2003},
 {emp,"989891","Brown","Gabriel",prod,1999}]
```

Assuming that we want the employee numbers of everyone in the sales department, there are several ways.

`ets:match/2` can be used:

```
1> ets:match(emp_tab, {'_', '$1', '_', '_', sales, '_'}).
[["011103"],["076324"]]
```

`ets:match/2` uses a simpler type of match specification, but it is still unreadable, and one has little control over the returned result. It is always a list of lists.

`ets:foldl/3` or *`ets:foldr/3`* can be used to avoid the nested lists:

```
ets:foldr(fun(#emp{empno = E, dept = sales},Acc) -> [E | Acc];
             (_,Acc) -> Acc
          end,
          [],
          emp_tab).
```

The result is `["011103","076324"]`. The fun is straightforward, so the only problem is that all the data from the table must be transferred from the table to the calling process for filtering. That is inefficient compared to the `ets:match/2` call where the filtering can be done "inside" the emulator and only the result is transferred to the process.

Consider a "pure" `ets:select/2` call that does what `ets:foldr` does:

```
ets:select(emp_tab, [{#emp{empno = '$1', dept = sales, _='_'},[],['$1']}]).
```

Although the record syntax is used, it is still 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 are not returned, as in the `ets:match/2` example. The second element, the empty list, is a list of guard expressions, which we do not need. The third element is the list of expressions constructing the return value (in ETS this is almost always a list containing one single term). In our case `'$1'` is bound to the employee number in the head (first element of the tuple), and hence the employee number is returned. The result is `["011103","076324"]`, as in the `ets:foldr/3` example, but the result is retrieved much more efficiently in terms of execution speed and memory consumption.

Using `ets:fun2ms/1`, we can combine the ease of use of the `ets:foldr/3` and the efficiency of the pure `ets:select/2` example:

```
-include_lib("stdlib/include/ms_transform.hrl").

ets:select(emp_tab, ets:fun2ms(
                      fun(#emp{empno = E, dept = sales}) ->
                              E
                      end)).
```

This example requires no special knowledge of match specifications to understand. The head of the fun matches what you want to filter out and the body returns what you want returned. As long as the fun can be kept within the limits of the match specifications, there is no need to transfer all table data to the process for filtering as in the `ets:foldr/3` example. It is easier to read than the `ets:foldr/3` example, as the select call in itself discards anything that does not match, while the fun of the `ets:foldr/3` call needs to handle both the elements matching and the ones not matching.

In the `ets:fun2ms/1` example above, it is needed to include `ms_transform.hrl` in the source code, as this is what triggers the parse transformation of the `ets:fun2ms/1` call to a valid match specification. This also implies that the transformation is done at compile time (except when called from the shell) and therefore takes no resources in runtime. That is, although you use the more intuitive fun syntax, it gets as efficient in runtime as writing match specifications by hand.

Example 2

Assume that we want to get all the employee numbers of employees hired before year 2000. Using `ets:match/2` is not an alternative here, as relational operators cannot be expressed there. Once again, `ets:foldr/3` can do it (slowly, but correct):

```
ets:foldr(fun(#emp{empno = E, empyear = Y},Acc) when Y < 2000 -> [E | Acc];
                  (_,Acc) -> Acc
          end,
          [],
          emp_tab).
```

The result is `["052341","076324","535216","789789","989891"]`, as expected. The equivalent expression using a handwritten match specification would look like this:

```
ets:select(emp_tab, [{#emp{empno = '$1', empyear = '$2', _='_'},
                      [{'<', '$2', 2000}],
                      ['$1']}]).
```

This gives the same result. `[{'<', '$2', 2000}]` is in the guard part and therefore discards anything that does not have an `empyear` (bound to `'$2'` in the head) less than 2000, as the guard in the `foldr/3` example.

We write it using `ets:fun2ms/1`:

```
-include_lib("stdlib/include/ms_transform.hrl").

ets:select(emp_tab, ets:fun2ms(
                      fun(#emp{empno = E, empyear = Y}) when Y < 2000 ->
                           E
                      end)).
```

Example 3

Assume that we want the whole object matching instead of only one element. One alternative is to assign a variable to every part of the record and build it up once again in the body of the fun, but the following is easier:

```
ets:select(emp_tab, ets:fun2ms(
                      fun(Obj = #emp{empno = E, empyear = Y})
                         when Y < 2000 ->
                              Obj
                      end)).
```

As in ordinary Erlang matching, you can bind a variable to the whole matched object using a "match inside the match", that is, a `=`. Unfortunately in funs translated to match specifications, it is allowed only at the "top-level", that is, matching the **whole** object arriving to be matched into a separate variable. If you are used to writing match specifications by hand, we mention that variable A is simply translated into '$_'. Alternatively, pseudo function `object/0` also returns the whole matched object, see section *Warnings and Restrictions*.

Example 4

This example concerns the body of the fun. Assume that all employee numbers beginning with zero (`0`) must be changed to begin with one (`1`) instead, and that we want to create the list `[{<Old empno>,<New empno>}]`:

```
ets:select(emp_tab, ets:fun2ms(
                      fun(#emp{empno = [$0 | Rest] }) ->
                              {[$0|Rest],[$1|Rest]}
                      end)).
```

This query hits the feature of partially bound keys in table type `ordered_set`, so that not the whole table needs to be searched, only the part containing keys beginning with `0` is looked into.

Example 5

The fun can have many clauses. Assume that we want to do the following:

- If an employee started before 1997, return the tuple `{inventory, <employee number>}`.

- If an employee started 1997 or later, but before 2001, return `{rookie, <employee number>}`.
- For all other employees, return `{newbie, <employee number>}`, except for those named `Smith` as they would be affronted by anything other than the tag `guru` and that is also what is returned for their numbers: `{guru, <employee number>}`.

This is accomplished as follows:

```
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 is as follows:

```
[{rookie,"011103"},
 {rookie,"041231"},
 {guru,"052341"},
 {guru,"076324"},
 {newbie,"122334"},
 {rookie,"535216"},
 {inventory,"789789"},
 {newbie,"963721"},
 {rookie,"989891"}]
```

Useful BIFs

What more can you do? A simple answer is: see the documentation of *match specifications* in ERTS User's Guide. However, the following is a brief overview of the most useful "built-in functions" that you can use when the fun is to be translated into a match specification by *`ets:fun2ms/1`*. It is not possible to call other functions than those allowed in match specifications. No "usual" Erlang code can be executed by the fun that is translated by `ets:fun2ms/1`. The fun is limited exactly to the power of the match specifications, which is unfortunate, but the price one must pay for the execution speed of `ets:select/2` compared to `ets:foldl/foldr`.

The head of the fun is 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 ETS, Dets, and Mnesia tables include that. The match specification returned by `ets:fun2ms/1` can be used with `dets:select/2` and `mnesia:select/2`, and with `ets:select/2`. The use of `=` in the head is allowed (and encouraged) at the top-level.

The guard section can contain any guard expression of Erlang. The following is a list of BIFs and expressions:

- 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`
- Boolean operators: `not`, `and`, `or`, `andalso`, `orelse`
- Relational operators: >, >=, <, =<, =:=, ==, =/=, /=
- Arithmetics: `+`, `-`, `*`, `div`, `rem`
- Bitwise operators: `band`, `bor`, `bxor`, `bnot`, `bsl`, `bsr`
- The guard BIFs: `abs`, `element`, `hd`, `length`, `node`, `round`, `size`, `tl`, `trunc`, `self`

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 is (as expected) one "match specification clause" for each semicolon-separated part of the guard. The semantics is identical to the Erlang semantics.

The body of the fun is used to construct the resulting value. When selecting from tables, one usually construct a suiting term here, using ordinary Erlang term construction, like tuple parentheses, list brackets, and variables matched out in the head, possibly with the occasional constant. Whatever expressions are allowed in guards are also allowed here, but no special functions exist 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 has not. The fun body for `ets:fun2ms/1` returns the result without side effects. 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.

Example with dbg

This section describes the slightly different match specifications translated by *`dbg:fun2ms/1`*.

The same reasons for using the parse transformation apply to `dbg`, maybe even more, as filtering using Erlang code is not a good idea when tracing (except afterwards, if you trace to file). The concept is similar to that of `ets:fun2ms/1` except that you usually use it directly from the shell (which can also be done with `ets:fun2ms/1`).

The following is an example module to trace on:

```
-module(toy).

-export([start/1, store/2, retrieve/1]).

start(Args) ->
    toy_table = ets:new(toy_table, Args).

store(Key, Value) ->
    ets:insert(toy_table, {Key,Value}).

retrieve(Key) ->
    [{Key, Value}] = ets:lookup(toy_table, Key),
    Value.
```

During model testing, the first test results in `{badmatch,16}` in `{toy,start,1}`, why?

We suspect the `ets:new/2` call, as we match hard on the return value, but want only the particular `new/2` call with `toy_table` as first parameter. So we start a default tracer on the node:

```
1> dbg:tracer().
{ok,<0.88.0>}
```

We turn on call tracing for all processes, we want to make a pretty restrictive trace pattern, so there is no need to call trace only a few processes (usually it is not):

```
2> dbg:p(all,call).
{ok,[{matched,nonode@nohost,25}]}
```

We 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 used with `dbg:fun2ms/1` 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 can also be used. The body of the fun expresses, in a more imperative way, actions to be taken if the fun head (and the guards) matches. `true` is returned here, only because the body of a fun cannot be empty. The return value is discarded.

The following trace output is received during test:

```
(<0.86.0>) call ets:new(toy_table, [ordered_set])
```

Assume that we have not found the problem yet, and want to see what `ets:new/2` returns. We use a slightly different trace pattern:

```
4> dbg:tp(ets,new,dbg:fun2ms(fun([toy_table,_]) -> return_trace() end)).
```

The following trace output is received during 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` results in a trace message 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 by far the most common call in the body of a `dbg` match specification.

The test now fails with `{badmatch,24}` because the atom `toy_table` does not match the number returned for an unnamed table. So, the problem is found, the table is to be named, and the arguments supplied by the test program do not include `named_table`. We rewrite the start function:

```
start(Args) ->
    toy_table = ets:new(toy_table, [named_table|Args]).
```

With the same tracing turned on, the following trace output is received:

```
(<0.86.0>) call ets:new(toy_table,[named_table,ordered_set])
(<0.86.0>) returned from ets:new/2 -> toy_table
```

Assume that the module now passes all testing and goes into the system. After a while, it is found that table `toy_table` grows while the system is running and that there are many elements with atoms as keys. We expected only integer keys and so does the rest of the system, but clearly not the entire system. We turn on call tracing and try to see calls to the 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/3` to ensure to catch local calls (assume that the module has grown since the smaller version and we are unsure if this inserting of atoms is not done locally). When in doubt, always use local call tracing.

Assume that nothing happens when tracing in this way. The function is never called with these parameters. We conclude that someone else (some other module) is doing it and realize that we must trace on `ets:insert/2` and want to see the calling function. The calling function can be retrieved using the match specification function `caller`. To get it into the trace message, the match specification function `message` must be used. The filter call looks like this (looking for calls to `ets:insert/2`):

```
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 is now displayed in the "additional message" part of the trace output, and the following is displayed after a while:

```
(<0.86.0>) call ets:insert(toy_table,{garbage,can}) ({evil_mod,evil_fun,2})
```

You have realized that function `evil_fun` of the `evil_mod` module, with arity `2`, is causing all this trouble.

This example illustrates the most used calls in match specifications for `dbg`. The other, more esoteric, calls are listed and explained in *Match specifications in Erlang* in ERTS User's Guide, as they are beyond the scope of this description.

Warnings and Restrictions

The following warnings and restrictions apply to the funs used in with `ets:fun2ms/1` and `dbg:fun2ms/1`.

Warning:

To use the pseudo functions triggering the translation, ensure to include the header file `ms_transform.hrl` in the source code. Failure to do so possibly results in runtime errors rather than compile time, as the expression can be valid as a plain Erlang program without translation.

Warning:

The fun must 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/1` or `dbg:fun2ms/1`. For example, `ets:fun2ms(fun(A) -> A end)` works, but not `F = fun(A) -> A end, ets:fun2ms(F)`. The latter results in a compile-time error if the header is included, otherwise a runtime error.

Many restrictions apply to the fun that is translated into a match specification. To put it simple: you cannot use anything in the fun that you cannot use in a match specification. This means that, among others, the following restrictions apply to the fun itself:

- Functions written in Erlang cannot be called, neither can local functions, global functions, or real funs.
- Everything that is written as a function call is translated into a match specification call to a built-in function, so that the call `is_list(X)` is translated to `{'is_list', '$1'}` (`'$1'` is only an example, the numbering can vary). If one tries to call a function that is not a match specification built-in, it causes an error.
- Variables occurring in the head of the fun are replaced by match specification variables in the order of occurrence, so that fragment `fun({A,B,C})` is replaced by `{'$1', '$2', '$3'}`, and so on. Every occurrence of such a variable in the match specification is replaced by a match specification variable in the same way, so that the fun `fun({A,B}) when is_atom(A) -> B end` is translated into `[{{'$1','$2'}, [{is_atom,'$1'}],['$2']}]`.
- Variables that are not included in the head are imported from the environment and made into match specification `const` expressions. Example from the shell:

```
1> X = 25.
25
2> ets:fun2ms(fun({A,B}) when A > X -> B end).
[{{'$1','$2'},[{'>','$1',{const,25}}],['$2']}]
```

- Matching with `=` cannot be used in the body. It can only be used on the top-level in the head of the fun. Example from the shell again:

```
1> ets:fun2ms(fun({A,[B|C]} = D) when A > B -> D end).
[{{'$1',['$2'|'$3']},[{'>','$1','$2'}],['$_']}]
2> ets:fun2ms(fun({A,[B|C]=D}) when A > B -> D end).
Error: fun with head matching ('=' in head) cannot be translated into
match_spec
{error,transform_error}
3> ets:fun2ms(fun({A,[B|C]}) when A > B -> D = [B|C], D end).
Error: fun with body matching ('=' in body) is illegal as match_spec
{error,transform_error}
```

All variables are bound in the head of a match specification, so the translator cannot allow multiple bindings. The special case when matching is done on the top-level makes the variable bind to `'$_'` in the resulting match specification. It is to allow a more natural access to the whole matched object. Pseudo function `object()` can 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 specification variables `'$_'` and `'$*'` can be accessed through the pseudo functions `object()` (for `'$_'`) and `bindings()` (for `'$*'`). As an example, one can translate the following `ets:match_object/2` call to a `ets:select/2` call:

  ```
  ets:match_object(Table, {'$1',test,'$2'}).
  ```

  This is the same as:

  ```
  ets:select(Table, ets:fun2ms(fun({A,test,B}) -> object() end)).
  ```

  In this simple case, the former expression is probably preferable in terms of readability.

  The `ets:select/2` call conceptually looks like this in the resulting code:

  ```
  ets:select(Table, [{{'$1',test,'$2'},[],['$_']}]).
  ```

  Matching on the top-level of the fun head can be a more natural way to access `'$_'`, see above.
- Term constructions/literals are translated as much as is needed to get them into valid match specification. This way tuples are made into match specification 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, and so on. The guard test `is_record/2` is translated into match specification code using the three parameter version that is built into match specification, so that `is_record(A,t)` is translated into `{is_record,'$1',t,5}` if the record size of record type `t` is 5.
- Language constructions such as `case`, `if`, and `catch` that are not present in match specifications are not allowed.
- If header file `ms_transform.hrl` is not included, the fun is not translated, which can result in a **runtime error** (depending on whether the fun is valid in a pure Erlang context).

  Ensure that the header is included when using `ets` and `dbg:fun2ms/1` in compiled code.
- If pseudo function triggering the translation is `ets:fun2ms/1`, the head of the fun must contain a single variable or a single tuple. If the pseudo function is `dbg:fun2ms/1`, the head of the fun must contain a single variable or a single list.

The translation from funs to match specifications is done at compile time, so runtime performance is not affected by using these pseudo functions.

For more information about match specifications, see the *Match specifications in Erlang* in ERTS User's Guide.

Exports

`format_error(Error) -> Chars`

Types:

`Error = {error, module(), term()}`

`Chars = io_lib:chars()`

Takes an error code returned by one of the other functions in the module and creates a textual description of the error.

`parse_transform(Forms, Options) -> Forms2`

Types:

`Forms = Forms2 = [erl_parse:abstract_form() | erl_parse:form_info()]`

`Options = term()`

Option list, required but not used.

Implements the transformation at compile time. This function is called by the compiler to do the source code transformation if and when header file `ms_transform.hrl` is included in the source code.

For information about how to use this parse transformation, see `ets` and `dbg:fun2ms/1`.

For a description of match specifications, see section *Match Specification in Erlang* in ERTS User's Guide.

`transform_from_shell(Dialect, Clauses, BoundEnvironment) -> term()`

Types:

`Dialect = ets | dbg`

`Clauses = [erl_parse:abstract_clause()]`

`BoundEnvironment = erl_eval:binding_struct()`

List of variable bindings in the shell environment.

Implements the transformation when the `fun2ms/1` functions are called from the shell. In this case, the abstract form is for one single fun (parsed by the Erlang shell). All imported variables are to be in the key-value list passed as `BoundEnvironment`. The result is a term, normalized, that is, not in abstract format.

orddict

Erlang module

This module provides 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 in the **Erlang term order**.

This module provides the same interface as the *`dict(3)`* module 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, Value) = [{Key, Value}]`

Dictionary as returned by *`new/0`*.

`orddict() = orddict(term(), term())`

Exports

`append(Key, Value, Orddict1) -> Orddict2`

Types:

`Orddict1 = Orddict2 = orddict(Key, Value)`

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.

See also section *Notes*.

`append_list(Key, ValList, Orddict1) -> Orddict2`

Types:

`ValList = [Value]`

`Orddict1 = Orddict2 = orddict(Key, Value)`

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.

See also section *Notes*.

`erase(Key, Orddict1) -> Orddict2`

Types:

`Orddict1 = Orddict2 = orddict(Key, Value)`

Erases all items with a specified key from a dictionary.

`fetch(Key, Orddict) -> Value`

Types:

`Orddict = orddict(Key, Value)`

Returns the value associated with `Key` in dictionary `Orddict`. This function assumes that the `Key` is present in the dictionary. An exception is generated if `Key` is not in the dictionary.

See also section *Notes*.

`fetch_keys(Orddict) -> Keys`

Types:

```
Orddict = orddict(Key, Value :: term())
Keys = [Key]
```

Returns a list of all keys in a dictionary.

`take(Key, Orddict) -> {Value, Orddict1} | error`

Types:

```
Orddict = Orddict1 = orddict(Key, Value)
Key = Value = term()
```

This function returns value from dictionary and new dictionary without this value. Returns `error` if the key is not present in the dictionary.

`filter(Pred, Orddict1) -> Orddict2`

Types:

```
Pred = fun((Key, Value) -> boolean())
Orddict1 = Orddict2 = orddict(Key, Value)
```

`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:

```
Orddict = orddict(Key, Value)
```

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.

See also section *Notes*.

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

Types:

```
Fun = fun((Key, Value, AccIn) -> AccOut)
Orddict = orddict(Key, Value)
Acc0 = Acc1 = AccIn = AccOut = Acc
```

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

`from_list(List) -> Orddict`

Types:

```
List = [{Key, Value}]
Orddict = orddict(Key, Value)
```

Converts the `Key-Value` list `List` to a dictionary.

is_empty(Orddict) -> boolean()

Types:

Orddict = *orddict()*

Returns `true` if `Orddict` has no elements, otherwise `false`.

is_key(Key, Orddict) -> boolean()

Types:

Orddict = *orddict*(Key, Value :: term())

Tests if `Key` is contained in dictionary `Orddict`.

map(Fun, Orddict1) -> Orddict2

Types:

Fun = fun((Key, Value1) -> Value2)
Orddict1 = *orddict*(Key, Value1)
Orddict2 = *orddict*(Key, Value2)

Calls `Fun` on successive keys and values of `Orddict1` tvo return a new value for each key.

merge(Fun, Orddict1, Orddict2) -> Orddict3

Types:

Fun = fun((Key, Value1, Value2) -> Value)
Orddict1 = *orddict*(Key, Value1)
Orddict2 = *orddict*(Key, Value2)
Orddict3 = *orddict*(Key, Value)

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, `Fun` is called with the key and both values to return a new value. `merge/3` can be defined as follows, but is faster:

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

new() -> *orddict()*

Creates a new dictionary.

size(Orddict) -> integer() >= 0

Types:

Orddict = *orddict()*

Returns the number of elements in an `Orddict`.

store(Key, Value, Orddict1) -> Orddict2

Types:

Orddict1 = Orddict2 = *orddict*(Key, Value)

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 = *orddict*(Key, Value)
List = [{Key, Value}]

Converts a dictionary to a list representation.

update(Key, Fun, Orddict1) -> Orddict2

Types:

Fun = fun((Value1 :: Value) -> Value2 :: Value)
Orddict1 = Orddict2 = *orddict*(Key, Value)

Updates 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:

Initial = Value
Fun = fun((Value1 :: Value) -> Value2 :: Value)
Orddict1 = Orddict2 = *orddict*(Key, Value)

Updates a value in a dictionary by calling `Fun` on the value to get a new value. If `Key` is not present in the dictionary, `Initial` is stored as the first value. For example, `append/3` can be defined as follows:

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

update_counter(Key, Increment, Orddict1) -> Orddict2

Types:

Orddict1 = Orddict2 = *orddict*(Key, Value)
Increment = number()

Adds `Increment` to the value associated with `Key` and store this value. If `Key` is not present in the dictionary, `Increment` is stored as the first value.

This can be defined as follows, but is faster:

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

Notes

Functions `append/3` and `append_list/3` are included so that keyed values can be stored in a list **accumulator**, for example:

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

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

Function `fetch/2` is to be used if the key is known to be in the dictionary, otherwise function `find/2`.

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. Elements are ordered according to the **Erlang term order**.

This module provides the same interface as the *`sets(3)`* module but with a defined representation. One difference is that while `sets` considers two elements as different if they do not match (`=:=`), this module considers two elements as different if and only if they do not compare equal (`==`).

Data Types

`ordset(T) = [T]`

As returned by *`new/0`*.

Exports

`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.

`filter(Pred, Ordset1) -> Ordset2`

Types:

`Pred = fun((Element :: T) -> boolean())`
`Ordset1 = Ordset2 = ordset(T)`

Filters elements in `Ordset1` with boolean function `Pred`.

`fold(Function, Acc0, Ordset) -> Acc1`

Types:

```
Function =
    fun((Element :: T, AccIn :: term()) -> AccOut :: term())
Ordset = ordset(T)
Acc0 = Acc1 = term()
```

Folds `Function` over every element in `Ordset` and returns the final value of the accumulator.

from_list(List) -> Ordset

Types:

List = [T]
Ordset = *ordset*(T)

Returns an ordered set of the elements in `List`.

intersection(OrdsetList) -> Ordset

Types:

OrdsetList = [*ordset*(term()), ...]
Ordset = *ordset*(term())

Returns the intersection of the non-empty list of sets.

intersection(Ordset1, Ordset2) -> Ordset3

Types:

Ordset1 = Ordset2 = Ordset3 = *ordset*(term())

Returns the intersection of `Ordset1` and `Ordset2`.

is_disjoint(Ordset1, Ordset2) -> boolean()

Types:

Ordset1 = Ordset2 = *ordset*(term())

Returns `true` if `Ordset1` and `Ordset2` are disjoint (have no elements in common), otherwise `false`.

is_element(Element, Ordset) -> boolean()

Types:

Element = term()
Ordset = *ordset*(term())

Returns `true` if `Element` is an element of `Ordset`, otherwise `false`.

is_set(Ordset) -> boolean()

Types:

Ordset = term()

Returns `true` if `Ordset` is an ordered set of elements, otherwise `false`.

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`.

new() -> []

Returns a new empty ordered set.

`size(Ordset) -> integer() >= 0`

Types:

`Ordset = ordset(term())`

Returns the number of elements in `Ordset`.

`subtract(Ordset1, Ordset2) -> Ordset3`

Types:

`Ordset1 = Ordset2 = Ordset3 = ordset(term())`

Returns only the elements of `Ordset1` that are not also elements of `Ordset2`.

`to_list(Ordset) -> List`

Types:

`Ordset = ordset(T)`
`List = [T]`

Returns the elements of `Ordset` as a list.

`union(OrdsetList) -> Ordset`

Types:

`OrdsetList = [ordset(T)]`
`Ordset = ordset(T)`

Returns the merged (union) set of the list of sets.

`union(Ordset1, Ordset2) -> Ordset3`

Types:

`Ordset1 = ordset(T1)`
`Ordset2 = ordset(T2)`
`Ordset3 = ordset(T1 | T2)`

Returns the merged (union) set of `Ordset1` and `Ordset2`.

See Also

gb_sets(3), *sets(3)*

pool

Erlang module

This module 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(3)`* module. This effects terminal I/O, file I/O, and code loading.

If the master node fails, the entire pool exits.

Exports

`attach(Node) -> already_attached | attached`

Types:

`Node = node()`

Ensures that a pool master is running and includes `Node` in the pool master's pool of nodes.

`get_node() -> node()`

Returns the node with the expected lowest future load.

`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 that is expected to have the lowest future load.

`pspawn_link(Mod, Fun, Args) -> pid()`

Types:

`Mod = module()`
`Fun = atom()`
`Args = [term()]`

Spawns and links to a process on the pool node that is expected to have the lowest future load.

`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*. The startup 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.

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, and all the system servers, are running when it returns a value.

`stop() -> stopped`

Stops the pool and kills all the slave nodes.

Files

`.hosts.erlang` is used to pick hosts where nodes can be started. For information about format and location of this file, see *`net_adm:host_file/0`*.

`$HOME/.erlang.slave.out.HOST` is used for all extra I/O that can come from the slave nodes on standard I/O. If the startup procedure does not work, this file can indicate the reason.

proc_lib

Erlang module

This module is used to start processes adhering to the *OTP Design Principles*. Specifically, the functions in this module are used by the OTP standard behaviors (for example, `gen_server` and `gen_statem`) when starting new processes. The functions can also be used to start **special processes**, user-defined processes that comply to the OTP design principles. For an example, see section *sys and proc_lib* in OTP Design Principles.

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 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 that is 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 section *SASL Error Logging* in the SASL User's Guide.

Unlike in "plain Erlang", `proc_lib` processes will not generate **error reports**, which are written to the terminal by the emulator and do not require SASL to be started. All exceptions are converted to **exits** which are ignored by the default `error_logger` handler.

The crash report contains the previously stored information, such as ancestors and initial function, the termination reason, and information about other processes that terminate as a result of this process terminating.

Data Types

```
spawn_option() =
    link |
    monitor |
    {priority, priority_level()} |
    {max_heap_size, max_heap_size()} |
    {min_heap_size, integer() >= 0} |
    {min_bin_vheap_size, integer() >= 0} |
    {fullsweep_after, integer() >= 0} |
    {message_queue_data, off_heap | on_heap | mixed}
```

See *`erlang:spawn_opt/2,3,4,5`*.

```
priority_level() = high | low | max | normal
```

```
max_heap_size() =
    integer() >= 0 |
    #{size => integer() >= 0,
      kill => true,
      error_logger => true}
```

See *erlang:process_flag(max_heap_size, MaxHeapSize)*.

```
dict_or_pid() =
    pid() |
    (ProcInfo :: [term()]) |
```

```
{X :: integer(), Y :: integer(), Z :: integer()}
```

Exports

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 process `Pid` that sent the crash report.

format(CrashReport, Encoding, Depth) -> string()

Types:

CrashReport = [term()]
Encoding = latin1 | unicode | utf8
Depth = unlimited | integer() >= 1

This function can be used by a user-defined event handler to format a crash report. When Depth is specified as a positive integer, it is used in the format string to limit the output as follows: `io_lib:format("~P", [Term,Depth])`.

hibernate(Module, Function, Args) -> no_return()

Types:

Module = module()
Function = atom()
Args = [term()]

This function does the same as (and does call) the *hibernate/3* BIF, 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.

init_ack(Ret) -> ok
init_ack(Parent, Ret) -> ok

Types:

Parent = pid()
Ret = term()

This function must be 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.

Function `init_ack/1` uses the parent value previously stored by the start function used.

If this function is not called, the start function returns an error tuple (if a link and/or a time-out is used) or hang otherwise.

The following example illustrates how this function and `proc_lib:start_link/3` are used:

```
-module(my_proc).
-export([start_link/0]).
-export([init/1]).

start_link() ->
    proc_lib:start_link(my_proc, init, [self()]).

init(Parent) ->
    case do_initialization() of
        ok ->
            proc_lib:init_ack(Parent, {ok, self()});
        {error, Reason} ->
            exit(Reason)
    end,
    loop().

...
```

`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 in this module. `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 arguments, but the same number of atoms as the number of arguments; the first atom is `'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 fun, but the module, function for the local function implementing the fun, and the arity, for example, `{some_module,-work/3-fun-0-,0}` (meaning that the fun was created in 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.

`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 in the beginning of this manual page. The process is spawned using the *spawn* BIFs.

```
spawn_link(Fun) -> pid()
spawn_link(Node, Fun) -> pid()
spawn_link(Module, Function, Args) -> pid()
spawn_link(Node, Module, Function, Args) -> pid()
```

Types:

```
Node = node()
Fun = function()
Module = module()
Function = atom()
Args = [term()]
```

Spawns a new process and initializes it as described in the beginning of this manual page. The process is spawned using the *spawn_link* BIFs.

```
spawn_opt(Fun, SpawnOpts) -> pid()
spawn_opt(Node, Function, SpawnOpts) -> pid()
spawn_opt(Module, Function, Args, SpawnOpts) -> pid()
spawn_opt(Node, Module, Function, Args, SpawnOpts) -> pid()
```

Types:

```
Node = node()
Fun = function()
Module = module()
Function = atom()
Args = [term()]
SpawnOpts = [spawn_option()]
```

Spawns a new process and initializes it as described in the beginning of this manual page. The process is spawned using the *spawn_opt* BIFs.

Note:

Using spawn option `monitor` is not allowed. It causes the function to fail with reason `badarg`.

```
start(Module, Function, Args) -> Ret
start(Module, Function, Args, Time) -> Ret
start(Module, Function, Args, Time, SpawnOpts) -> Ret
start_link(Module, Function, Args) -> Ret
start_link(Module, Function, Args, Time) -> Ret
start_link(Module, Function, Args, Time, SpawnOpts) -> Ret
```

Types:

```
Module = 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 function `start_link/3,4,5` 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.

Argument `SpawnOpts`, if specified, is passed as the last argument to the *`spawn_opt/2,3,4,5`* BIF.

> **Note:**
>
> Using spawn option `monitor` is not allowed. It causes the function to fail with reason `badarg`.

```
stop(Process) -> ok
```

Types:

```
Process = pid() | RegName | {RegName, node()}
```

Equivalent to *`stop(Process, normal, infinity)`*.

```
stop(Process, Reason, Timeout) -> ok
```

Types:

```
Process = pid() | RegName | {RegName, node()}
Reason = term()
Timeout = timeout()
```

Orders the process to exit with the specified `Reason` and waits for it to terminate.

Returns `ok` if the process exits with the specified `Reason` within `Timeout` milliseconds.

If the call times out, a `timeout` exception is raised.

If the process does not exist, a `noproc` exception is raised.

The implementation of this function is based on the `terminate` system message, and requires that the process handles system messages correctly. For information about system messages, see *`sys(3)`* and section *sys and proc_lib* in OTP Design Principles.

`translate_initial_call(Process) -> {Module, Function, Arity}`

Types:

`Process = dict_or_pid()`
`Module = module()`
`Function = atom()`
`Arity = byte()`

This function is used by functions *`c:i/0`* and *`c:regs/0`* to present process information.

This function extracts the initial call of a process that was started using one of the spawn or start functions in this module, 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. To return information that this process is a supervisor and the name of the callback module, `Module` is `supervisor` and `Function` is the name of the supervisor callback module. `Arity` is `1`, as 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.

See Also

`error_logger(3)`

proplists

Erlang module

Property lists are ordinary lists containing entries in the form of either tuples, whose first elements are keys used for lookup and insertion, or atoms, which work as shorthand for tuples `{Atom, true}`. (Other terms are allowed in the lists, but are ignored by this module.) If there is more than one entry in a list for a certain key, the first occurrence normally overrides any later (irrespective of the arity of the tuples).

Property lists are useful for representing inherited properties, such as options passed to a function where a user can specify options overriding the default settings, object properties, annotations, and so on.

Two keys are considered equal if they match (`=:=`). That is, numbers are compared literally rather than by value, so that, for example, `1` and `1.0` are different keys.

Data Types

`property() = atom() | tuple()`

Exports

`append_values(Key, ListIn) -> ListOut`

Types:

`Key = term()`
`ListIn = ListOut = [term()]`

Similar to *`get_all_values/2`*, but each value is wrapped in a list unless it is already itself a list. The resulting list of lists is concatenated. This is often useful for "incremental" options.

Example:

```
append_values(a, [{a, [1,2]}, {b, 0}, {a, 3}, {c, -1}, {a, [4]}])
```

returns:

```
[1,2,3,4]
```

`compact(ListIn) -> ListOut`

Types:

`ListIn = ListOut = [`*`property()`*`]`

Minimizes the representation of all entries in the list. This is equivalent to `[property(P) || P <- ListIn]`.

See also *`property/1`*, *`unfold/1`*.

`delete(Key, List) -> List`

Types:

`Key = term()`
`List = [term()]`

Deletes all entries associated with `Key` from `List`.

expand(Expansions, ListIn) -> 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 because `{foo, false}` shadows `foo`:

```
expand([{{foo, true}, [bar, baz]}], [{foo, false}, fie, foo, fum])
```

Notice 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.

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`.

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`.

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`.

`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, function *`substitute_aliases/2`* is applied using the specified list of aliases:

- For a `negations` operation, `substitute_negations/2` is applied using the specified negation list.
- For an `expand` operation, function *`expand/2`* is applied using the specified list of expansions.

The final result is automatically compacted (compare *`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 *`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, `Key` is returned, 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 specified keys, also with their original relative order preserved.

Example:

```
split([{c, 2}, {e, 1}, a, {c, 3, 4}, d, {b, 5}, b], [a, b, c])
```

returns:

```
{[[a], [{b, 5}, b],[{c, 2}, {c, 3, 4}]], [{e, 1}, d]}
```

substitute_aliases(Aliases, 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.

For example, `substitute_aliases([{color, colour}], L)` replaces 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 = [{Key1, Key2}]
Key1 = Key2 = 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`: if the entry was `{K1, true}`, it is replaced with `{K2, false}`, otherwise with `{K2, true}`, thus changing the name of the option and simultaneously negating the value specified by *get_bool(Key, ListIn)*. If the same `K1` occurs more than once in `Negations`, only the first occurrence is used.

For example, `substitute_negations([{no_foo, foo}], L)` replaces 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}`.

qlc

Erlang module

This module provides a query interface to *Mnesia*, *ETS*, *Dets*, and other data structures that provide an iterator style traversal of objects.

Overview

This module provides a query interface to **QLC tables**. Typical QLC tables are Mnesia, ETS, and Dets tables. Support is also provided for user-defined tables, see section *Implementing a QLC Table*. A **query** is expressed 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 *Erlang Reference Manual* and *Programming Examples*, except that variables introduced in patterns cannot be used in list expressions. In the absence of optimizations and options such as `cache` and `unique` (see section *Common Options*, 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 *`q/1,2`* returns a **query handle**. To obtain all the answers to a query, *`eval/1,2`* is to 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 code, be careful not to keep query handles too long if the module code is to be replaced. Code replacement is described in section *Compilation and Code Loading* 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 *`cursor/1,2`* with a query handle as first argument. Query cursors are essentially Erlang processes. One answer at a time is sent from the query cursor process to the process that created the cursor.

Syntax

Syntactically QLCs have the same parts as ordinary list comprehensions:

```
[Expression || Qualifier1, Qualifier2, ...]
```

`Expression` (the **template**) is any Erlang expression. Qualifiers are either **filters** or **generators**. Filters are Erlang expressions returning `boolean()`. Generators have the form `Pattern <- ListExpression`, where `ListExpression` is an expression evaluating to a query handle or a list. Query handles are returned from *`append/1,2`*, *`keysort/2,3`*, *`q/1,2`*, *`sort/1,2`*, *`string_to_handle/1,2,3`*, and *`table/2`*.

Evaluation

A query handle is evaluated in the following order:

- Inspection of options and the collection of information about tables. As a result, qualifiers are modified during the optimization phase.
- All list expressions are evaluated. If a cursor has been created, evaluation takes place in the cursor process. For list expressions that are QLCs, the list expressions of the generators of the QLCs are evaluated as well. Be careful if list expressions have side effects, as list expressions are evaluated in unspecified order.
- 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 `boolean()` 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 generator, otherwise the candidate objects are not restricted to a small set. The reason is that objects that could make the query evaluation fail must not be excluded by looking up a key or running a match specification.

Join

The `qlc` module supports fast join of two query handles. Fast join is possible if some position `P1` of one query handler and some position `P2` of another query handler are tested for equality. Two fast join methods are provided:

- **Lookup join** traverses all objects of one query handle and finds objects of the other handle (a QLC table) such that the values at `P1` and `P2` match or compare equal. The `qlc` module does not create any indexes but looks up values using the key position and the indexed positions of the QLC table.
- **Merge join** sorts the objects of each query handle if necessary and filters out objects where the values at `P1` and `P2` do not compare equal. If many objects with the same value of `P2` exist, a temporary file is 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. That is, no query planner is provided that can select 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. Defaults to `{cache_all, no}`. 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 option `{cache, list}` and by the merge join method. Defaults to 512*1024 bytes.
- `{tmpdir_usage, TmpFileUsage}` determines the action taken when `qlc` is about to create temporary files on the directory set by option `tmpdir`. 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 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 option `{cache, list}`. The option also overrides option `tmpdir` of *`keysort/3`* and *`sort/2`*. Defaults to `""`, 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. Defaults to `{unique_all, false}`. Option `unique_all` is equivalent to `{unique_all, true}`.

Getting Started

As mentioned earlier, queries are expressed in the list comprehension syntax as described in section *Expressions* in Erlang Reference Manual. In the following, some familiarity with list comprehensions is assumed. The examples in section *List Comprehensions* in Programming Examples can get you started. Notice 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 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 *`eval/1,2`* collects answers in a list that is finally reversed, while list comprehensions collect answers on the stack that 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 through an interface function of some data structure. Erlang/OTP includes a few examples of such functions: *`mnesia:table/1,2`*, *`ets:table/1,2`*, and *`dets:table/1,2`*. For a given data structure, many functions can create QLC tables, but common for these functions is that they return a query handle created by *`qlc:table/2`*. Using the QLC tables provided by Erlang/OTP is usually probably sufficient, but for the more advanced user section *Implementing a QLC Table* describes the implementation of a function calling `qlc:table/2`.

Besides `qlc:table/2`, other functions return query handles. They are used more seldom than tables, but are sometimes useful. *`qlc:append/1,2`* traverses objects from many tables or lists after each other. If, for example, 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/2` first returns all objects of `QH`, then `{finished}`. If a tuple `{finished}` exists among the answers to `QH`, it is returned twice from `append/2`.

As another example, consider concatenating the answers to two queries `QH1` and `QH2` while removing all duplicates. This is accomplished by using option `unique`:

```
qlc:q([X || X <- qlc:append(QH1, QH2)], {unique, true})
```

The cost is substantial: every returned answer is 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` option, all answers to `QH1` would be returned followed by all answers to `QH2`. The `unique` option keeps the order between the remaining answers.

If the order of the answers is not important, there is an alternative to the `unique` option, namely 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 are sorted. If there are many answers, temporary files are used. Notice that to get the first unique answer, all answers must 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 only 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.
```

QLCs are convenient for stating constraints on data from two or more tables. The following example does a natural join on two query handles on position 2:

```
qlc:q([{X1,X2,X3,Y1} ||
          {X1,X2,X3} <- QH1,
          {Y1,Y2} <- QH2,
          X2 =:= Y2])
```

The `qlc` module evaluates 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 traverses the objects of `QH2` while looking up key values in the table. However, if not `X2` or `Y2` is matched against the key or an indexed position of a QLC table, the merge join method ensures that `QH1` and `QH2` are both sorted on position 2 and next do the join by traversing the objects one by one.

Option `join` can be used to force the `qlc` module to use a certain join method. For the rest of this section it is assumed that the excessively slow join method called "nested loop" has been chosen:

```
qlc:q([{X1,X2,X3,Y1} ||
          {X1,X2,X3} <- QH1,
          {Y1,Y2} <- QH2,
          X2 =:= Y2],
      {join, nested_loop})
```

In this case the filter is 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 is run M*N times.

If `QH2` is a call to the function for *`gb_trees`*, as defined in section *Implementing a QLC Table*, then `gb_table:table/1`, the iterator for the gb-tree is initiated for each answer to `QH1`. The objects of the gb-tree are then returned one by one. This is probably the most efficient way of traversing the table in that case, as 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 to 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 option `cache`. It is expressed like this:

```
QH2' = qlc:q([X || X <- QH2], {cache, ets})
```

or only

```
QH2' = qlc:q([X || X <- QH2], cache)
```

The effect of option `cache` is that when generator `QH2'` is run the first time, every answer is stored in an ETS table. When the next answer of `QH1` is tried, answers to `QH2'` are copied from the ETS table, which is very fast. As for option `unique` the cost is a possibly substantial amount of RAM memory.

Option `{cache, list}` offers the possibility to store the answers in a list on the process heap. This has the potential of being faster than ETS tables, as there is no need to copy answers from the table. However, it can often result in slower evaluation because of more garbage collections of the process heap and increased RAM memory consumption because of larger heaps. Another drawback with cache lists is that if the list size exceeds a limit, a temporary file is used. Reading the answers from a file is 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.

Option `cache_all` 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 are to be cached.

Implementing a QLC Table

As an example of how to use function *`table/2`*, the implementation of a QLC table for the *`gb_trees`* module is given:

```
-module(gb_table).

-export([table/1]).

table(T) ->
    TF = fun() -> qlc_next(gb_trees:next(gb_trees:iterator(T))) end,
    InfoFun = fun(num_of_objects) -> gb_trees:size(T);
                 (keypos) -> 1;
                 (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,'=='}]).

qlc_next({X, V, S}) ->
    [{X,V} | fun() -> qlc_next(gb_trees:next(S)) end];
qlc_next(none) ->
    [].

gb_nodes(T, infinity, ElementFun) ->
    gb_nodes(T, -1, ElementFun);
gb_nodes(T, NElements, ElementFun) ->
    gb_iter(gb_trees:iterator(T), NElements, ElementFun).

gb_iter(_I, 0, _EFun) ->
    '...';
gb_iter(I0, N, EFun) ->
    case gb_trees:next(I0) of
        {X, V, I} ->
            [EFun({X,V}) | gb_iter(I, N-1, EFun)];
        none ->
            []
    end.
```

`TF` is the traversal function. The `qlc` module requires that there is a way of traversing all objects of the data structure. `gb_trees` has an iterator function suitable for that purpose. Notice 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. As `qlc_next/1` returns the objects as `{Key, Value}` pairs, the position is 1. Notice that the lookup function is to return `{Key, Value}` pairs, as the traversal function does.

The format function is also optional. It is called by *`info/1,2`* to give feedback at runtime of how the query is to be evaluated. Try to give as good feedback as possible without showing too much details. In the example, at most seven objects of the table are shown. The format function handles two cases: `all` means that all objects of the table are traversed; `{lookup, 1, KeyValues}` means that the lookup function is used for looking up key values.

Whether the whole table is traversed or only some keys looked up depends on how the query is expressed. If the query has the form

```
qlc:q([T || P <- LE, F])
```

and P is a tuple, the `qlc` module analyzes P and F in compile time to find positions of 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 must be traversed. The info function `InfoFun` 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 indexes upon request, and introduces 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

Erlang/OTP has two operators for testing term equality: `==/2` and `=:=/2`. The difference is all about the integers that can be represented by floats. For example, `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 at compile time can determine that some constant is free of integers, it does not matter which one of `==/2` or `=:=/2` is used:

```
1> E1 = ets:new(t, [set]), % uses =:=/2 for key equality
Q1 = qlc:q([K ||
{K} <- ets:table(E1),
K == 2.71 orelse K == a]),
io:format("~s~n", [qlc:info(Q1)]).
ets:match_spec_run(lists:flatmap(fun(V) ->
                                     ets:lookup(20493, V)
                                 end,
                                 [a,2.71]),
                   ets:match_spec_compile([{{'$1'},[],['$1']}]))
```

In the example, operator `==/2` has been handled exactly as `=:=/2` would have been handled. However, 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 option *key_equality*), then the `qlc` module does 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 must be looked up:

```
2> E2 = ets:new(t, [set]),
true = ets:insert(E2, [{{2,2},a},{{2,2.0},b},{{2.0,2},c}]),
F2 = fun(I) ->
qlc:q([V || {K,V} <- ets:table(E2), K == I])
end,
Q2 = F2({2,2}),
io:format("~s~n", [qlc:info(Q2)]).
ets:table(53264,
          [{traverse,
            {select,[{{'$1','$2'},[{'==','$1',{const,{2,2}}}],['$2']}]}}])
3> lists:sort(qlc:e(Q2)).
[a,b,c]
```

Looking up only `{2,2}` would not return `b` and `c`.

If the table uses `==/2` when comparing keys for equality, the `qlc` module looks up the constant regardless of which operator is used in the QLC. However, `==/2` is to be preferred:

```
4> E3 = ets:new(t, [ordered_set]), % uses ==/2 for key equality
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, then the `qlc` module does not consider lookup join for that table.

Data Types

`abstract_expr() = erl_parse:abstract_expr()`

Parse trees for Erlang expression, see section *The Abstract Format* in the ERTS User's Guide.

`answer() = term()`

`answers() = [answer()]`

`cache() = ets | list | no`

`match_expression() = ets:match_spec()`

Match specification, see section *Match Specifications in Erlang* in the ERTS User's Guide and `ms_transform(3)`.

`no_files() = integer() >= 1`

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())`

`query_cursor()`

A *query cursor*.

query_handle()

A *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 query handle `QH`, all answers to the first query handle in `QHL` are returned, followed by all answers to the remaining query handles in `QHL`.

append(QH1, QH2) -> QH3

Types:

QH1 = QH2 = *query_handle_or_list()*

QH3 = *query_handle()*

Returns a query handle. When evaluating 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/2* to spawn and link to a process that evaluates the query handle. The value of option `spawn_options` is used as last argument when calling `spawn_opt/2`. Defaults to `[link]`.

Example:

```
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.

e(QH) -> Answers | Error
e(QH, Options) -> Answers | Error
eval(QH) -> Answers | Error
eval(QH, Options) -> Answers | Error

Types:

QH = *query_handle_or_list()*
Answers = *answers()*
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()
```

Evaluates a query handle in the calling process and collects all answers in a list.

Example:

```
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.

Example:

```
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:

```
  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 mainly useful 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 is not certain that the information is absolutely accurate.

Options:

- The default is to return a sequence of QLCs in a block, but if option `{flat, false}` is specified, one single QLC is returned.
- The default is to return a string, but if option `{format, abstract_code}` is specified, 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 option `{n_elements, NElements}` is specified, only a limited number of elements are returned.
- The default is to show all parts of objects and match specifications, but if option `{depth, Depth}` is specified, parts of terms below a certain depth are replaced by `'...'`.

`info(QH)` is equivalent to `info(QH, [])`.

Examples:

In the following example two simple QLCs are inserted only to hold option `{unique, true}`:

```
1> QH = qlc:q([{X,Y} || X <- [x,y], Y <- [a,b]]),
io:format("~s~n", [qlc:info(QH, unique_all)]).
begin
    V1 =
        qlc:q([
               SQV ||
                   SQV <- [x,y]
              ],
              [{unique,true}]),
    V2 =
        qlc:q([
               SQV ||
                   SQV <- [a,b]
              ],
              [{unique,true}]),
    qlc:q([
           {X,Y} ||
               X <- V1,
               Y <- V2
          ],
          [{unique,true}])
end
```

In the following example QLC `V2` has been inserted to show the joined generators and the join method chosen. A convention is used for lookup join: the first generator (`G2`) is the one traversed, the second (`G1`) is the table where constants are looked up.

```
1> E1 = ets:new(e1, []),
E2 = ets:new(e2, []),
true = ets:insert(E1, [{1,a},{2,b}]),
true = ets:insert(E2, [{a,1},{b,2}]),
Q = qlc:q([{X,Z,W} ||
{X, Z} <- ets:table(E1),
{W, Y} <- ets:table(E2),
X =:= Y]),
io:format("~s~n", [qlc:info(Q)]).
begin
    V1 =
        qlc:q([
               P0 ||
                   P0 = {W,Y} <- ets:table(17)
              ]),
    V2 =
        qlc:q([
               [G1|G2] ||
                   G2 <- V1,
                   G1 <- ets:table(16),
                   element(2, G1) =:= element(1, G2)
              ],
              [{join,lookup}]),
    qlc:q([
           {X,Z,W} ||
               [{X,Z}|{W,Y}] <- V2
          ])
end
```

`keysort(KeyPos, QH1) -> QH2`

`keysort(KeyPos, QH1, SortOptions) -> QH2`

Types:

KeyPos = *key_pos()*
SortOptions = *sort_options()*
QH1 = *query_handle_or_list()*
QH2 = *query_handle()*

Returns a query handle. When evaluating query handle `QH2`, the answers to query handle `QH1` are sorted by *`file_sorter:keysort/4`* according to the options.

The sorter uses 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 option `size`.

`keysort(KeyPos, QH1)` is equivalent to `keysort(KeyPos, QH1, [])`.

next_answers(QueryCursor) -> Answers | Error
next_answers(QueryCursor, NumberOfAnswers) -> Answers | Error

Types:

QueryCursor = *query_cursor()*
Answers = *answers()*
NumberOfAnswers = all_remaining | integer() >= 1
Error = {error, module(), Reason}
Reason = *file_sorter:reason()*

Returns some or all of the remaining answers to a query cursor. Only the owner of `QueryCursor` can retrieve answers.

Optional argument `NumberOfAnswers` determines the maximum number of answers returned. Defaults to `10`. If less than the requested number of answers is returned, subsequent calls to `next_answers` return `[]`.

q(QLC) -> QH
q(QLC, Options) -> QH

Types:

QH = *query_handle()*
Options = [Option] | Option
Option =
{max_lookup, MaxLookup} |
{cache, *cache()*} |
cache |
{join, Join} |
{lookup, Lookup} |
{unique, boolean()} |
unique
MaxLookup = integer() >= 0 | infinity
Join = any | lookup | merge | nested_loop
Lookup = boolean() | any
QLC = *query_list_comprehension()*

Returns a query handle for a QLC. The QLC must be the first argument to this function, otherwise it is evaluated as an ordinary list comprehension. It is also necessary to add the following line to the source code:

```
-include_lib("stdlib/include/qlc.hrl").
```

This causes a parse transform to substitute a fun for the QLC. The (compiled) fun is 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 occurs, the fun substituted for the QLC is not compiled but is evaluated by *`erl_eval(3)`*. This is also true when expressions are evaluated by `file:eval/1,2` or in the debugger.

To be explicit, this does not work:

```
...
A = [X || {X} <- [{1},{2}]],
QH = qlc:q(A),
...
```

Variable `A` is 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, [])`.

Options:

- Option `{cache, ets}` can be used to cache the answers to a QLC. The answers are stored in one ETS table for each cached QLC. When a cached QLC is evaluated again, answers are fetched from the table without any further computations. Therefore, when all answers to a cached QLC have been found, the ETS tables used for caching answers to the qualifiers of the QLC can be emptied. Option `cache` is equivalent to `{cache, ets}`.
- Option `{cache, list}` can be used to cache the answers to a QLC 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. Option `max_list_size` sets the limit in bytes and the temporary file is put on the directory set by option `tmpdir`.

  Option `cache` has no effect if it is known that the QLC is to be evaluated at most once. This is always true for the top-most QLC and also for the list expression of the first generator in a list of qualifiers. Notice that in the presence of side effects in filters or callback functions, the answers to QLCs can be affected by option `cache`.
- Option `{unique, true}` can be used to remove duplicate answers to a QLC. The unique answers are stored in one ETS table for each QLC. The table is emptied every time it is known that there are no more answers to the QLC. Option `unique` is equivalent to `{unique, true}`. If option `unique` is combined with option `{cache, ets}`, two ETS tables are used, but the full answers are stored in one table only. If option `unique` is combined with option `{cache, list}`, the answers are sorted twice using *`keysort/3`*; once to remove duplicates and once to restore the order.

Options `cache` and `unique` apply not only to the QLC itself but also to the results of looking up constants, running match specifications, and joining handles.

Example:

In the following example the cached results of the merge join are traversed for each value of `A`. Notice that without option `cache` the join would have been carried out three times, once for each value of `A`.

```
1> Q = qlc:q([{A,X,Z,W} ||
A <- [a,b,c],
{X,Z} <- [{a,1},{b,4},{c,6}],
{W,Y} <- [{2,a},{3,b},{4,c}],
X =:= Y],
{cache, list}),
io:format("~s~n", [qlc:info(Q)]).
begin
    V1 =
        qlc:q([
               P0 ||
                   P0 = {X,Z} <-
                       qlc:keysort(1, [{a,1},{b,4},{c,6}], [])
              ]),
    V2 =
        qlc:q([
               P0 ||
                   P0 = {W,Y} <-
                       qlc:keysort(2, [{2,a},{3,b},{4,c}], [])
              ]),
    V3 =
        qlc:q([
               [G1|G2] ||
                   G1 <- V1,
                   G2 <- V2,
                   element(1, G1) == element(2, G2)
              ],
              [{join,merge},{cache,list}]),
    qlc:q([
           {A,X,Z,W} ||
               A <- [a,b,c],
               [{X,Z}|{W,Y}] <- V3,
               X =:= Y
          ])
end
```

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 *table/2*) traversal of tables can be done by looking up key values, which is assumed to be fast. Under certain (rare) circumstances there can be too many key values to look up. Option `{max_lookup, MaxLookup}` 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 is traversed instead. Defaults to `infinity`, which means that there is no limit on the number of keys to look up.

Example:

In the following example, using the `gb_table` module from section *Implementing a QLC Table*, 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 key `{X, Y}` are compared separately.

```
1> T = gb_trees:empty(),
QH = qlc:q([X || {{X,Y},_} <- gb_table:table(T),
((X == 1) or (X == 2)) andalso
((Y == a) or (Y == b) or (Y == c))]),
io:format("~s~n", [qlc:info(QH)]).
ets:match_spec_run(
       lists:flatmap(fun(K) ->
                             case
                                 gb_trees:lookup(K,
                                                 gb_trees:from_orddict([]))
                             of
                                 {value,V} ->
                                     [{K,V}];
                                 none ->
                                     []
                             end
                     end,
                     [{1,a},{1,b},{1,c},{2,a},{2,b},{2,c}]),
       ets:match_spec_compile([{{{'$1','$2'},'_'},[],['$1']}]))
```

Options:

- Option `{lookup, true}` can be used to ensure that the `qlc` module looks up constants in some QLC table. If there are more than one QLC table among the list expressions of the generators, constants must 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 when it would be unacceptable to traverse all objects in some table. Setting option `lookup` to `false` ensures that no constants are looked up (`{max_lookup, 0}` has the same effect). Defaults to `any`, which means that constants are looked up whenever possible.
- Option `{join, Join}` can be used to ensure that a certain join method is used:
  - `{join, lookup}` invokes the lookup join method.
  - `{join, merge}` invokes the merge join method.
  - `{join, nested_loop}` invokes the method of matching every pair of objects from two handles. This method is mostly very slow.

  The evaluation of the query fails if the `qlc` module cannot carry out the chosen join method. Defaults to `any`, which means that some fast join method is used if possible.

`sort(QH1) -> QH2`

`sort(QH1, SortOptions) -> QH2`

Types:

`SortOptions = `*`sort_options()`*

`QH1 = `*`query_handle_or_list()`*

`QH2 = `*`query_handle()`*

Returns a query handle. When evaluating query handle `QH2`, the answers to query handle `QH1` are sorted by *`file_sorter:sort/3`* according to the options.

The sorter uses 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 option `size`.

`sort(QH1)` is equivalent to `sort(QH1, [])`.

string_to_handle(QueryString) -> QH | Error
string_to_handle(QueryString, Options) -> QH | Error
string_to_handle(QueryString, Options, Bindings) -> QH | Error

Types:

QueryString = string()
Options = [Option] | Option
Option =
{max_lookup, MaxLookup} |
{cache, *cache()*} |
cache |
{join, Join} |
{lookup, Lookup} |
{unique, boolean()} |
unique
MaxLookup = integer() >= 0 | infinity
Join = any | lookup | merge | nested_loop
Lookup = boolean() | any
Bindings = *erl_eval:binding_struct()*
QH = *query_handle()*
Error = {error, module(), Reason}
Reason = *erl_parse:error_info()* | *erl_scan:error_info()*

A string version of *q/1,2*. When the query handle is evaluated, the fun created by the parse transform is interpreted by *erl_eval(3)*. The query string is to be one single QLC terminated by a period.

Example:

```
1> L = [1,2,3],
Bs = erl_eval:add_binding('L', L, erl_eval:new_bindings()),
QH = qlc:string_to_handle("[X+1 || X <- L].", [], Bs),
qlc:eval(QH).
[2,3,4]
```

`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 mainly useful when called from outside of Erlang, for example 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())
  FormatedTable = {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()] | term()
  ParentFun = undefined | fun(() -> ParentFunValue)
  PostFun = undefined | fun(() -> term())
  PreFun = undefined | fun((PreArgs) -> term())
  PreArgs = [PreArg]
  PreArg = {parent_value, ParentFunValue} | {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 many other data structures can be turned into QLC tables. This is accomplished by letting 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 and properties of the table are handled by callback functions provided as options to `qlc:table/2`.

- 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 `TraverseFun`s are to accept a match specification as argument. The match specification is created by the parse transform by analyzing the pattern of the generator calling `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` is called with a match specification returning every object.

  - Modules that can use match specifications for optimized traversal of tables are to call `qlc:table/2` with an unary `TraverseFun`. An example is *`ets:table/2`*.
  - Other modules can provide a nullary `TraverseFun`. An example is `gb_table:table/1` in section *Implementing a QLC Table*.

- Unary callback function `PreFun` is called once before the table is read for the first time. If the call fails, the query evaluation fails.

  Argument `PreArgs` is a list of tagged values. 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/1,2`*, *`fold/3,4`*, or *`cursor/1,2`*.
  - 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.

- 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 pre (post) functions for different tables are evaluated in unspecified order.

  Other table access than reading, such as calling `InfoFun`, is assumed to be OK at any time.

- 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 determine that the filters match and compare the element at `Position` in such a way that only `Keys` need to be looked up 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.

- Unary callback function `InfoFun` is to return information about the table. `undefined` is to 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 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.

- Unary callback function `FormatFun` is used by *`info/1,2`* for displaying the call that created the query handle of the table. Defaults to `undefined`, which 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 is added by `info/1,2` 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 argument can have the following values:

`{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 is used if `TraverseFun` is unary.

`NElements` is the value of the `info/1,2` option `n_elements`.

`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 option `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. Defaults to `'=:='`.

For the various options recognized by `table/1,2` in respective module, see *`ets(3)`*, *`dets(3)`*, and *`mnesia(3)`*.

See Also

`dets(3)`, *`erl_eval(3)`*, *`erlang(3)`*, *`error_logger(3)`*, *`ets(3)`*, *`file(3)`*, *`file_sorter(3)`*, *`mnesia(3)`*, *`shell(3)`*, *Erlang Reference Manual*, *Programming Examples*

queue

Erlang module

This module provides (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, and 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 is to 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 *`filter/2`*, *`join/2`*, *`len/1`*, *`member/2`*, *`split/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.

This module has three sets of interface functions: the "Original API", the "Extended API", and the "Okasaki API".

The "Original API" and the "Extended API" both use the mental picture of a waiting line of items. Both have reverse operations suffixed "_r".

The "Original API" item removal functions return compound terms with both the removed item and the resulting queue. The "Extended API" contains alternative functions that build less garbage and 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. This 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`*`(term())`

Exports

`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.

`from_list(L :: [Item]) -> queue(Item)`

Returns a queue containing the items in `L` in the same order; the head item of the list becomes the front item of the queue.

`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`.

`is_empty(Q :: queue()) -> boolean()`

Tests if `Q` is empty and returns `true` if so, otherwise otherwise.

`is_queue(Term :: term()) -> boolean()`

Tests if `Term` is a queue and returns `true` if so, otherwise `false`.

`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`.

`len(Q :: queue()) -> integer() >= 0`

Calculates and returns the length of queue `Q`.

`member(Item, Q :: queue(Item)) -> boolean()`

Returns `true` if `Item` matches some element in `Q`, otherwise `false`.

`new() -> queue()`

Returns an empty queue.

```
out(Q1 :: queue(Item)) ->
       {{value, Item}, Q2 :: queue(Item)} |
       {empty, Q1 :: queue(Item)}
```

Removes the item at the front of queue `Q1`. Returns tuple `{{value, Item}, Q2}`, where `Item` is the item removed and `Q2` is the resulting queue. If `Q1` is empty, 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 queue `Q1`. Returns tuple `{{value, Item}, Q2}`, where `Item` is the item removed and `Q2` is the new queue. If `Q1` is empty, tuple `{empty, Q1}` is returned.

`reverse(Q1 :: queue(Item)) -> Q2 :: queue(Item)`

Returns a queue `Q2` containing 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`.

`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 becomes the head of the list.

Extended API

Exports

`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.

`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.

`peek(Q :: queue(Item)) -> empty | {value, Item}`

Returns 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 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`.

daeh(Q :: *queue*(Item)) -> Item

Returns the tail item of queue `Q`.

Fails with reason `empty` if `Q` is empty.

head(Q :: *queue*(Item)) -> Item

Returns `Item` from the head of queue `Q`.

Fails with reason `empty` if `Q` is empty.

init(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.

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.

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)

Returns a queue `Q2` that is the result of removing the tail 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`.

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.

rand

Erlang module

This module provides a pseudo random number generator. The module contains a number of algorithms. The uniform distribution algorithms use the **xoroshiro116+ and xorshift1024* algorithms by Sebastiano Vigna.** The normal distribution algorithm uses the **Ziggurat Method by Marsaglia and Tsang** on top of the uniform distribution algorithm.

For some algorithms, jump functions are provided for generating non-overlapping sequences for parallel computations. The jump functions perform calculations equivalent to perform a large number of repeated calls for calculating new states.

The following algorithms are provided:

`exrop`

Xoroshiro116+, 58 bits precision and period of 2^116-1

Jump function: equivalent to 2^64 calls

`exs1024s`

Xorshift1024*, 64 bits precision and a period of 2^1024-1

Jump function: equivalent to 2^512 calls

`exsp`

Xorshift116+, 58 bits precision and period of 2^116-1

Jump function: equivalent to 2^64 calls

This is a corrected version of the previous default algorithm, that now has been superseded by Xoroshiro116+ (`exrop`). Since there is no native 58 bit rotate instruction this algorithm executes a little (say < 15%) faster than `exrop`. See the **algorithms' homepage**.

The default algorithm is `exrop` (Xoroshiro116+). If a specific algorithm is required, ensure to always use *`seed/1`* to initialize the state.

Undocumented (old) algorithms are deprecated but still implemented so old code relying on them will produce the same pseudo random sequences as before.

Note:

There were a number of problems in the implementation of the now undocumented algorithms, which is why they are deprecated. The new algorithms are a bit slower but do not have these problems:

Uniform integer ranges had a skew in the probability distribution that was not noticable for small ranges but for large ranges less than the generator's precision the probability to produce a low number could be twice the probability for a high.

Uniform integer ranges larger than or equal to the generator's precision used a floating point fallback that only calculated with 52 bits which is smaller than the requested range and therefore were not all numbers in the requested range even possible to produce.

Uniform floats had a non-uniform density so small values i.e less than 0.5 had got smaller intervals decreasing as the generated value approached 0.0 although still uniformly distributed for sufficiently large subranges. The new algorithms produces uniformly distributed floats on the form N * 2.0^(-53) hence equally spaced.

Every time a random number is requested, a state is used to calculate it and a new state is produced. The state can either be implicit or be an explicit argument and return value.

The functions with implicit state use the process dictionary variable `rand_seed` to remember the current state.

If a process calls *`uniform/0`* or *`uniform/1`* without setting a seed first, *`seed/1`* is called automatically with the default algorithm and creates a non-constant seed.

The functions with explicit state never use the process dictionary.

Examples:

Simple use; creates and seeds the default algorithm with a non-constant seed if not already done:

```
R0 = rand:uniform(),
R1 = rand:uniform(),
```

Use a specified algorithm:

```
_ = rand:seed(exs1024s),
R2 = rand:uniform(),
```

Use a specified algorithm with a constant seed:

```
_ = rand:seed(exs1024s, {123, 123534, 345345}),
R3 = rand:uniform(),
```

Use the functional API with a non-constant seed:

```
S0 = rand:seed_s(exrop),
{R4, S1} = rand:uniform_s(S0),
```

Create a standard normal deviate:

```
{SND0, S2} = rand:normal_s(S1),
```

Create a normal deviate with mean -3 and variance 0.5:

```
{ND0, S3} = rand:normal_s(-3, 0.5, S2),
```

Note:

The builtin random number generator algorithms are not cryptographically strong. If a cryptographically strong random number generator is needed, use something like *`crypto:rand_seed/0`*.

For all these generators the lowest bit(s) has got a slightly less random behaviour than all other bits. 1 bit for `exrop` (and `exsp`), and 3 bits for `exs1024s`. See for example the explanation in the **Xoroshiro128+** generator source code:

```
Beside passing BigCrush, this generator passes the PractRand test suite
up to (and included) 16TB, with the exception of binary rank tests,
which fail due to the lowest bit being an LFSR; all other bits pass all
tests. We suggest to use a sign test to extract a random Boolean value.
```

If this is a problem; to generate a boolean use something like this:

```
(rand:uniform(16) > 8)
```

And for a general range, with `N = 1` for `exrop`, and `N = 3` for `exs1024s`:

```
(((rand:uniform(Range bsl N) - 1) bsr N) + 1)
```

The floating point generating functions in this module waste the lowest bits when converting from an integer so they avoid this snag.

Data Types

```
builtin_alg() =
    exs64 | exsplus | exsp | exs1024 | exs1024s | exrop
alg() = builtin_alg() | atom()
alg_handler() =
    #{type := alg(),
      bits => integer() >= 0,
      weak_low_bits => integer() >= 0,
      max => integer() >= 0,
      next :=
          fun((alg_state()) -> {integer() >= 0, alg_state()}),
      uniform => fun((state()) -> {float(), state()}),
      uniform_n =>
          fun((integer() >= 1, state()) -> {integer() >= 1, state()}),
      jump => fun((state()) -> state())}
alg_state() =
    exs64_state() |
    exsplus_state() |
    exs1024_state() |
    exrop_state() |
    term()
state() = {alg_handler(), alg_state()}
```

Algorithm-dependent state.

`export_state() = {alg(), alg_state()}`

Algorithm-dependent state that can be printed or saved to file.

`exs64_state()`

Algorithm specific internal state

`exsplus_state()`

Algorithm specific internal state

`exs1024_state()`

Algorithm specific internal state

`exrop_state()`

Algorithm specific internal state

Exports

`export_seed() -> undefined | export_state()`

Returns the random number state in an external format. To be used with `seed/1`.

`export_seed_s(State :: state()) -> export_state()`

Returns the random number generator state in an external format. To be used with `seed/1`.

`jump() -> NewState :: state()`

Returns the state after performing jump calculation to the state in the process dictionary.

This function generates a `not_implemented` error exception when the jump function is not implemented for the algorithm specified in the state in the process dictionary.

`jump(State :: state()) -> NewState :: state()`

Returns the state after performing jump calculation to the given state.

This function generates a `not_implemented` error exception when the jump function is not implemented for the algorithm specified in the state.

`normal() -> float()`

Returns a standard normal deviate float (that is, the mean is 0 and the standard deviation is 1) and updates the state in the process dictionary.

`normal(Mean :: number(), Variance :: number()) -> float()`

Returns a normal N(Mean, Variance) deviate float and updates the state in the process dictionary.

`normal_s(State :: state()) -> {float(), NewState :: state()}`

Returns, for a specified state, a standard normal deviate float (that is, the mean is 0 and the standard deviation is 1) and a new state.

```
normal_s(Mean :: number(),
         Variance :: number(),
         State0 :: state()) ->
            {float(), NewS :: state()}
```

Returns, for a specified state, a normal N(Mean, Variance) deviate float and a new state.

```
seed(AlgOrStateOrExpState ::
         builtin_alg() | state() | export_state()) ->
        state()
```

Seeds random number generation with the specifed algorithm and time-dependent data if `AlgOrStateOrExpState` is an algorithm.

Otherwise recreates the exported seed in the process dictionary, and returns the state. See also `export_seed/0`.

```
seed(Alg :: builtin_alg(),
     Seed :: {integer(), integer(), integer()}) ->
```

state()

Seeds random number generation with the specified algorithm and integers in the process dictionary and returns the state.

seed_s(AlgOrStateOrExpState ::
_builtin_alg() | state() | export_state()) ->
state()

Seeds random number generation with the specifed algorithm and time-dependent data if `AlgOrStateOrExpState` is an algorithm.

Otherwise recreates the exported seed and returns the state. See also *`export_seed/0`*.

seed_s(Alg :: _builtin_alg(),
Seed :: {integer(), integer(), integer()}) ->
state()

Seeds random number generation with the specified algorithm and integers and returns the state.

uniform() -> X :: float()

Returns a random float uniformly distributed in the value range `0.0 =< X < 1.0` and updates the state in the process dictionary.

The generated numbers are on the form N * 2.0^(-53), that is; equally spaced in the interval.

Warning:

This function may return exactly `0.0` which can be fatal for certain applications. If that is undesired you can use `(1.0 - rand:uniform())` to get the interval `0.0 < X =< 1.0`.

If neither endpoint is desired you can test and re-try like this:

```
my_uniform() ->
    case rand:uniform() of
        0.0 -> my_uniform();
 X -> X
    end
end.
```

uniform(N :: integer() >= 1) -> X :: integer() >= 1

Returns, for a specified integer `N >= 1`, a random integer uniformly distributed in the value range `1 =< X =< N` and updates the state in the process dictionary.

uniform_s(State :: _state()_) -> {X :: float(), NewState :: _state()_}

Returns, for a specified state, random float uniformly distributed in the value range `0.0 =< X < 1.0` and a new state.

The generated numbers are on the form N * 2.0^(-53), that is; equally spaced in the interval.

Warning:

This function may return exactly `0.0` which can be fatal for certain applications. If that is undesired you can use `(1.0 - rand:uniform(State))` to get the interval `0.0 < X =< 1.0`.

If neither endpoint is desired you can test and re-try like this:

```
my_uniform(State) ->
    case rand:uniform(State) of
        {0.0, NewState} -> my_uniform(NewState);
 Result -> Result
    end
end.
```

`uniform_s(N :: integer() >= 1, State :: state()) ->`
`{X :: integer() >= 1, NewState :: state()}`

Returns, for a specified integer `N >= 1` and a state, a random integer uniformly distributed in the value range `1 =< X =< N` and a new state.

random

Erlang module

This module provides a random number generator. The method is attributed to B.A. Wichmann and I.D. Hill in 'An efficient and portable pseudo-random number generator', Journal of Applied Statistics. AS183. 1982. Also Byte March 1987.

The 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 is 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.

Note:

This random number generator is not cryptographically strong. If a strong cryptographic random number generator is needed, use one of functions in the *`crypto`* module, for example, *`crypto:strong_rand_bytes/1`*.

Note:

The improved *`rand`* module is to be used instead of this module.

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(SValue) -> undefined | ran()`

Types:

`SValue = {A1, A2, A3} | integer()`
`A1 = A2 = A3 = integer()`

`seed({A1, A2, A3})` is equivalent to `seed(A1, A2, A3)`.

`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.

The following is an easy way of obtaining a unique value to seed with:

```
random:seed(erlang:phash2([node()]),
            erlang:monotonic_time(),
            erlang:unique_integer())
```

For details, see *erlang:phash2/1*, *erlang:node/0*, *erlang:monotonic_time/0*, and *erlang:unique_integer/0*.

seed0() -> *ran()*

Returns the default state.

uniform() -> float()

Returns a random float uniformly distributed between `0.0` and `1.0`, updating the state in the process dictionary.

uniform(N) -> integer() >= 1

Types:

N = integer() >= 1

Returns, for a specified integer `N >= 1`, 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()*

Returns, for a specified state, 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()*

Returns, for a specified integer `N >= 1` and a state, 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 Erlang/OTP R15. Upgrading to R15 breaks applications that expect a specific output for a specified seed. The output is still deterministic number series, but different compared to releases older than R15. Seed `{0,0,0}` does, 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 matching algorithms of the library are 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. Currently PCRE version 8.40 (release date 2017-01-11) is used. The sections of the PCRE documentation that 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 extra backslash, that is, "\\".

Data Types

`mp() = {re_pattern, term(), term(), term(), term()}`

Opaque data type containing a compiled regular expression. `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 can change in future Erlang/OTP 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

`version() -> binary()`

The return of this function is a string with the PCRE version of the system that was used in the Erlang/OTP compilation.

`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}
```

Compiles a regular expression, with the syntax described below, into an internal format to be used later as a parameter to *`run/2`* and *`run/3`*.

Compiling the regular expression before matching is useful if the same expression is to be used in matching against multiple subjects during the lifetime of the program. Compiling once and executing many times is far more efficient than compiling each time one wants to match.

When option `unicode` is specified, the regular expression is to be specified as a valid Unicode `charlist()`, otherwise as any valid `iodata()`.

Options:

`unicode`

The regular expression is specified as a Unicode `charlist()` and the resulting regular expression code is to be run against a valid Unicode `charlist()` subject. Also consider option `ucp` 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 searched (the "subject string"). This effect can also be achieved by appropriate constructs in the pattern itself.

`caseless`

Letters in the pattern match both uppercase and lowercase letters. It is equivalent to Perl option `/i` and 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). This option is ignored if option `multiline` is specified. There is no equivalent option in Perl, and it cannot be set within a pattern.

`dotall`

A dot in the pattern matches all characters, including those indicating newline. Without it, a dot does not match when the current position is at a newline. This option is equivalent to Perl option `/s` and it can be changed within a pattern by a `(?s)` option setting. A negative class, such as `[^a]`, always matches newline characters, independent of the setting of this option.

`extended`

If this option is set, most white space characters in the pattern are totally ignored except when escaped or inside a character class. However, white space is not allowed within sequences such as `(?>` that introduce various parenthesized subpatterns, nor within a numerical quantifier such as `{1,3}`. However, ignorable white space is permitted between an item and a following quantifier and between a quantifier and a following + that indicates possessiveness.

White space did not used to include the VT character (code 11), because Perl did not treat this character as white space. However, Perl changed at release 5.18, so PCRE followed at release 8.34, and VT is now treated as white space.

This also causes characters between an unescaped # outside a character class and the next newline, inclusive, to be ignored. This is equivalent to Perl's `/x` option, and it can be changed within a pattern by a `(?x)` option setting.

With this option, comments inside complicated patterns can be included. However, notice that this applies only to data characters. Whitespace characters can never appear within special character sequences in a pattern, for example within sequence `(?(` that introduces a conditional subpattern.

`firstline`

An unanchored pattern is required to match before or at the first newline in the subject string, although the matched text can continue over the newline.

`multiline`

By default, PCRE treats the subject string as consisting of a single line of characters (even if it 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 option `dollar_endonly` is specified). This is the same as in Perl.

When this option is specified, 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 option `/m` and 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 is followed by `?:`. Named parentheses can still be used for capturing (and they acquire numbers in the usual way). There is no equivalent 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. More details of named subpatterns are provided below.

`ungreedy`

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}`

Overrides 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 is to be recognized.

`any`

Any of the newline sequences above, and 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, and so on, the default).

`no_start_optimize`

Disables optimization that can 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 "A" and never realize that the (*COMMIT) instruction is to have made the matching fail. This option is only relevant if you use "start-of-pattern items", as discussed in section *PCRE Regular Expression Details*.

`ucp`

Specifies that Unicode character properties are to 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 cannot be combined with option `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`

Takes a compiled regular expression and an item, and returns the relevant data from the regular expression. The only supported item is `namelist`, which returns the tuple `{namelist, [binary()]}`, containing the names of all (unique) named subpatterns in the regular expression. For example:

```
1> {ok,MP} = re:compile("(?<A>A)|(?<B>B)|(?<C>C)").
{ok,{re_pattern,3,0,0,
                <<69,82,67,80,119,0,0,0,0,0,0,0,1,0,0,0,255,255,255,255,
                  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,
                <<69,82,67,80,119,0,0,0,0,0,8,0,1,0,0,0,255,255,255,255,
                  255,255,...>>}}
4> re:inspect(MPD,namelist).
{namelist,[<<"B">>,<<"C">>]}
```

Notice 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 specified as an option to *`run/3`*. You can therefore create a name-to-value mapping from the result of `run/3` like this:

```
1> {ok,MP} = re:compile("(?<A>A)|(?<B>B)|(?<C>C)").
{ok,{re_pattern,3,0,0,
                <<69,82,67,80,119,0,0,0,0,0,0,0,1,0,0,0,255,255,255,255,
                  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">>,<<"A">>},{<<"B">>,<<>>},{<<"C">>,<<>>}]
```

`replace(Subject, RE, Replacement) -> iodata() | `*`unicode:charlist()`*

Types:

`Subject = iodata() | `*`unicode:charlist()`*

`RE = `*`mp()`* **` | iodata()`

`Replacement = iodata() | `*`unicode:charlist()`*

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 |
    {match_limit, integer() >= 0} |
    {match_limit_recursion, integer() >= 0} |
    bsr_unicode |
    {return, ReturnType} |
    CompileOpt
ReturnType = iodata | list | binary
CompileOpt = compile_option()
NLSpec = cr | crlf | lf | anycrlf | any
```

Replaces the matched part of the `Subject` string with the contents of `Replacement`.

The permissible options are the same as for *run/3*, except that option `capture` 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}`. If a binary is desired, specify `{return, binary}`.

As in function `run/3`, an `mp()` compiled with option `unicode` requires `Subject` to be a Unicode `charlist()`. If compilation is done implicitly and the `unicode` compilation option is specified to this function, both the regular expression and `Subject` are to specified 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), \gN, or \g{N}, resulting in the subexpression number N, is inserted in the result. If no subexpression with that number is generated by the regular expression, nothing is inserted.

To insert an & or a \ in the result, precede it with a \. Notice that Erlang already gives a special meaning to \ in literal strings, so a single \ must be written as `"\\"` and therefore a double \ as `"\\\\"`.

Example:

```
re:replace("abcd","c","[&]",[{return,list}]).
```

gives

```
"ab[c]d"
```

while

```
re:replace("abcd","c","[\\&]",[{return,list}]).
```

gives

```
"ab[&]d"
```

As with `run/3`, compilation errors raise the `badarg` exception. *`compile/2`* can be used to get more information about the error.

`run(Subject, RE) -> {match, Captured} | nomatch`

Types:

`Subject = iodata() | unicode:charlist()`
`RE = mp() | iodata()`
`Captured = [CaptureData]`
`CaptureData = {integer(), integer()}`

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*.

```
Captured = [CaptureData] | [[CaptureData]]
CaptureData =
    {integer(), integer()} | ListConversionData | binary()
ListConversionData =
    string() |
    {error, string(), binary()} |
    {incomplete, string(), binary()}
ErrType =
    match_limit | match_limit_recursion | {compile, CompileErr}
CompileErr =
    {ErrString :: string(), Position :: integer() >= 0}
```

Executes a regular expression matching, and returns `match/{match, Captured}` or `nomatch`. The regular expression can be specified either as `iodata()` in which case it is automatically compiled (as by `compile/2`) and executed, or as a precompiled `mp()` in which case it is executed against the subject directly.

When compilation is involved, exception `badarg` is thrown if a compilation error occurs. Call `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 following options:

- `anchored`
- `{capture, ValueSpec}/{capture, ValueSpec, Type}`
- `global`
- `{match_limit, integer() >= 0}`
- `{match_limit_recursion, integer() >= 0}`
- `{newline, NLSpec}`
- `notbol`
- `notempty`
- `notempty_atstart`
- `noteol`

- `{offset, integer() >= 0}`
- `report_errors`

Otherwise all options valid for function `compile/2` are also allowed. Options allowed both for compilation and execution of a match, namely `anchored` and `{newline, NLSpec}`, affect both the compilation and execution if present together with a non-precompiled regular expression.

If the regular expression was previously compiled with option `unicode`, `Subject` is to be provided as a valid Unicode `charlist()`, otherwise any `iodata()` will do. If compilation is involved and option `unicode` is specified, both `Subject` and the regular expression are to be specified as valid Unicode `charlists()`.

`{capture, ValueSpec}/{capture, ValueSpec, Type}` defines what to return from the function upon successful matching. The `capture` tuple can 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 options are described in detail below.

If the capture options describe that no substring capturing is to be done (`{capture, none}`), the function returns the single atom `match` upon successful matching, otherwise the tuple `{match, ValueList}`. Disabling capturing can be done either by specifying `none` or an empty list as `ValueSpec`.

Option `report_errors` adds the possibility that an error tuple is returned. The tuple either indicates a matching error (`match_limit` or `match_limit_recursion`), or a compilation error, where the error tuple has the format `{error, {compile, CompileErr}}`. Notice that if option `report_errors` is not specified, the function never returns error tuples, but reports compilation errors as a `badarg` exception and failed matches because of exceeded match limits simply as `nomatch`.

The following options are relevant for execution:

`anchored`

Limits `run/3` to matching at the first matching position. If a pattern was compiled with `anchored`, or turned out to be anchored by virtue of its contents, it cannot be made unanchored at matching time, hence there is no `unanchored` option.

`global`

Implements global (repetitive) search (flag `g` in Perl). Each match is returned as a separate `list()` containing the specific match and any matching subexpressions (or as specified by option `capture`. The `Captured` part of the return value is hence a `list()` of `list()`s when this option is specified.

The interaction of option `global` with a regular expression that matches an empty string surprises some users. When option `global` is specified, `run/3` handles empty matches in the same way as Perl: a zero-length match at any point is also retried with options `[anchored, notempty_atstart]`. If that search gives a result of length > 0, the result is included. Example:

```
re:run("cat","(|at)",[global]).
```

The following matchings are performed:

At offset `0`

The regular expression `(|at)` first match at the initial position of string `cat`, giving the result set `[{0,0}, {0,0}]` (the second `{0,0}` is because of the subexpression marked by the parentheses). As the length of the match is 0, we do not advance to the next position yet.

At offset `0` with `[anchored, notempty_atstart]`

The search is retried with options `[anchored, notempty_atstart]` at the same position, which does not give any interesting result of longer length, so the search position is advanced to the next character (`a`).

At offset `1`

The search results in `[{1,0},{1,0}]`, so this search is also repeated with the extra options.

At offset `1` with `[anchored, notempty_atstart]`

Alternative `ab` is found and the result is [{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 once again matches the empty string, giving `[{3,0},{3,0}]`.

At offset `1` with `[anchored, notempty_atstart]`

This gives no result of length > 0 and we are at the last position, so the global search is complete.

The result of the call is:

```
{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 specified. If alternatives in the pattern exist, they are tried. If all the alternatives match the empty string, the entire match fails.

Example:

If the following pattern is applied to a string not beginning with "a" or "b", it would normally match the empty string at the start of the subject:

```
a?b?
```

With option `notempty`, this match is invalid, so `run/3` searches further into the string for occurrences of "a" or "b".

`notempty_atstart`

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 modifier `/g`. The Perl behavior can be emulated 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`

Specifies that the first character of the subject string is not the beginning of a line, so the circumflex metacharacter is not to 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.

`noteol`

Specifies that the end of the subject string is not the end of a line, so the dollar metacharacter is not to 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`

Gives better control of the error handling in `run/3`. When specified, compilation errors (if the regular expression is not already compiled) and runtime errors are explicitly returned as an error tuple.

The following are the possible runtime errors:

`match_limit`

The PCRE library sets a limit on how many times the internal match function can be called. Defaults to 10,000,000 in the library compiled for Erlang. If `{error, match_limit}` is returned, the execution of the regular expression has reached this limit. This is normally to be regarded as a `nomatch`, which is the default return value when this occurs, but by specifying `report_errors`, you are informed when the match fails because of 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 defaults to 10,000,000 as well. Notice that as long as the `match_limit` and `match_limit_default` values are kept at the default values, the `match_limit_recursion` error cannot occur, as the `match_limit` error occurs before that (each recursive call is also a call, but not conversely). Both limits can however be changed, either by setting limits directly in the regular expression string (see section *PCRE Regular Eexpression Details*) or by specifying options to `run/3`.

It is important to understand that what is referred to as "recursion" when limiting matches is not recursion on the C stack of the Erlang machine or on the Erlang process stack. The PCRE version compiled into the Erlang VM uses machine "heap" memory to store values that must be kept over recursion in regular expression matches.

`{match_limit, integer() >= 0}`

Limits the execution time of a match in an implementation-specific way. It is described as follows 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 10,000,000 is compiled into the Erlang VM.

Note:

This option does in no way affect the execution of the Erlang VM in terms of "long running BIFs". `run/3` always gives 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}`

Limits the execution time and memory consumption of a match in an implementation-specific way, very similar to `match_limit`. It is described as follows 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 a smaller number
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 VM uses a PCRE library where heap memory is used when regular expression match recursion occurs. This therefore limits the use of machine heap, not C stack.

Specifying a lower value can result in matches with deep recursion failing, when they should have matched:

```
1> re:run("aaaaaaaaaaaaaz","(a+)*z").
{match,[{0,14},{0,13}]}
2> re:run("aaaaaaaaaaaaaz","(a+)*z",[{match_limit_recursion,5}]).
nomatch
3> re:run("aaaaaaaaaaaaaz","(a+)*z",[{match_limit_recursion,5},report_errors]).
{error,match_limit_recursion}
```

This option and option `match_limit` are only to be used in rare cases. Understanding of the PCRE library internals is recommended before tampering with these limits.

`{offset, integer() >= 0}`

Start matching at the offset (position) specified in the subject string. The offset is zero-based, so that the default is `{offset,0}` (all of the subject string).

`{newline, NLSpec}`

Overrides 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 is be recognized.

`any`

Any of the newline sequences above, and 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 the compilation option.)

`bsr_unicode`

Specifies specifically that \R is to match all the Unicode newline characters (including CRLF, and so on, the default). (Overrides the compilation option.)

`{capture, ValueSpec}`/`{capture, ValueSpec, Type}`

Specifies which captured substrings are returned and in what format. By default, `run/3` captures all of the matching part of the substring and 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, specified as `{Offset,Length}` pairs (the `index` `Type` of capturing).

As an example of the default behavior, the following call returns, as first and only captured string, the matching part of the subject ("abcd" in the middle) as an index pair `{3,4}`, where character positions are zero-based, just as in offsets:

```
re:run("ABCabcdABC","abcd",[]).
```

The return value of this call is:

```
{match,[{3,4}]}
```

Another (and quite common) case is where the regular expression matches all of the subject:

```
re:run("ABCabcdABC",".*abcd.*",[]).
```

Here the return value correspondingly points out all of the string, beginning at index 0, and it is 10 characters long:

```
{match,[{0,10}]}
```

If the regular expression contains capturing subpatterns, like in:

```
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 gives the first return value in the list and the remaining subpatterns are 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. `ValueSpec` can either be an atom describing a predefined set of return values, or a list containing the indexes or the names of specific subpatterns to return.

The following are the predefined sets of subpatterns:

`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 specified. The list of all names can also be retrieved with *`inspect/2`*.

`first`

Only the first captured subpattern, which is always the complete matching part of the subject. All explicitly captured subpatterns are discarded.

`all_but_first`

All but the first matching subpattern, that is, 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 are interested in is in an explicitly captured subpattern. If the return type is `list` or `binary`, not returning subpatterns you are not interested in is a good way to optimize.

`none`

Returns no matching subpatterns, gives 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 on. 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 string "ABCabcdABC", capturing only the "abcd" part (the first explicit subpattern):

```
re:run("ABCabcdABC",".*(abcd).*",[{capture,[1]}]).
```

The call gives the following result, as the first explicitly captured subpattern is "(abcd)", matching "abcd" in the subject, at (zero-based) position 3, of length 4:

```
{match,[{3,4}]}
```

Consider the same regular expression, but with the subpattern explicitly named 'FOO':

```
".*(?<FOO>abcd).*"
```

With this expression, we could still give the index of the subpattern with the following call:

```
re:run("ABCabcdABC",".*(?<FOO>abcd).*",[{capture,[1]}]).
```

giving the same result as before. But, as the subpattern is named, we can also specify its name in the value list:

```
re:run("ABCabcdABC",".*(?<FOO>abcd).*",[{capture,['FOO']}]).
```

This would give the same result as the earlier examples, namely:

```
{match,[{3,4}]}
```

The values list can 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 with no corresponding subpattern in the regular expression, 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.

`Type` can be one of the following:

`index`

Returns 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 *`erlang:iolist_to_binary/1`* or *`unicode:characters_to_binary/2`* before matching). Notice that option `unicode` results in **byte-oriented** indexes in a (possibly virtual) **UTF-8 encoded** binary. A byte index tuple `{0,2}` can therefore represent one or two characters when `unicode` is in effect. This can seem counter-intuitive, but has been deemed the most effective and useful way to do it. To return lists instead can result in simpler code if that is desired. This return type is the default.

`list`

Returns matching substrings as lists of characters (Erlang `string()`s). It option `unicode` 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 can result in the same types of tuples that *`unicode:characters_to_list/2`* can return, namely three-tuples with 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`

Returns matching substrings as binaries. If option `unicode` is used, these binaries are in UTF-8. If the \C sequence is used together with `unicode`, the binaries can 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 following regular expression:

```
".*((?<FOO>abdd)|a(..d)).*"
```

There are three explicitly capturing subpatterns, where the opening parenthesis position determines the order in the result, hence `((?<FOO>abdd)|a(..d))` is subpattern index 1, `(?<FOO>abdd)` is subpattern index 2, and `(..d)` is subpattern index 3. When matched against the following string:

```
"ABCabcdABC"
```

the subpattern at index 2 does not match, as "abdd" is not present in the string, but the complete pattern matches (because of the alternative `a(..d)`). The subpattern at index 2 is therefore unassigned and the default return value is:

```
{match,[{0,10},{3,4},{-1,0},{4,3}]}
```

Setting the capture `Type` to `binary` gives:

```
{match,[<<"ABCabcdABC">>,<<"abcd">>,<<>>,<<"bcd">>]}
```

Here the empty binary (`<<>>`) represents the unassigned subpattern. In the `binary` case, some information about the matching is therefore lost, as `<<>>` can also 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 option `global` is speciified, the `capture` specification affects each match separately, so that:

```
re:run("cacb","c(a|b)",[global,{capture,[1],list}]).
```

gives

```
{match,[["a"],["b"]]}
```

For a descriptions of options only affecting the compilation step, see *`compile/2`*.

`split(Subject, RE) -> SplitList`

Types:

Subject = iodata() | *unicode:charlist()*
RE = *mp()* | iodata()
SplitList = [iodata() | *unicode:charlist()*]

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*.

SplitList = [RetData] | [GroupedRetData]
GroupedRetData = [RetData]
RetData = iodata() | *unicode:charlist()* | binary() | list()

Splits the input into parts by finding tokens according to the regular expression supplied. The splitting is basically done by running a global regular expression match and dividing the initial string wherever a match occurs. The matching part of the string is removed from the output.

As in *run/3*, an `mp()` compiled with option `unicode` requires `Subject` to be a Unicode `charlist()`. If compilation is done implicitly and the `unicode` compilation option is specified to this function, both the regular expression and `Subject` are to be specified as valid Unicode `charlist()`s.

The result is given as a list of "strings", the preferred data type specified in option `return` (default `iodata`).

If subexpressions are specified in the regular expression, the matching subexpressions are returned in the resulting list as well. For example:

```
re:split("Erlang","[ln]",[{return,list}]).
```

gives

```
["Er","a","g"]
```

while

```
re:split("Erlang","([ln])",[{return,list}]).
```

gives

```
["Er","l","a","n","g"]
```

The text matching the subexpression (marked by the parentheses in the regular expression) is inserted in the result list where it was found. This means that concatenating the result of a split where the whole regular expression is a single subexpression (as in the last example) always results in the original string.

As there is no matching subexpression for the last part in the example (the "g"), nothing is inserted after that. To make the group of strings and the parts matching the subexpressions more obvious, one can use option `group`, 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 first matched 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. As the subexpression is bound to substring "n" in this case, the "n" is inserted into this group. The last group consists of the remaining 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}]).
```

gives

```
["Er","an",[]]
```

as the matching of the "g" in the end of the string leaves an empty rest, which is also returned. This behavior differs from the default behavior 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]).
```

gives

```
["Er","an"]
```

The "trim" option says; "give me as many parts as possible except the empty ones", which sometimes can be useful. You can also specify how many parts you want, by specifying `{parts,`N`}`:

```
re:split("Erlang","[lg]",[{return,list},{parts,2}]).
```

gives

```
["Er","ang"]
```

Notice that the last part is "ang", not "an", as splitting was specified 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

```
re:split("Erlang","[lg]",[{return,list},{parts,4}]).
```

gives 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 option `trim`. If subexpressions are captured, empty subexpressions matched at the end are also stripped from the result if `trim` or `{parts,0}` is specified.

The `trim` behavior corresponds exactly to the Perl default. `{parts,N}`, where N is a positive integer, corresponds exactly to the Perl behavior with a positive numerical third parameter. The default behavior of `split/3` corresponds to the Perl behavior when a negative integer is specified as the third parameter for the Perl routine.

Summary of options not previously described for function `run/3`:

`{return,ReturnType}`

Specifies how the parts of the original string are presented in the result list. Valid types:

`iodata`

The variant of `iodata()` that gives the least copying of data with the current implementation (often a binary, but do not 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 regular expression.

The return value from the function is in this case 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 is to be a positive integer for a specific maximum 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 behavior of the `split` built-in function in Perl.

Perl-Like Regular Expression Syntax

The following sections contain reference material for the regular expressions used by this module. The information is based on the PCRE documentation, with changes where this module behaves differently to the PCRE library.

PCRE Regular Expression Details

The syntax and semantics of the regular expressions supported by PCRE are described in detail in the following sections. Perl's regular expressions are described in its own documentation, and regular expressions in general are covered in many books, some with copious examples. Jeffrey Friedl's "Mastering Regular Expressions", published

by O'Reilly, covers regular expressions in great detail. This description of the PCRE 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

Some options that can be passed to `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 can appear, but they must all be together right at the start of the pattern string, and the letters must be in upper case.

UTF Support

Unicode support is basically UTF-8 based. To use Unicode characters, you either call `compile/2` or `run/3` with option `unicode`, or the pattern must start with one of these special sequences:

```
(*UTF8)
(*UTF)
```

Both options give the same effect, the input string is interpreted as UTF-8. Notice that with these instructions, the automatic conversion of lists to UTF-8 is not performed by the `re` functions. Therefore, using these sequences is not recommended. Add option `unicode` when running `compile/2` instead.

Some applications that allow their users to supply patterns can wish to restrict them to non-UTF data for security reasons. If option `never_utf` is set at compile time, (*UTF), and so on, are not allowed, and their appearance causes an error.

Unicode Property Support

The following is another special sequence that can appear at the start of a pattern:

```
(*UCP)
```

This has the same effect as setting option `ucp`: it causes sequences such as \d and \w to use Unicode properties to determine character types, instead of recognizing only characters with codes < 256 through a lookup table.

Disabling Startup Optimizations

If a pattern starts with `(*NO_START_OPT)`, it has the same effect as setting option `no_start_optimize` at compile time.

Newline Conventions

PCRE supports five conventions for indicating line breaks in strings: a single CR (carriage return) character, a single LF (line feed) character, the two-character sequence CRLF, any of the three preceding, and any Unicode newline sequence.

A newline convention can also be specified by starting a pattern string with one of the following five sequences:

(*CR)
: Carriage return

(*LF)
: Line feed

(*CRLF)
: >Carriage return followed by line feed

(*ANYCRLF)
: Any of the three above

(*ANY)
: All Unicode newline sequences

These override the default and the options specified to *`compile/2`*. For example, the following pattern changes the convention to CR:

```
(*CR)a.b
```

This pattern matches `a\nb`, as 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 behavior 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 section *Newline Sequences*. A change of the \R setting can be combined with a change of the newline convention.

Setting Match and Recursion Limits

The caller of *`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 following forms:

```
(*LIMIT_MATCH=d)
(*LIMIT_RECURSION=d)
```

Here d is any number of decimal digits. However, the value of the setting must be less than the value set by the caller of `run/3` for it to have any effect. That is, 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 default value for both the limits is 10,000,000 in the Erlang VM. Notice that the recursion limit does not 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.

Note that `LIMIT_MATCH` and `LIMIT_RECURSION` can only reduce the value of the limits set by the caller, not increase them.

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 following pattern matches a portion of a subject string that is identical to itself:

```
The quick brown fox
```

When caseless matching is specified (option `caseless`), letters are matched independently of case.

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of **metacharacters**, which do not stand for themselves but instead are interpreted in some special way.

Two sets of metacharacters exist: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized within square brackets. Outside square brackets, the metacharacters are as follows:

\
: General escape character with many uses

^
: Assert start of string (or line, in multiline mode)

$
: Assert end of string (or line, in multiline mode)

.
: Match any character except newline (by default)

[
: Start character class definition

|
: Start of alternative branch

(
: Start subpattern

)
: End subpattern

?
: Extends the meaning of (, also 0 or 1 quantifier, also quantifier minimizer

*
: 0 or more quantifiers

+
: 1 or more quantifier, also "possessive quantifier"

{
: Start min/max quantifier

Part of a pattern within square brackets is called a "character class". The following are the only metacharacters in a character class:

\
: General escape character

^
: Negate the class, but only if the first character

-
: Indicates character range

[
: Posix character class (only if followed by Posix syntax)

]
: Terminates the character class

The following sections describe the use of each metacharacter.

Backslash

The backslash character has many uses. First, if it is followed by a character that is not a number or a letter, it takes away any special meaning that a character can 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 if 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, write \\.

In `unicode` mode, only ASCII numbers and letters have any special meaning after a backslash. All other characters (in particular, those whose code points are > 127) are treated as literals.

If a pattern is compiled with option `extended`, whitespace in the pattern (other than in a character class) and characters between a # outside a character class and the next newline are ignored. An escaping backslash can be used to include a whitespace or # character as part of the pattern.

To remove the special meaning from a sequence of characters, put them between \Q and \E. This is different from Perl in that $ and @ are handled as literals in \Q...\E sequences in PCRE, while $ and @ cause variable interpolation in Perl. Notice the following examples:

```
Pattern            PCRE matches   Perl matches

\Qabc$xyz\E        abc$xyz        abc followed by the contents of $xyz
\Qabc\$xyz\E       abc\$xyz       abc\$xyz
\Qabc\E\$\Qxyz\E   abc$xyz        abc$xyz
```

The \Q...\E sequence is recognized both inside and outside character classes. 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, as 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. When a pattern is 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
: Line feed (hex 0A)

\r
Carriage return (hex 0D)
\t
Tab (hex 09)
\0dd
Character with octal code 0dd
\ddd
Character with octal code ddd, or back reference
\o{ddd..}
character with octal code ddd..
\xhh
Character with hex code hh
\x{hhh..}
Character with hex code hhh..

Note:

Note that \0dd is always an octal code, and that \8 and \9 are the literal characters "8" and "9".

The precise effect of \cx on ASCII characters is as follows: if x is a lowercase 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 > 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.

After \0 up to two further octal digits are read. If there are fewer than two digits, just those that are present are used. Thus the sequence \0\x\015 specifies two binary zeros followed by a CR character (code value 13). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

The escape \o must be followed by a sequence of octal digits, enclosed in braces. An error occurs if this is not the case. This escape is a recent addition to Perl; it provides way of specifying character code points as octal numbers greater than 0777, and it also allows octal numbers and back references to be unambiguously specified.

For greater clarity and unambiguity, it is best to avoid following \ by a digit greater than zero. Instead, use \o{} or \x{} to specify character numbers, and \g{} to specify back references. The following paragraphs describe the old, ambiguous syntax.

The handling of a backslash followed by a digit other than 0 is complicated, and Perl has changed in recent releases, causing PCRE also to change. Outside a character class, PCRE reads the digit and any following digits as a decimal number. If the number is < 8, 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 provided later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number following \ is > 7 and there have not been that many capturing subpatterns, PCRE handles \8 and \9 as the literal characters "8" and "9", and otherwise re-reads up to three octal digits following the backslash, and using them to generate a data character. Any subsequent digits stand for themselves. For example:

\040
Another way of writing an ASCII space
\40
The same, provided there are < 40 previous capturing subpatterns
\7
Always a back reference

\11
: Can be a back reference, or another way of writing a tab

\011
: Always a tab

\0113
: A tab followed by character "3"

\113
: Can be a back reference, otherwise the character with octal code 113

\377
: Can be a back reference, otherwise value 255 (decimal)

\81
: Either a back reference, or the two characters "8" and "1"

Notice that octal values >= 100 that are specified using this syntax must not be introduced by a leading zero, as no more than three octal digits are ever read.

By default, after \x that is not followed by {, 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 }. If a character other than a hexadecimal digit appears between \x{ and }, or if there is no terminating }, an error occurs.

Characters whose value is less than 256 can be defined by either of the two syntaxes for \x. There is no difference in the way they are handled. For example, \xdc is exactly the same as \x{dc}.

Constraints on character values

Characters that are specified using octal or hexadecimal numbers are limited to certain values, as follows:

8-bit non-UTF mode

: < 0x100

8-bit UTF-8 mode

: < 0x10ffff and a valid codepoint

Invalid Unicode codepoints are the range 0xd800 to 0xdfff (the so-called "surrogate" codepoints), and 0xffef.

Escape sequences in character classes

All the sequences that define a single character value can be used both inside and outside character classes. Also, 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 alternative syntax for referencing a subpattern as a "subroutine". Details are discussed later. Notice that \g{...} (Perl syntax) and \g<...> (Oniguruma syntax) are **not** synonymous. The former is a back reference and 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 whitespace character

\H
: Any character that is not a horizontal whitespace character

\s
: Any whitespace character

\S
: Any character that is not a whitespace character

\v
: Any vertical whitespace character

\V
: Any character that is not a vertical whitespace character

\w
: Any "word" character

\W
: Any "non-word" character

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, but PCRE does not support this.

Each pair of lowercase and uppercase 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 fail, as there is no character to match.

For compatibility with Perl, \s did not used to match the VT character (code 11), which made it different from the the POSIX "space" class. However, Perl added VT at release 5.18, and PCRE followed suit at release 8.34. The default \s characters are now HT (9), LF (10), VT (11), FF (12), CR (13), and space (32), which are defined as white space in the "C" locale. This list may vary if locale-specific matching is taking place. For example, in some locales the "non-breaking space" character (\xA0) is recognized as white space, and in others the VT character is not.

A "word" character is an underscore or any character that is a letter or a digit. By default, the definition of letters and digits is controlled by the PCRE low-valued character tables, in Erlang's case (and without option `unicode`), the ISO Latin-1 character set.

By default, in `unicode` mode, characters with values > 255, that is, all characters outside the ISO Latin-1 character set, 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 option `ucp` is set, the behavior 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} or \h or \v

\w
: Any character that matches \p{L} or \p{N} matches, plus underscore

The uppercase escapes match the inverse sets of characters. Notice that \d matches only decimal digits, while \w matches any Unicode digit, any Unicode letter, and underscore. Notice also that `ucp` affects \b and \B, as 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 in release 5.10. In contrast to the other sequences, which match only ASCII characters by default, these always match certain high-valued code points, regardless if `ucp` is set.

The following are the horizontal space characters:

U+0009
: Horizontal tab (HT)

U+0020
: Space

U+00A0
: Non-break space

U+1680
: Ogham space mark

U+180E
: Mongolian vowel separator

U+2000
: En quad

U+2001
: Em quad

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 following are the vertical space characters:

U+000A
: Line feed (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 code points < 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:

```
(?>\r\n|\n|\x0b|\f|\r|\x85)
```

This is an example of an "atomic group", details are provided below.

This particular group matches either the two-character sequence CR followed by LF, or one of the single characters LF (line feed, 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 more characters whose code points are > 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.

\R can be restricted to match only CR, LF, or CRLF (instead of the complete set of Unicode line endings) by setting option `bsr_anycrlf` either at compile time or when the pattern is matched. (BSR is an acronym for "backslash R".) This can be made the default when PCRE is built; if so, the other behavior can be requested through option `bsr_unicode`. These settings can also be specified by starting a pattern string with one of the following sequences:

(*BSR_ANYCRLF)
: CR, LF, or CRLF only

(*BSR_UNICODE)
: Any Unicode newline sequence

These override the default and the options specified to the compiling function, but they can themselves be overridden by options specified to a matching function. Notice that these special settings, which are not Perl-compatible, are recognized only at the very start of a pattern, and that they must be in upper case. If more than one of them is present, the last one is used. They can be combined with a change of newline convention; for example, a pattern can start with:

```
(*ANY)(*BSR_ANYCRLF)
```

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 more escape sequences that match characters with specific properties are available. When in 8-bit non-UTF-8 mode, these sequences are limited to testing characters whose code points are < 256, but they do work in this mode. The following are the extra escape sequences:

\p{**xx**}
: A character with property **xx**

\P{**xx**}
: A character without property **xx**

\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 currently not supported by PCRE. Notice that \P{Any} does not match any characters and 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 following is the current list of scripts:

- Arabic
- Armenian
- Avestan
- Balinese
- Bamum
- Bassa_Vah
- Batak
- Bengali
- Bopomofo
- Braille
- Buginese
- Buhid
- Canadian_Aboriginal
- Carian
- Caucasian_Albanian
- Chakma
- Cham
- Cherokee
- Common
- Coptic
- Cuneiform
- Cypriot
- Cyrillic
- Deseret
- Devanagari
- Duployan
- Egyptian_Hieroglyphs
- Elbasan
- Ethiopic
- Georgian
- Glagolitic
- Gothic

- Grantha
- Greek
- Gujarati
- Gurmukhi
- Han
- Hangul
- Hanunoo
- Hebrew
- Hiragana
- Imperial_Aramaic
- Inherited
- Inscriptional_Pahlavi
- Inscriptional_Parthian
- Javanese
- Kaithi
- Kannada
- Katakana
- Kayah_Li
- Kharoshthi
- Khmer
- Khojki
- Khudawadi
- Lao
- Latin
- Lepcha
- Limbu
- Linear_A
- Linear_B
- Lisu
- Lycian
- Lydian
- Mahajani
- Malayalam
- Mandaic
- Manichaean
- Meetei_Mayek
- Mende_Kikakui
- Meroitic_Cursive
- Meroitic_Hieroglyphs
- Miao
- Modi
- Mongolian
- Mro

- Myanmar
- Nabataean
- New_Tai_Lue
- Nko
- Ogham
- Ol_Chiki
- Old_Italic
- Old_North_Arabian
- Old_Permic
- Old_Persian
- Oriya
- Old_South_Arabian
- Old_Turkic
- Osmanya
- Pahawh_Hmong
- Palmyrene
- Pau_Cin_Hau
- Phags_Pa
- Phoenician
- Psalter_Pahlavi
- Rejang
- Runic
- Samaritan
- Saurashtra
- Sharada
- Shavian
- Siddham
- Sinhala
- Sora_Sompeng
- Sundanese
- Syloti_Nagri
- Syriac
- Tagalog
- Tagbanwa
- Tai_Le
- Tai_Tham
- Tai_Viet
- Takri
- Tamil
- Telugu
- Thaana
- Thai
- Tibetan

- Tifinagh
- Tirhuta
- Ugaritic
- Vai
- Warang_Citi
- Yi

Each character has exactly one Unicode general category property, specified by a two-letter acronym. 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. The following 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
: Lowercase letter

Lm
: Modifier letter

Lo
: Other letter

Lt
: Title case letter

Lu
: Uppercase 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, that is, 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 invalid in Unicode strings and so cannot be tested by PCRE. Perl does not support the Cs property.

The long synonyms for property names supported by Perl (such as \p{Letter}) are not supported by PCRE. It is not permitted to prefix any of these properties with "Is".

No character in the Unicode table has the Cn (unassigned) property. This property is instead 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 uppercase letters. This is different from the behavior of current versions of Perl.

Matching characters by Unicode property is not fast, as PCRE must do a multistage table lookup to find a character property. That is why the traditional escape sequences such as \d and \w do not use Unicode properties in PCRE by default. However, you can make them do so by setting option `ucp` 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 `(?>\PM\pM*)`. 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 PCRE releases later than 8.31, \X matches one of these clusters.

\X always matches at least one character. Then it decides whether to add more characters according to the following rules for ending a cluster:

- End at the end of the subject string.
- Do not end between CR and LF; otherwise end after any control character.
- Do not break Hangul (a Korean script) syllable sequences. Hangul characters are of five types: L, V, T, LV, and LVT. An L character can be followed by an L, V, LV, or LVT character. An LV or V character can be followed by a V or T character. An LVT or T character can be followed only by a T character.
- Do not end before extending characters or spacing marks. Characters with the "mark" property always have the "extend" grapheme breaking property.
- Do not end after prepend characters.
- Otherwise, end the cluster.

PCRE Additional Properties

In addition to the standard Unicode properties described earlier, PCRE supports four more that make it possible to convert traditional escape sequences, such as \w and \s to use Unicode properties. PCRE uses these non-standard, non-Perl properties internally when the `ucp` option is passed. However, they can also be used explicitly. The properties are as follows:

Xan

Any alphanumeric character. Matches characters that have either the L (letter) or the N (number) property.

Xps

Any Posix space character. Matches the characters tab, line feed, vertical tab, form feed, carriage return, and any other character that has the Z (separator) property.

Xsp

Any Perl space character. Matches the same as Xps, except that vertical tab is excluded.

Xwd

Any Perl "word" character. Matches the same characters as Xan, plus underscore.

Perl and POSIX space are now the same. Perl added VT to its space character set at release 5.18 and PCRE changed at release 8.34.

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; it used to exclude vertical tab, for Perl compatibility, but Perl changed, and so PCRE followed at release 8.34. 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 >= U+00A0, except for the surrogates U+D800 to U+DFFF. Notice that most base (ASCII) characters are excluded. (Universal Character Names are of the form \uHHHH or \UHHHHHHHH, where H is a hexadecimal digit. Notice 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 following pattern matches "foobar", but reports that it has matched "bar":

```
foo\Kbar
```

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 following pattern matches "foobar", the first substring is still set to "foo":

```
(foo)\Kbar
```

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. Note that when a pattern such as (?=ab\K) matches, the reported start of the match can be greater than the end of the match.

Simple Assertions

The final use of backslash is for certain simple assertions. An assertion specifies a condition that must 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 following are the backslashed assertions:

\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, and 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 (that is, 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 UTF mode, the meanings of \w and \W can be changed by setting option `ucp`. When this is done, it also affects \b and \B. PCRE and Perl do not have 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 options `notbol` or `noteol`, which affect only the behavior of the circumflex and dollar metacharacters. However, if argument `startoffset` of *`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 and at the very end, while \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 argument `startoffset` of `run/3`. It differs from \A when the value of `startoffset` is non-zero. By calling `run/3` multiple times with appropriate arguments, you can mimic the Perl option `/g`, and it is in this kind of implementation where \G can be useful.

Notice, however, that the PCRE interpretation of \G, as the start of the current match, is subtly different from Perl, which defines it as the end of the previous match. In Perl, these can be different when the previously matched string was empty. As PCRE does only one match at a time, it cannot reproduce this behavior.

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 to be 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 argument `startoffset` of *`run/3`* is non-zero, circumflex can never match if option `multiline` is unset. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex needs not to be the first character of the pattern if some alternatives are involved, but it is to 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). Notice however that it does not match the newline. Dollar needs not to be the last character of the pattern if some alternatives are involved, but it is to 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 option `dollar_endonly` at compile time. This does not affect the \Z assertion.

The meanings of the circumflex and dollar characters are changed if option `multiline` is set. When this is the case, a circumflex matches immediately after internal newlines and 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, and at the very end, when `multiline` is set. When newline is specified as the two-character sequence CRLF, isolated CR and LF characters do not indicate newlines.

For example, the pattern /^abc$/ matches the subject string "def\nabc" (where \n represents a newline) in multiline mode, but not otherwise. So, 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 argument **startoffset** of `run/3` is non-zero. Option `dollar_endonly` is ignored if `multiline` is set.

Notice that the sequences \A, \Z, and \z can be used to match the start and end of the subject in both modes. If all branches of a pattern start with \A, it is always anchored, regardless if `multiline` is set.

Full Stop (Period, Dot) and \N

Outside a character class, a dot in the pattern matches any 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, otherwise it matches all characters (including isolated CRs and LFs). When any Unicode line endings are recognized, dot does not match CR, LF, or any of the other line-ending characters.

The behavior of dot regarding newlines can be changed. If option `dotall` is set, a dot matches any 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 is that 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 option `PCRE_DOTALL`. That is, it matches any character except one that signifies the end of a line. Perl also uses \N to match characters by name but PCRE does not support this.

Matching a Single Data Unit

Outside a character class, the escape sequence \C matches any data unit, regardless if 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 to match individual bytes in UTF-8 mode, but it is unclear how it can usefully be used. As \C breaks up characters into individual data units, matching one unit with \C in a UTF mode means that the remaining string can start with a malformed UTF character. This has undefined results, as PCRE assumes that it deals with valid UTF strings.

PCRE does not allow \C to appear in lookbehind assertions (described below) in a UTF mode, as this would make it impossible to calculate the length of the lookbehind.

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 the following pattern, which can be used with a UTF-8 string (ignore whitespace and line breaks):

```
(?| (?=[\x00-\x7f])(\C) |
    (?=[\x80-\x{7ff}])(\C)(\C) |
    (?=[\x{800}-\x{ffff}])(\C)(\C)(\C) |
    (?=[\x{10000}-\x{1fffff}])(\C)(\C)(\C)(\C))
```

A group that starts with (?| resets the capturing parentheses numbers in each alternative (see section *Duplicate Subpattern Numbers*). 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 individual bytes of the character 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 option `PCRE_JAVASCRIPT_COMPAT` 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 is to 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 can 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 required as a member of the class, ensure that it is not the first character, or escape it with a backslash.

For example, the character class `[aeiou]` matches any lowercase vowel, while `[^aeiou]` matches any character that is not a lowercase vowel. Notice 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 > 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 uppercase and lowercase versions. For example, a caseless `[aeiou]` matches "A" and "a", and a caseless `[^aeiou]` does not match "A", but a caseful version would. In a UTF mode, PCRE always understands the concept of case for characters whose values are < 256, so caseless matching is always possible. For characters with higher values, the concept of case is supported only if PCRE is compiled with Unicode property support. If you want to use caseless matching in a UTF mode for characters >=, ensure that PCRE is compiled with Unicode property support and with UTF support.

Characters that can 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 options `PCRE_DOTALL` and `PCRE_MULTILINE` 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, or immediately after a range. For example, [b-d-z] matches letters in the range b to d, a hyphen character, or z.

The literal character "]" cannot be the end character of a range. A pattern such as [W-]46] is interpreted as a class of two characters ("W" and "-") followed by a literal string "46]", so it would match "W46]" or "-46]". However, if "]" is escaped with a backslash, it is interpreted as the end of range, so [W-\]46] is interpreted as a class containing a range followed by two other characters. The octal or hexadecimal representation of "]" can also be used to end a range.

An error is generated if a POSIX character class (see below) or an escape sequence other than one that defines a single character appears at a point where a range ending character is expected. For example, [z-\xff] is valid, but [A-\d] and [A-[:digit:]] are not.

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 [][\\^_`wxyzabc], matched caselessly. 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 > 255 only when it is compiled with Unicode property support.

The character escape sequences \d, \D, \h, \H, \p, \P, \s, \S, \v, \V, \w, and \W can 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, option `ucp` affects the meanings of \d, \s, \w and their uppercase partners, just as it does when they appear outside a character class, as described in section *Generic Character Types* earlier. 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 uppercase character types to specify a more restricted set of characters than the matching lowercase type. For example, class [^\W_] matches any letter or digit, but not underscore, while [\w] includes underscore. A positive character class is to be read as "something OR something OR ..." and a negative class as "NOT something AND NOT something AND NOT ...".

Only the following metacharacters are recognized in character classes:

- 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, or for a special compatibility feature; see the next two sections)
- 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, the following matches "0", "1", any alphabetic character, or "%":

```
[01[:alpha:]%]
```

The following are the supported class names:

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
: Lowercase letters

print
: Printing characters, including space

punct
: Printing characters, excluding letters, digits, and space

space
: Whitespace (the same as \s from PCRE 8.34)

upper
: Uppercase letters

word
: "Word" characters (same as \w)

xdigit
: Hexadecimal digits

The default "space" characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). If locale-specific matching is taking place, the list of space characters may be different; there may be fewer or more of them. "Space" used to be different to \s, which did not include VT, for Perl compatibility. However, Perl changed at release 5.18, and PCRE followed at release 8.34. "Space" and \s now match the same set of characters.

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, the following matches "1", "2", or any non-digit:

```
[12[:^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, characters with values > 255 do not match any of the Posix character classes. However, if option `PCRE_UCP` is passed to `pcre_compile()`, some of the classes are changed so that Unicode character properties are used. This is achieved by replacing certain Posix classes by other sequences, as follows:

[:alnum:]
Becomes **\p{Xan}**
[:alpha:]
Becomes **\p{L}**
[:blank:]
Becomes **\h**
[:digit:]
Becomes **\p{Nd}**
[:lower:]
Becomes **\p{Ll}**
[:space:]
Becomes **\p{Xps}**
[:upper:]
Becomes **\p{Lu}**
[:word:]
Becomes **\p{Xwd}**

Negated versions, such as [:^alpha:], use \P instead of \p. Three other POSIX classes are handled specially in UCP mode:

[:graph:]

This matches characters that have glyphs that mark the page when printed. In Unicode property terms, it matches all characters with the L, M, N, P, S, or Cf properties, except for:

U+061C

Arabic Letter Mark

U+180E

Mongolian Vowel Separator

U+2066 - U+2069

Various "isolate"s

[:print:]

This matches the same characters as [:graph:] plus space characters that are not controls, that is, characters with the Zs property.

[:punct:]

This matches all characters that have the Unicode P (punctuation) property, plus those characters whose code points are less than 128 that have the S (Symbol) property.

The other POSIX classes are unchanged, and match only characters with code points less than 128.

Compatibility Feature for Word Boundaries

In the POSIX.2 compliant library that was included in 4.4BSD Unix, the ugly syntax [[:<:]] and [[:>:]] is used for matching "start of word" and "end of word". PCRE treats these items as follows:

[[:<:]]

is converted to \b(?=\w)

[[:>:]]

is converted to \b(?<=\w)

Only these exact character sequences are recognized. A sequence such as [a[:<:]b] provokes error for an unrecognized POSIX class name. This support is not compatible with Perl. It is provided to help migrations from other environments, and is best not used in any new patterns. Note that \b matches at the start and the end of a word (see "Simple assertions" above), and in a Perl-style pattern the preceding or following character normally shows which is wanted, without the need for the assertions that are used above in order to give exactly the POSIX behaviour.

Vertical Bar

Vertical bar characters are used to separate alternative patterns. For example, the following pattern matches either "gilbert" or "sullivan":

```
gilbert|sullivan
```

Any number of alternatives can 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 that succeeds is used. If the alternatives are within a subpattern (defined in section *Subpatterns*), "succeeds" means matching the remaining main pattern and the alternative in the subpattern.

Internal Option Setting

The settings of the Perl-compatible options `caseless`, `multiline`, `dotall`, and `extended` can be changed from within the pattern by a sequence of Perl option letters enclosed between "(?" and ")". The option letters are as follows:

i
: For `caseless`

m
: For `multiline`

s
: For `dotall`

x
: For `extended`

For example, `(?im)` sets caseless, multiline matching. These options can also be unset by preceding the letter with a hyphen. 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.

An option change within a subpattern (see section *Subpatterns*) affects only that part of the subpattern that follows it. So, the following matches abc and aBc and no other strings (assuming `caseless` is not used):

```
(a(?i)b)c
```

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", although when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings occur at compile time. There would be some weird behavior otherwise.

Note:

Other PCRE-specific options can be set by the application when the compiling or matching functions are called. Sometimes the pattern can contain special leading sequences, such as (*CRLF), to override what the application has set or what has been defaulted. Details are provided in section *Newline Sequences* earlier.

The (*UTF8) and (*UCP) leading sequences can be used to set UTF and Unicode property modes. They are equivalent to setting options `unicode` and `ucp`, respectively. The (*UTF) sequence is a generic version that can be used with any of the libraries. However, the application can set option `never_utf`, 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 following pattern matches "cataract", "caterpillar", or "cat":

   ```
   cat(aract|erpillar|)
   ```

   Without the parentheses, it would match "cataract", "erpillar", or an empty string.

2.

   It sets up the subpattern as a capturing subpattern. That is, when the complete pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller through the return value of *`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 following pattern, the captured substrings are "red king", "red", and "king", and are numbered 1, 2, and 3, respectively:

```
the ((red|white) (king|queen))
```

It is not always helpful that plain parentheses fulfill two functions. Often 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 following pattern, the captured substrings are "white queen" and "queen", and are numbered 1 and 2:

```
the ((?:red|white) (king|queen))
```

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 can appear between "?" and ":". Thus, the following two patterns match the same set of strings:

```
(?i:saturday|sunday)
(?:(?i)saturday|sunday)
```

As 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 both "SUNDAY" and "Saturday".

Duplicate Subpattern Numbers

Perl 5.10 introduced a feature where 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 the following pattern:

```
(?|(Sat)ur|(Sun))day
```

As 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 a part, but not all, of one of many 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 from the Perl documentation; the numbers underneath show in which buffer the captured content is stored:

```
# before  ---------------branch-reset----------- after
/ ( a )  (?| x ( y ) z | (p (q) r) | (t) u (v) ) ( z ) /x
# 1           2         2  3        2     3     4
```

A 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/
```

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 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 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 hard to keep track of the numbers in complicated regular expressions. Also, if an expression is modified, the numbers can 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 and by number.

Names consist of up to 32 alphanumeric characters and underscores, but must start with a non-digit. Named capturing parentheses are still allocated numbers as well as names, exactly as if the names were not present. The `capture` specification to *`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 this constraint can be relaxed by setting option `dupnames` 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 that 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. The following 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.)

For capturing named subpatterns which names are not unique, the first matching occurrence (counted from left to right in the subject) is returned from *`run/3`*, if the name is specified in the `values` part of the `capture` statement. The `all_names` capturing value matches all the names in the same way.

Note:

You cannot use different names to distinguish between two subpatterns with the same number, as PCRE uses only the numbers when matching. For this reason, an error is given at compile time if different names are specified to subpatterns with the same number. However, you can specify 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 the 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 < 65536, and the first must be less than or equal to the second. For example, the following matches "zz", "zzz", or "zzzz":

```
z{2,4}
```

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, the following matches at least three successive vowels, but can match many more:

```
[aeiou]{3,}
```

The following matches exactly eight digits:

```
\d{8}
```

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 2-byte sequence in a UTF-8 string. Similarly, \X{3} matches three Unicode extended grapheme clusters, each of which can be many data units long (and they can 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 can be useful for subpatterns that are referenced as subroutines from elsewhere in the pattern (but see also section *Defining Subpatterns for Use by Reference Only*). 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:

*
: Equivalent to {0,}

+
: Equivalent to {1,}

?
: Equivalent to {0,1}

Infinite loops can be constructed 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, as there are cases where this can be useful, such patterns are now accepted. However, if any repetition of the subpattern matches 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 remaining pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between /* and */. Within the comment, individual * and / characters can appear. An attempt to match C comments by applying the pattern

```
/\*.*\*/
```

to the string

```
/* first comment */  not comment  /* second comment */
```

fails, as 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 following pattern does the right thing with the C comments:

```
/\*.*?\*/
```

The meaning of the various quantifiers is not otherwise changed, only the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. As 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 remaining pattern matches.

If option `ungreedy` 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. That is, it inverts the default behavior.

When a parenthesized subpattern is quantified with a minimum repeat count that is > 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 option `dotall` (equivalent to Perl option `/s`) is set, thus allowing the dot to match newlines, the pattern is implicitly anchored, because whatever follows is 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 if it was preceded by \A.

In cases where it is known that the subject string contains no newlines, it is worth setting `dotall` 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 can fail where a later one succeeds. Consider, for example:

```
(.*)abc\1
```

If the subject is "xyz123abc123", the match point is the fourth character. Therefore, 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 can fail where a later one succeeds. Consider the following pattern:

```
(?>.*?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

```
(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 can have been set in previous iterations. For example, after

```
/(a|(b))+/
```

matches "aba", the value of the second captured substring is "b".

Atomic Grouping and Possessive Quantifiers

With both maximizing ("greedy") and minimizing ("ungreedy" or "lazy") repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the remaining pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it to fail earlier than it otherwise might, when the author of the pattern knows that there is no point in carrying on.

Consider, for example, the pattern \d+foo when applied to the following subject line:

```
123456bar
```

After matching all six digits and then failing to match "foo", the normal action of the matcher is to try again with only five digits matching item \d+, and then with four, 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 atomic grouping is used 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 the following example:

```
(?>\d+)foo
```

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both \d+ and \d+? are prepared to adjust the number of digits they match to make the remaining pattern match, `(?>\d+)` can only match an entire sequence of digits.

Atomic groups in general can contain any 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 extra + character following a quantifier. Using this notation, the previous example can be rewritten as

```
\d++foo
```

Notice 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 option `ungreedy` is ignored. They are a convenient notation for the simpler forms of an atomic group. However, there is no difference in the meaning of a possessive quantifier and the equivalent atomic group, but there can be a performance difference; possessive quantifiers are probably 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 the Sun 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, as 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 long time. The pattern

```
(\D+|<\d+>)*[!?]
```

matches an unlimited number of substrings that either consist of non-digits, or digits enclosed in <>, followed by ! or ?. When it matches, it runs quickly. However, if it is applied to

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the internal \D+ repeat and the external * repeat in many ways, and all must be tried. (The example uses [!?] rather than a single character at the end, as 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 the following, sequences of non-digits cannot be broken, and failure happens quickly:

```
((?>\D+)|<\d+>)*[!?]
```

Back References

Outside a character class, a backslash followed by a digit > 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 < 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. That is, the parentheses that are referenced do need not be to the left of the reference for numbers < 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, as a sequence such as \50 is interpreted as a character defined in octal. For more details of the handling of digits following a backslash, see section *Non-Printing Characters* earlier. 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 to avoid 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. The following examples are 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 the following 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 fragments containing references within themselves.

A back reference matches whatever matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (section *Subpattern as Subroutines* describes a way of doing that). So, the following pattern matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility":

```
(sens|respons)e and \1ibility
```

If caseful matching is in force at the time of the back reference, the case of letters is relevant. For example, the following matches "rah rah" and "RAH RAH", but not "RAH rah", although the original capturing subpattern is matched caselessly:

```
((?i)rah)\s+\1
```

There are many 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)`. The unified back reference syntax in Perl 5.10, in which \g can be used for both numeric and named references, is also supported. The previous example can be rewritten in 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 can appear in the pattern before or after the reference.

There can be more than one back reference to the same subpattern. If a subpattern has not been used in a particular match, any back references to it always fails. For example, the following pattern always fails if it starts to match "a" rather than "bc":

```
(a|(bc))\2
```

As there can 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 option `extended` is set, this can be whitespace. Otherwise an empty comment (see section *Comments*) 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 following pattern matches any number of "a"s and also "aba", "ababbaa", and so on:

```
(a|b\1)+
```

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 consume any characters. The simple assertions coded as \b, \B, \A, \G, \Z, \z, ^, and $ are described in the previous sections.

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 done only for positive assertions. (Perl sometimes, but not always, performs capturing in negative assertions.)

Warning:

If a positive assertion containing one or more capturing subpatterns succeeds, but failure to match later in the pattern causes backtracking over this assertion, the captures within the assertion are reset only if no higher numbered captures are already set. This is, unfortunately, a fundamental limitation of the current implementation, and as PCRE1 is now in maintenance-only status, it is unlikely ever to change.

For compatibility with Perl, assertion subpatterns can be repeated. However, it makes no sense to assert the same thing many times, the side effect of capturing parentheses can occasionally be useful. In practice, there are only three cases:

- If the quantifier is {0}, the assertion is never obeyed during matching. However, it can contain internal capturing parenthesized groups that are called from elsewhere through the subroutine mechanism.
- If quantifier is {0,n}, where n > 0, it is treated as if it was {0,1}. At runtime, the remaining pattern match is tried with and without the assertion, the order depends on the greediness of the quantifier.
- If the minimum repetition is > 0, the quantifier is ignored. The assertion is obeyed only once when encountered during matching.

Lookahead Assertions

Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example, the following matches a word followed by a semicolon, but does not include the semicolon in the match:

```
\w+(?=;)
```

The following matches any occurrence of "foo" that is not followed by "bar":

```
foo(?!bar)
```

Notice 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, as 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 (?!), as an empty string always matches. So, an assertion that requires there is 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, the following finds an occurrence of "bar" that is not preceded by "foo":

```
(?<!foo)bar
```

The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are many top-level alternatives, they do not all have to have the same fixed length. Thus, the following is permitted:

```
(?<=bullock|donkey)
```

The following causes an error at compile time:

```
(?<!dogs?|cats?)
```

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 the following is not permitted, as its single top-level branch can match two different lengths:

```
(?<=ab(c|de))
```

However, it is acceptable to PCRE if rewritten to use two top-level branches:

```
(?<=abc|abde)
```

Sometimes 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 move the current position back temporarily 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, as 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 not permitted either.

"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 with lookbehind assertions to specify efficient matching of fixed-length strings at the end of subject strings. Consider the following simple pattern when applied to a long string that does not match:

```
abcd$
```

As matching proceeds from left to right, PCRE looks for each "a" in the subject and then sees if what follows matches the remaining pattern. If the pattern is specified as

```
^.*abcd$
```

the initial .* matches the entire string at first. However, when this fails (as 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

Many assertions (of any sort) can occur in succession. For example, the following matches "foo" preceded by three digits that are not "999":

```
(?<=\d{3})(?<!999)foo
```

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 does not match "123abcfoo". A pattern to do that is the following:

```
(?<=\d{3}...)(?<!999)foo
```

This time the first assertion looks at the preceding six characters, checks 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, the following matches an occurrence of "baz" that is preceded by "bar", which in turn is not preceded by "foo":

```
(?<=(?<!foo)bar)baz
```

The following pattern matches "foo" preceded by three digits and any three characters that are not "999":

```
(?<=\d{3}(?!999)...)foo
```

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 following are the two possible forms of conditional subpattern:

```
(?(condition)yes-pattern)
(?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used, otherwise the no-pattern (if present). If more than two alternatives exist in the subpattern, a compile-time error occurs. Each of the two alternatives can itself contain nested subpatterns of any form, including conditional subpatterns; the restriction to two alternatives applies only at the level of the condition. The following 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 more than one capturing subpattern with the same number exists (see section *Duplicate Subpattern Numbers* earlier), 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 option `extended`) and to divide it into three parts for ease of discussion:

```
( \( )?    [^()]+    (?(1) \) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, as no-pattern is not present, the subpattern matches nothing. That is, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If this pattern is embedded in a larger one, a relative reference can be used:

```
...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.

Rewriting the previous example to use a named subpattern gives:

```
(?<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:

```
(?(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 can 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", can be written like this (ignore whitespace and line breaks):

```
(?(DEFINE) (?<byte> 2[0-4]\d | 25[0-5] | 1\d\d | [1-9]?\d) ) \b (?&byte) (\.(?&byte)){3} \b
```

The first part of the pattern is a DEFINE group inside which is a another group named "byte" is defined. This matches an individual component of an IPv4 address (a number < 256). When matching takes place, this part of the pattern is skipped, as DEFINE acts like a false condition. The remaining 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 can be a positive or negative lookahead or lookbehind assertion. Consider the following pattern, containing non-significant whitespace, and with the two alternatives on the second line:

```
(?(?=[^a-z]*[a-z])
\d{2}-[a-z]{3}-\d{2}  |  \d{2}-\d{2}-\d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. That is, 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 to include comments in patterns that are processed by PCRE. In both cases, the start of the comment must not be in a character class, or 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 option PCRE_EXTENDED 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 section *Newline Conventions* earlier.

Notice 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 the following pattern when `extended` is set, and the default newline convention is in force:

```
abc #comment \n still comment
```

On encountering 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 a character with 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 (among other things). It does this by interpolating Perl code in the expression at runtime, 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;
```

Item (?p{...}) interpolates Perl code at runtime, 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 for individual subpattern recursion. After its introduction in PCRE and Python, this kind of recursion was later introduced into Perl at release 5.10.

A special item that consists of (? followed by a number > 0 and a closing parenthesis is a recursive subroutine call of the subpattern of the given number, if 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 that option `extended` 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. Notice the use of a possessive quantifier to avoid backtracking into sequences of non-parentheses.

If this was part of a larger pattern, you would not want to recurse the entire pattern, so instead you can use:

```
( \( ( [^()]++ | (?1) )* \) )
```

The pattern is here within parentheses so that the recursion refers 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 preceding the recursion. That is, a negative number counts capturing parentheses leftwards from the point at which it is encountered.

It is also possible to refer to later opened parentheses, by writing references such as (?+2). However, these cannot be recursive, as 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). The earlier PCRE syntax (?P>name) is also supported. We can 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 studied 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

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

it gives "no match" quickly. However, if a possessive quantifier is not used, the match runs for a long time, as there are so many different ways the + and * repeats can carve up the subject, and all must 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 item (?R) with condition (R), which tests for recursion. Consider the following pattern, which matches text in angle brackets, allowing for arbitrary nesting. Only digits are allowed in nested brackets (that is, when recursing), while any characters are permitted at the outer level.

```
< (?: (?(R) \d++  | [^<>]*+) | (?R)) * >
```

Here (?(R) is the start of a conditional subpattern, with two different alternatives for the recursive and non-recursive cases. Item (?R) 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 untried alternatives and there is a subsequent matching failure. This can be illustrated by the following pattern, which means to match a palindromic string containing 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 subpalindrome. In Perl, this pattern works; in PCRE it does not work if the pattern is longer than three characters. Consider the subject string "abcba".

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"). (Notice 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. As the recursion is treated as an atomic group, there are now no backtracking points, and so the entire match fails. (Perl can now 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:

```
^((.)(?1)\2|.)$
```

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 have another alternative to try at the higher level. That is the significant 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 only those with an odd number of characters, it is tempting to change the pattern to this:

```
^((.)(?1)\2|.?)$
```

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 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)\2|)|((.)(?3)\4|.))
```

If you want to match typical palindromic phrases, the pattern must ignore all non-word characters, which can be done as follows:

```
^\W*+(?:((.)\W*+(?1)\W*+\2|)|((.)\W*+(?3)\W*+\4|\W*+.\W*+))\W*+$
```

If run with option `caseless`, this pattern matches phrases such as "A man, a plan, a canal: Panama!" and it works well in both PCRE and Perl. Notice the use of the possessive quantifier *+ to avoid backtracking into sequences of non-word characters. Without this, PCRE takes much longer (10 times or more) to match typical phrases, and Perl takes so long that you think it has gone into a loop.

> **Note:**
>
> 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 palindrome "aba" at the start, and then fails at top level, as 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. In PCRE these values can be referenced. Consider the following pattern:

```
^(.)(\1|a(?2))
```

In PCRE, it 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 can be defined before or after the reference. A numbered reference can be absolute or relative, as in the following examples:

```
(...(absolute)...)...(?2)...
(...(relative)...)...(?-1)...
(...(?+1)...(relative)...
```

An earlier example pointed out that the following pattern matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility":

```
(sens|respons)e and \1ibility
```

If instead the following pattern is used, it matches "sense and responsibility" and the other two strings:

```
(sens|respons)e and (?1)ibility
```

Another example is provided in the discussion of DEFINE earlier.

All subroutine calls, 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, the following pattern matches "abcabc" but not "abcABC", as the change of processing option does not affect the called subpattern:

```
(abc)(?i:(?-1))
```

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 alternative syntax for referencing a subpattern as a subroutine, possibly recursively. Here follows 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 minus sign, it is taken as a relative reference, for example:

```
(abc)(?i:\g<-1>)
```

Notice 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 some "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 can take either form, possibly behaving differently depending on whether a name is present. A name is any sequence of characters that does not include a closing parenthesis. The maximum name length 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 was not there. Any number of these verbs can occur in a pattern.

The behavior of these verbs in repeated groups, assertions, and in subpatterns called as subroutines (whether or not recursively) is described 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 can 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 are not processed. processed. You can suppress the start-of-match optimizations by setting option `no_start_optimize` when calling *`compile/2`* or *`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 must 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) is 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, the following matches "AB", "AAD", or "ACD". When it matches "AB", "B" is captured by the outer parentheses.

```
A((?:A|B(*ACCEPT)|C)D)
```

The following verb causes a matching failure, forcing backtracking to occur. It is equivalent to (?!) but easier to read.

```
(*FAIL) or (*F)
```

The Perl documentation states that it is probably useful only when combined with (?{}) or (??{}). Those are Perl features that are not present in PCRE.

A match with the string "aaaa" always fails, but the callout is taken before each backtrack occurs (in this example, 10 times).

Recording Which Path Was Taken

The main purpose of this verb is to track how a match was arrived at, although it also has a secondary use in with advancing the match starting point (see (*SKIP) below).

> **Note:**
>
> In Erlang, there is no interface to retrieve a mark with *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, but the examples can help in understanding NAMES as they can be used by (*SKIP).

```
(*MARK:NAME) or (*:NAME)
```

A name is always required with this verb. There can 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 section "Extra data for `pcre_exec()`" in the `pcreapi` documentation. In the following example of `pcretest` output, 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 occur 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> /X(*MARK:A)Y|X(*MARK:B)Z/K
data> XP
No match, mark = B
```

Notice that in this unanchored example, the mark is retained from the match attempt that started at 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, 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, as 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 behavior described below is what occurs when the verb is not in a subroutine or an assertion. Subsequent sections cover these special cases.

The following verb, which must 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.

```
(*COMMIT)
```

If (*COMMIT) is the only backtracking verb that is encountered, once it has been passed, *run/2,3* is committed to find a match at the current starting point, or not at all, for example:

```
a+(*COMMIT)b
```

This matches "xxaab" but not "aacaab". It can be thought of as a kind of dynamic anchor, or "I've started, so I must finish". The name of the most recently passed (*MARK) in the path is passed back when (*COMMIT) forces a match failure.

If more than one backtracking verb exists in a pattern, a different one that follows (*COMMIT) can be triggered first, so merely passing (*COMMIT) during a match does not always guarantee that a match must be at this starting point.

Notice that (*COMMIT) at the start of a pattern is not the same as an anchor, unless the PCRE start-of-match optimizations are turned off, as shown in the following example:

```
1> re:run("xyzabc","(*COMMIT)abc",[{capture,all,list}]).
{match,["abc"]}
2> re:run("xyzabc","(*COMMIT)abc",[{capture,all,list},no_start_optimize]).
nomatch
```

For this pattern, PCRE knows that any match must start with "a", so the optimization skips along the subject to "a" before applying the pattern to the first set of data. The match attempt then succeeds. In the second call the `no_start_optimize` disables the optimization that skips along to the first character. The pattern is now applied starting at "x", and so the (*COMMIT) causes the match to fail without trying any other starting points.

The following 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:

```
(*PRUNE) or (*PRUNE:NAME)
```

If the pattern is unanchored, the normal "bumpalong" advance to the next starting character then occurs. 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 behavior of (*PRUNE:NAME) is the 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).

Note:

The fact that (*PRUNE:NAME) remembers the name is useless to the Erlang programmer, as names cannot be retrieved.

The following verb, when specified 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)
```

(*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". Notice 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".

When (*SKIP) has an associated name, its behavior is modified:

```
(*SKIP:NAME)
```

When this 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, (*SKIP) is ignored.

Notice that (*SKIP:NAME) searches only for names set by (*MARK:NAME). It ignores names that are set by (*PRUNE:NAME) or (*THEN:NAME).

The following verb causes a skip to the next innermost alternative when backtracking reaches it. That is, it cancels any further backtracking within the current alternative.

```
(*THEN) or (*THEN:NAME)
```

The verb 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 BAZ then 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 behavior 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).

Note:

The fact that (*THEN:NAME) remembers the name is useless to the Erlang programmer, as names cannot 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 the following pattern, where A, B, and so on, 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, as there are no more alternatives to try. In this case, matching does now backtrack into A.

Notice that a conditional subpattern is not considered as having two alternatives, as only one is ever used. That is, the | character in a conditional subpattern has a different meaning. Ignoring whitespace, consider:

```
^.*? (?(?=a) a | b(*THEN)c )
```

If the subject is "ba", this pattern does not match. As .*? 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 can 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 described above 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, fails the match at the current starting position, but allows an advance to the next character (for an unanchored pattern).
- (*SKIP) is similar, except that the advance can 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 the following pattern, where A, B, and so on, 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 behavior is consistent, but is not always the same as in Perl. It means that if two or more backtracking verbs appear in succession, the last of them has no effect. Consider the following example:

```
...(*COMMIT)(*PRUNE)...
```

If there is a matching failure to the right, backtracking onto (*PRUNE) causes 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:

```
/(a(*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, regardless if this is within the assertion.

Negative assertions are, however, different, 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 behavior), but if the assertion does not have such an alternative, (*THEN) behaves like (*PRUNE).

Backtracking Verbs in Subroutines

These behaviors occur regardless if the subpattern is called recursively. The treatment of subroutines in Perl 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

Erlang module

Sets are collections of elements with no duplicate elements. The representation of a set is undefined.

This module provides the same interface as the *ordsets(3)* module but with an undefined representation. One difference is that while this module considers two elements as different if they do not match (`=:=`), `ordsets` considers two elements as different if and only if they do not compare equal (`==`).

Data Types

set(Element)

As returned by *new/0*.

set() = *set*(term())

Exports

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.

filter(Pred, Set1) -> Set2

Types:

Pred = fun((Element) -> boolean())

Set1 = Set2 = *set*(Element)

Filters elements in `Set1` with boolean function `Pred`.

fold(Function, Acc0, Set) -> Acc1

Types:

Function = fun((Element, AccIn) -> AccOut)

Set = *set*(Element)

Acc0 = Acc1 = AccIn = AccOut = Acc

Folds `Function` over every element in `Set` and returns the final value of the accumulator. The evaluation order is undefined.

from_list(List) -> Set

Types:

List = [Element]
Set = *set*(Element)

Returns a set of the elements in `List`.

intersection(SetList) -> Set

Types:

SetList = [*set*(Element), ...]
Set = *set*(Element)

Returns the intersection of the non-empty list of sets.

intersection(Set1, Set2) -> Set3

Types:

Set1 = Set2 = Set3 = *set*(Element)

Returns the intersection of `Set1` and `Set2`.

is_disjoint(Set1, Set2) -> boolean()

Types:

Set1 = Set2 = *set*(Element)

Returns `true` if `Set1` and `Set2` are disjoint (have no elements in common), otherwise `false`.

is_element(Element, Set) -> boolean()

Types:

Set = *set*(Element)

Returns `true` if `Element` is an element of `Set`, otherwise `false`.

is_set(Set) -> boolean()

Types:

Set = term()

Returns `true` if `Set` is a set of elements, 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`.

new() -> *set()*

Returns a new empty set.

size(Set) -> integer() >= 0

Types:

Set = *set()*

Returns the number of elements in `Set`.

subtract(Set1, Set2) -> Set3

Types:

Set1 = Set2 = Set3 = *set*(Element)

Returns only the elements of `Set1` that are not also elements of `Set2`.

to_list(Set) -> List

Types:

Set = *set*(Element)
List = [Element]

Returns the elements of `Set` as a list. The order of the returned elements is undefined.

union(SetList) -> Set

Types:

SetList = [*set*(Element)]
Set = *set*(Element)

Returns the merged (union) set of the list of sets.

union(Set1, Set2) -> Set3

Types:

Set1 = Set2 = Set3 = *set*(Element)

Returns the merged (union) set of `Set1` and `Set2`.

See Also

gb_sets(3), *ordsets(3)*

shell

Erlang module

This module provides 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 *`history/1`* and *`results/1`*, or by setting the application configuration parameters `shell_history_length` and `shell_saved_results` for the STDLIB application.

The shell uses a helper process for evaluating commands to protect the history mechanism from exceptions. By default the evaluator process is killed when an exception occurs, but by calling *`catch_exception/1`* or by setting the application configuration parameter `shell_catch_exception` for the STDLIB application this behavior can be changed. See also the example below.

Variable bindings, and local process dictionary changes that 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 reused.

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 module `user_default`, where customized local commands can be placed. If found, the function is evaluated, otherwise an attempt is made to evaluate the function in module `shell_default`. Module `user_default` must be explicitly loaded.

The shell also permits the user to start multiple concurrent jobs. A job can be regarded as a set of processes that 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 represents a record, and the record definitions used by the compiler are unavailable at runtime. So, to read the record syntax and print tuples as records when possible, record definitions must be maintained by the shell itself.

The shell commands for reading, defining, forgetting, listing, and printing records are described below. Notice that each job has its own set of record definitions. To facilitate matters, record definitions in modules `shell_default` and `user_default` (if loaded) are read each time a new job is started. For example, adding the following line to `user_default` makes the definition of `file_info` readily available in the shell:

```
-include_lib("kernel/include/file.hrl").
```

The shell runs in two modes:

- `Normal (possibly restricted)` mode, in which commands can be edited and expressions evaluated
- Job Control Mode, `JCL`, in which jobs can be started, killed, detached, and connected

Only the currently connected job can 'talk' to the shell.

Shell Commands

`b()`

Prints the current variable bindings.

`f()`

Removes all variable bindings.

`f(X)`

Removes the binding of variable `X`.

`h()`

Prints the history list.

`history(N)`

Sets the number of previous commands to keep in the history list to `N`. The previous number is returned. Defaults to 20.

`results(N)`

Sets the number of results from previous commands to keep in the history list to `N`. The previous number is returned. Defaults to 20.

`e(N)`

Repeats command `N`, if `N` is positive. If it is negative, the `N`th previous command is repeated (that is, `e(-1)` repeats the previous command).

`v(N)`

Uses the return value of command `N` in the current command, if `N` is positive. If it is negative, the return value of the `N`th previous command is used (that is, `v(-1)` uses the value of the previous command).

`help()`

Evaluates `shell_default:help()`.

`c(Mod)`

Evaluates `shell_default:c(Mod)`. This compiles and loads the module `Mod` and purges old versions of the code, if necessary. `Mod` can be either a module name or a a source file path, with or without `.erl` extension.

`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. This means, for example, that 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/1,2,3` 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. To remove all record definitions, use `'_'`.

`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. To read all record definitions, use `'_'` as value of `RecordNames`.

Example

The following example is a long dialog with the shell. Commands starting with `>` are inputs to the shell. All other lines are output from the shell.

```
strider 1> erl
Erlang (BEAM) emulator version 5.3 [hipe] [threads:0]

Eshell V5.3  (abort with ^G)
1> Str = "abcd".
"abcd"
```

Command 1 sets variable `Str` to string `"abcd"`.

```
2> L = length(Str).
4
```

Command 2 sets `L` to the length of string `Str`.

```
3> Descriptor = {L, list_to_atom(Str)}.
{4,abcd}
```

Command 3 builds the tuple `Descriptor`, evaluating the BIF *`list_to_atom/1`* .

```
4> L.
4
```

Command 4 prints the value of variable `L`.

```
5> b().
Descriptor = {4,abcd}
L = 4
Str = "abcd"
ok
```

Command 5 evaluates the internal shell command `b()`, which is an abbreviation of "bindings". This prints the current shell variables and their bindings. `ok` at the end is the return value of function `b()`.

```
6> f(L).
ok
```

Command 6 evaluates the internal shell command `f(L)` (abbreviation of "forget"). The value of variable `L` is removed.

```
7> b().
Descriptor = {4,abcd}
Str = "abcd"
ok
```

Command 7 prints the new bindings.

```
8> f(L).
ok
```

Command 8 has no effect, as `L` has no value.

```
9> {L, _} = Descriptor.
{4,abcd}
```

Command 9 performs a pattern matching operation on `Descriptor`, binding a new value to `L`.

```
10> L.
4
```

Command 10 prints the current value of `L`.

```
11> {P, Q, R} = Descriptor.
** exception error: no match of right hand side value {4,abcd}
```

Command 11 tries to match `{P, Q, R}` against `Descriptor`, which is `{4, abc}`. The match fails and none of the new variables become bound. The printout starting with "`** exception error:`" is not the value of the expression (the expression had no value because its evaluation failed), but a warning printed by the system to inform the user that an error has occurred. The values of the other variables (`L`, `Str`, and so on) are unchanged.

```
12> P.
* 1: variable 'P' is unbound
13> Descriptor.
{4,abcd}
```

Commands 12 and 13 show that `P` is unbound because the previous command failed, and that `Descriptor` has not changed.

```
14>{P, Q} = Descriptor.
{4,abcd}
15> P.
4
```

Commands 14 and 15 show a correct match where `P` and `Q` are bound.

```
16> f().
ok
```

Command 16 clears all bindings.

The next few commands assume that `test1:demo(X)` is defined as follows:

```
demo(X) ->
  put(aa, worked),
  X = 1,
  X + 10.
```

```
17> put(aa, hello).
undefined
18> get(aa).
hello
```

Commands 17 and 18 set and inspect the value of item `aa` in the process dictionary.

```
19> Y = test1:demo(1).
11
```

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 dictionary item `aa` can be seen in command 20.

```
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
```

Commands 21 and 22 change the value of 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.

```
23> Z.
* 1: variable 'Z' is unbound
24> get(aa).
hello
```

Commands 23 and 24 show that `Z` was not bound and that dictionary item `aa` has retained its original value.

```
25> erase(), put(aa, hello).
undefined
26> spawn(test1, demo, [1]).
<0.57.0>
27> get(aa).
hello
```

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.

```
28> io:format("hello hello\n").
hello hello
ok
29> e(28).
hello hello
ok
30> v(28).
ok
```

Commands 28, 29 and 30 use the history facilities of the shell. Command 29 re-evaluates command 28. Command 30 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 as follows:

```
-record(rec, {a, b = val()}).

val() ->
  3.
```

```
31> c(ex).
{ok,ex}
32> rr(ex).
[rec]
```

Commands 31 and 32 compile file `ex.erl` and read 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.

```
33> rl(rec).
-record(rec,{a,b = val()}).
ok
```

Command 33 prints the definition of the record named `rec`.

```
34> #rec{}.
** exception error: undefined shell command val/0
```

Command 34 tries to create a `rec` record, but fails as function `val/0` is undefined.

```
35> #rec{b = 3}.
#rec{a = undefined,b = 3}
```

Command 35 shows the workaround: explicitly assign values to record fields that cannot otherwise be initialized.

```
36> rp(v(-1)).
#rec{a = undefined,b = 3}
ok
```

Command 36 prints the newly created record using record definitions maintained by the shell.

```
37> rd(rec, {f = orddict:new()}).
rec
```

Command 37 defines a record directly in the shell. The definition replaces the one read from file `ex.beam`.

```
38> #rec{}.
#rec{f = []}
ok
```

Command 38 creates a record using the new definition, and prints the result.

```
39> rd(rec, {c}), A.
* 1: variable 'A' is unbound
40> #rec{}.
#rec{c = undefined}
ok
```

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 as follows:

```
loop(N) ->
  io:format("Hello Number: ~w~n", [N]),
  loop(N+1).
```

```
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
```

Command 41 evaluates `test1:loop(0)`, which puts the system into an infinite loop. At this point the user types `^G` (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, command `i` ("interrupt") terminates the looping program, and command `c` connects to the shell again. As the process was running in the background before we killed it, more printouts occur before message "`** exception exit: killed`" is shown.

```
42> E = ets:new(t, []).
#Ref<0.1662103692.2407923716.214192>
```

Command 42 creates an ETS table.

```
43> ets:insert({d,1,2}).
** exception error: undefined function ets:insert/1
```

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.

```
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})
```

Command 44 corrects the mistake, but the ETS table has been destroyed as it was owned by the killed evaluator process.

```
45> f(E).
ok
46> catch_exception(true).
false
```

Command 46 sets the exception handling of the evaluator process to `true`. The exception handling can also be set when starting Erlang by `erl -stdlib shell_catch_exception true`.

```
47> E = ets:new(t, []).
#Ref<0.1662103692.2407923716.214197>
48> ets:insert({d,1,2}).
* exception error: undefined function ets:insert/1
```

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.

```
49> ets:insert(E, {d,1,2}).
true
```

Command 49 successfully inserts the tuple into the ETS table.

```
50> ets:insert(#Ref<0.1662103692.2407923716.214197>, {e,3,4}).
true
```

Command 50 inserts another tuple into the ETS table. This time the first argument is the table identifier itself. The shell can parse commands with pids (`<0.60.0>`), ports (`#Port<0.536>`), references (`#Ref<0.1662103692.2407792644.214210>`), and external functions (`#Fun<a.b.1>`), but the command fails unless the corresponding pid, port, reference, or function can be created in the running system.

```
51> halt().
strider 2>
```

Command 51 exits the Erlang runtime system.

JCL Mode

When the shell starts, it starts a single evaluator process. This process, together with any local processes that it spawns, is referred to as a `job`. Only the current job, which is said to be `connected`, can perform operations with standard I/O. All other jobs, which are said to be `detached`, are `blocked` if they attempt to use standard I/O.

All jobs that do not use standard I/O 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 that use standard I/O by the current job are 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. So, any variable bindings and the process dictionary are 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 are not 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 is assigned the new index `[nn]`, which can be used in references.

`s [shell]`

Starts a new job. This is assigned the new index `[nn]`, which can be used in references. If optional argument `shell` is specified, 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 optional argument `shell` is specified, it is assumed to be a module that implements an alternative shell.

`q`

Quits Erlang. Notice that this option is disabled if Erlang is started with the ignore break, `+Bi`, system flag (which can be useful, for example when running a restricted shell, see the next section).

`?`

Displays the help message above.

The behavior of shell escape can be changed by 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), start Erlang with flag `-remsh`, for example, `erl -sname this_node -remsh other_node@other_host`

Restricted Shell

The shell can 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 system flag `+Bi`).

When the restricted shell evaluates an expression and encounters a function call or an operator application, it calls a callback function (with information about the function call in question). This callback function returns `true` to let the shell go ahead with the evaluation, or `false` to abort it. There are two possible callback functions for the user to implement:

- `local_allowed(Func, ArgList, State) -> {boolean(),NewState}`

  This is used to determine if the call to the local function `Func` with arguments `ArgList` is to be allowed.
- `non_local_allowed(FuncSpec, ArgList, State) -> {boolean(),NewState} | {{redirect,NewFuncSpec,NewArgList},NewState}`

  This is used to determine if the call to non-local function `FuncSpec` (`{Module,Func}` or a fun) with arguments `ArgList` is to 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 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.)

Argument `State` is a tuple `{ShellState,ExprState}`. The return value `NewState` has the same form. This can 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 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 *`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 run in restricted or normal mode, respectively.
- If restricted mode has been enabled on a particular node, remote shells connecting to this node 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 STDLIB 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 *`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 *`prompt_func/1`* or by setting application configuration parameter `shell_prompt_func` for the STDLIB application.

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 is to return a list of characters or an atom. This constraint is because of the Erlang I/O protocol. Unicode characters beyond code point 255 are allowed in the list and the atom. Notice that in restricted mode the call `Mod:Func(L)` must be allowed or the default shell prompt function is called.

Exports

`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 example, ports and ETS tables as well as processes linked to the evaluator process survive the exception.

`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. Defaults to 20.

`prompt_func(PromptFunc) -> PromptFunc2`

Types:

`PromptFunc = PromptFunc2 = default | {module(), atom()}`

Sets the shell prompt function to `PromptFunc`. The previous prompt function is returned.

`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. Defaults to 20.

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`. Defaults to `true`, which means that lists of integers are printed using the string syntax, when possible. Value `false` means that no lists are printed using the string syntax.

shell_default

Erlang module

The functions in this module are called when no module name is specified in a shell command.

Consider the following shell dialog:

```
1> lists:reverse("abc").
"cba"
2> c(foo).
{ok, foo}
```

In command one, module *`lists`* is called. In command two, no module name is specified. The shell searches module `user_default` followed by module `shell_default` for function `c/1`.

`shell_default` is intended for "system wide" customizations to the shell. `user_default` is intended for "local" or individual user customizations.

Hint

To add your own commands to the shell, create a module called `user_default` and add the commands you want. Then add the following line as the **first** line in your `.erlang` file in your home directory.

```
code:load_abs("$PATH/user_default").
```

`$PATH` is the directory where your `user_default` module can be found.

slave

Erlang module

This module provides functions for starting Erlang slave nodes. All slave nodes that are started by a master terminate automatically when the master terminates. All terminal output produced at the slave is sent back to the master node. File I/O is done through the master.

Slave nodes on other hosts than the current one are started with the `rsh` program. 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 (for details, see the `rsh` documentation). 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, see the documentation for the `rsh` program.

An alternative to the `rsh` program can be specified on the command line to *`erl(1)`* as follows:

```
-rsh Program
```

The slave node is to use the same file system at the master. At least, Erlang/OTP is to be installed in the same place on both computers and the same version of Erlang is to be used.

A node running on Windows can only start slave nodes on the host on which it is running.

The master node must be alive.

Exports

`pseudo([Master | ServerList]) -> ok`

Types:

`Master = node()`
`ServerList = [atom()]`

Calls `pseudo(Master, ServerList)`. If you want to start a node from the command line and set up a number of pseudo servers, an Erlang runtime system can be started as follows:

```
% erl -name abc -s slave pseudo klacke@super x --
```

`pseudo(Master, ServerList) -> ok`

Types:

`Master = node()`
`ServerList = [atom()]`

Starts a number of pseudo servers. A pseudo server is a server with a registered name that does nothing but pass on all message to the real server that executes at a master node. A pseudo server is an intermediary that only has the same registered name as the real server.

For example, if you have started a slave node `N` and want to execute `pxw` graphics code on this node, you can start server `pxw_server` as a pseudo server at the slave node. This is illustrated as follows:

```
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 that executes the function receives messages. All messages received are simply passed on to `Pid`.

start(Host) -> {ok, Node} | {error, Reason}
start(Host, Name) -> {ok, Node} | {error, Reason}
start(Host, Name, Args) -> {ok, Node} | {error, Reason}

Types:

Host = *inet:hostname()*
Name = atom() | string()
Args = string()
Node = node()
Reason = timeout | no_rsh | {already_running, Node}

Starts a slave node on 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 becomes `Name@Host`. If no name is provided, the name becomes the same as the node that executes the call (except the host name part of the node name).

The slave node resets its `user` process so that all terminal I/O that is produced at the slave is automatically relayed to the master. Also, the file process is relayed to the master.

Argument `Args` 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 you want to start a slave node at host `H` with node name `Name@H` and want the slave node to have the following properties:

- Directory `Dir` is to be added to the code path.
- The Mnesia directory is to be set to `M`.
- The Unix `DISPLAY` environment variable is to 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).
```

The function returns `{ok, Node}`, where `Node` is the name of the new node, otherwise `{error, Reason}`, where `Reason` can be one of:

`timeout`

The master node failed to get in contact with the slave node. This can occur 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 name `Name@Host` already exists.

```
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 = inet:hostname()
Name = atom() | string()
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.

For a description of arguments and return values, see *`start/1,2,3`*.

```
stop(Node) -> ok
```

Types:

```
Node = node()
```

Stops (kills) a node.

sofs

Erlang module

This module provides 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 is always the case in this module), this is denoted 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 **empty set** contains no elements.

  Set A is **equal** to set B if they contain the same elements, which is denoted A = B. Two ordered sets are equal if they contain the same number of elements and have equal elements at each coordinate.

  Set B is a **subset** of set A if A contains all elements that B contains.

  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 × Y, is the set {a : a = (x, y) for some x in X and for some y in Y}.

  A **relation** is a subset of X × Y. Let R be a relation. The fact that (x, y) belongs to R is written as x R y. As relations are sets, the definitions of the last item (subset, union, and so on) apply to relations as well.

  The **domain** of R is the set {x : x R y for some y in Y}.

  The **range** of R is the set {y : x R y for some x in X}.

  The **converse** of R is the set {a : a = (y, x) for some (x, y) in R}.

  If A is a subset of X, the **image** of A under R is the set {y : x R y for some x in A}. If B is a subset of Y, the **inverse image** of B is the set {x : x R y for some y in B}.

  If R is a relation from X to Y, and S is a relation from Y to Z, 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.

If X = Y, then R is called a relation **in** X.

The **field** of a relation R in X is the union of the domain of R and the range of R.

If R is a relation in X, and if S is defined so that x S y if x R y and not x = y, then S is the **strict** relation corresponding to R. Conversely, 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 × 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 is not required that the domain of F is 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.

  As functions are relations, the definitions of the last item (domain, range, and so on) apply to functions as well.

  If the converse of a function F is a function F', then F' is called the **inverse** of F.

  The relative product of two functions F1 and F2 is called the **composite** of F1 and F2 if the range of F1 is a subset of the domain of F2.

- Sometimes, when the range of a function is more important than the function itself, the function is called a **family**.

  The domain of a family is called the **index set**, and the range is called the **indexed set**.

  If x is a family from I to X, then x[i] denotes the value of the function at index i. The notation "a family in X" is used for such a family.

  When the indexed set is a set of subsets of a set X, we call x a **family of subsets** of X.

  If x is a family of subsets of X, the union of the range of x is called the **union of the family** x.

  If x is non-empty (the index set is non-empty), the **intersection of the family** x is the intersection of the range of x.

  In this module, the only families that are considered are families of subsets of some set X; in the following, the word "family" is used for such families of subsets.

- A **partition** of a set X is a collection S of non-empty subsets of X whose union is X and whose elements are pairwise disjoint.

  A relation in a set is an **equivalence relation** if it is reflexive, symmetric, and transitive.

  If R is an equivalence relation in X, and x is an element of X, the **equivalence class** of x with respect to R is the set of all those elements y of X for which x R y holds. The equivalence classes constitute a partitioning of X. Conversely, if C is a partition of X, 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, the **canonical map** is the function that maps every element of X onto its equivalence class.

- Relations as defined above (as sets of ordered pairs) are from now on referred to as **binary relations**.

  We call a set of ordered sets (x[1], ..., x[n]) an **(n-ary) relation**, and say that the relation is a subset of the Cartesian product X[1] × ... × X[n], where x[i] is an element of X[i], 1 <= i <= n.

The **projection** of an n-ary relation R onto coordinate i is the set {x[i] : (x[1], ..., x[i], ..., x[n]) in R for some x[j] in X[j], 1 <= j <= n and not i = j}. The projections of a binary relation R onto the first and second coordinates are the domain and the range of R, respectively.

The relative product of binary relations can be generalized to n-ary relations as follows. Let TR be an ordered set (R[1], ..., R[n]) of binary relations from X to Y[i] and S a binary relation from (Y[1] × ... × Y[n]) to Z. The **relative product** of TR and S is the binary relation T from X to Z defined so that x T z if and only if there exists an element y[i] in Y[i] for each 1 <= i <= n such that x R[i] y[i] and (y[1], ..., y[n]) S z. Now let TR be a an ordered set (R[1], ..., R[n]) of binary relations from X[i] to Y[i] and S a subset of X[1] × ... × X[n]. The **multiple relative product** of TR and S is defined to be the set {z : z = ((x[1], ..., x[n]), (y[1],...,y[n])) for some (x[1], ..., x[n]) in S and for some (x[i], y[i]) in R[i], 1 <= i <= n}.

The **natural join** of an n-ary relation R and an m-ary relation S on coordinate i and j is defined to be the set {z : z = (x[1], ..., x[n], y[1], ..., y[j-1], y[j+1], ..., y[m]) for some (x[1], ..., x[n]) in R and for some (y[1], ..., y[m]) in S such that x[i] = y[j]}.

- The sets recognized by this module are represented by elements of the relation Sets, which is defined as the smallest set such that:
  - For every atom T, except '_', and for every term X, (T, X) belongs to Sets (**atomic sets**).
  - (['_'], []) belongs to Sets (the **untyped empty set**).
  - For every tuple T = {T[1], ..., T[n]} and for every tuple X = {X[1], ..., X[n]}, if (T[i], X[i]) belongs to Sets for every 1 <= i <= n, then (T, X) belongs to Sets (**ordered sets**).
  - For every term T, if X is the empty list or a non-empty sorted list [X[1], ..., X[n]] without duplicates such that (T, X[i]) belongs to Sets for every 1 <= i <= n, then ([T], X) belongs to Sets (**typed unordered sets**).

  An **external set** is an element of the range of Sets.

  A **type** is an element of the domain of Sets.

  If S is an element (T, X) of Sets, then T is a **valid type** of X, T is the type of S, and X is the external set of S. *`from_term/2`* creates a set from a type and an Erlang term turned into an external set.

  The sets represented by Sets are the elements of the range of function Set from Sets to Erlang terms and sets of Erlang terms:

  - Set(T,Term) = Term, where T is an atom
  - Set({T[1], ..., T[n]}, {X[1], ..., X[n]}) = (Set(T[1], X[1]), ..., Set(T[n], X[n]))
  - Set([T], [X[1], ..., X[n]]) = {Set(T, X[1]), ..., Set(T, X[n])}
  - Set([T], []) = {}

  When there is no risk of confusion, elements of Sets are identified with the sets they represent. For example, if U is the result of calling *`union/2`* with S1 and S2 as arguments, then U is said to be the union of S1 and S2. A more precise formulation is that Set(U) is the union of Set(S1) and Set(S2).

The types are used to implement the various conditions that sets must 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`*) 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 used when the elements of the unordered set are atomic or ordered sets. It must also be the case that the type of the elements matches some clause of Fun (the type of the created set is the result of applying Fun to the type of the given set), and that Fun does nothing but selecting, duplicating, or rearranging parts of the elements.
- Specifying a SetFun as an integer I is equivalent to specifying `{external, fun(X) -> element(I, X) end}`, but is to be preferred, as it makes it possible to handle this case even more efficiently.

Examples of SetFuns:

```
fun sofs:union/1
fun(S) -> sofs:partition(1, S) end
{external, fun(A) -> A end}
{external, fun({A,_,C}) -> {C,A} end}
{external, fun({_,{_,C}}) -> C end}
{external, fun({_,{_,{_,E}=C}}) -> {E,{E,C}} end}
2
```

The order in which a SetFun is applied to the elements of an unordered set is not specified, and can change in future versions of this module.

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/2`*, *`is_empty_set/1`*, *`is_set/1`*, *`is_sofs_set/1`*, *`to_external/1`* *`type/1`*.

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, 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

a_function(Tuples) -> Function

a_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 specified, `[{atom, atom}]` is used as the function type.

canonical_relation(SetOfSets) -> BinRel

Types:

BinRel = *binary_relation()*

SetOfSets = *set_of_sets()*

Returns the binary relation containing the elements (E, Set) such that Set belongs to `SetOfSets` and E belongs to Set. If `SetOfSets` is a *partition* of a set X and R is the equivalence relation in X induced by `SetOfSets`, then the returned relation is the *canonical map* from X onto the equivalence classes with respect to R.

```
1> Ss = sofs:from_term([[a,b],[b,c]]),
CR = sofs:canonical_relation(Ss),
sofs:to_external(CR).
[{a,[a,b]},{b,[a,b]},{b,[b,c]},{c,[b,c]}]
```

composite(Function1, Function2) -> Function3

Types:

Function1 = Function2 = Function3 = *a_function()*

Returns the *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}]
```

constant_function(Set, AnySet) -> Function

Types:

AnySet = *anyset()*

Function = *a_function()*

Set = *a_set()*

Creates the *function* that maps each element of set `Set` onto `AnySet`.

```
1> S = sofs:set([a,b]),
E = sofs:from_term(1),
R = sofs:constant_function(S, E),
sofs:to_external(R).
[{a,1},{b,1}]
```

converse(BinRel1) -> BinRel2

Types:

BinRel1 = BinRel2 = *binary_relation()*

Returns the *converse* of the binary relation `BinRel1`.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,a}]),
R2 = sofs:converse(R1),
sofs:to_external(R2).
[{a,1},{a,3},{b,2}]
```

difference(Set1, Set2) -> Set3

Types:

Set1 = Set2 = Set3 = *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-neighbors of a. If no type is explicitly specified, [{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`.

```
1> R = sofs:relation([{1,a},{1,b},{2,b},{2,c}]),
S = sofs:domain(R),
sofs:to_external(S).
[1,2]
```

drestriction(BinRel1, Set) -> BinRel2

Types:

BinRel1 = BinRel2 = *binary_relation()*
Set = *a_set()*

Returns the difference between the binary relation `BinRel1` and the *restriction* of `BinRel1` to `Set`.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,c}]),
S = sofs:set([2,4,6]),
R2 = sofs:drestriction(R1, S),
sofs:to_external(R2).
[{1,a},{3,c}]
```

`drestriction(R, S)` is equivalent to `difference(R, restriction(R, S))`.

drestriction(SetFun, Set1, Set2) -> Set3

Types:

SetFun = *set_fun()*
Set1 = Set2 = Set3 = *a_set()*

Returns a subset of `Set1` containing those elements that do not give an element in `Set2` as the result of applying `SetFun`.

```
1> SetFun = {external, fun({_A,B,C}) -> {B,C} end},
R1 = sofs:relation([{a,aa,1},{b,bb,2},{c,cc,3}]),
R2 = sofs:relation([{bb,2},{cc,3},{dd,4}]),
R3 = sofs:drestriction(SetFun, R1, R2),
sofs:to_external(R3).
[{a,aa,1}]
```

`drestriction(F, S1, S2)` is equivalent to `difference(S1, restriction(F, S1, S2))`.

`empty_set() -> Set`

Types:

`Set = 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`).

```
1> S = sofs:set([b,c]),
A = sofs:empty_set(),
R = sofs:family([{a,[1,2]},{b,[3]}]),
X = sofs:extension(R, S, A),
sofs:to_external(X).
[{a,[1,2]},{b,[3]},{c,[]}]
```

`family(Tuples) -> Family`

`family(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 specified, `[{atom, [atom]}]` is used as the family type.

`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, otherwise `Family1[i]`.

```
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]`.

```
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].

```
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, the process exits with a `badarg` message.

```
1> F1 = sofs:from_term([{a,[[1,2,3],[2,3,4]]},{b,[[x,y,z],[x,y]]}]),
F2 = sofs:family_intersection(F1),
sofs:to_external(F2).
[{a,[2,3]},{b,[x,y]}]
```

family_intersection(Family1, Family2) -> Family3

Types:

Family1 = Family2 = Family3 = *family()*

If `Family1` and `Family2` are *families*, 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:

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}`, then `Fun2` is applied to the *external set* of `Family1`[i], otherwise `Fun` is applied to `Family1`[i].

```
1> F1 = sofs:family([{a,[1,2,3]},{b,[1,2]},{c,[1]}]),
SpecFun = fun(S) -> sofs:no_elements(S) =:= 2 end,
F2 = sofs:family_specification(SpecFun, F1),
sofs:to_external(F2).
[{b,[1,2]}]
```

family_to_digraph(Family) -> Graph
family_to_digraph(Family, GraphType) -> Graph

Types:

Graph = *digraph:graph()*
Family = *family()*
GraphType = [*digraph:d_type()*]

Creates a directed graph from *family* `Family`. For each pair (a, {b[1], ..., b[n]}) of `Family`, vertex a and the edges (a, b[i]) for 1 <= i <= n are added to a newly created directed graph.

If no graph type is specified, *`digraph:new/0`* is used for creating the directed graph, otherwise argument `GraphType` 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(family_to_digraph(F), type(F))`. Equality holds if `union_of_family(F)` is a subset of `domain(F)`.

Creating a cycle in an acyclic graph exits the process with a `cyclic` message.

family_to_relation(Family) -> BinRel

Types:

Family = *family()*
BinRel = *binary_relation()*

If `Family` is a *family*, then `BinRel` is the binary relation containing all pairs (i, x) such that i belongs to the index set of `Family` and x belongs to `Family`[i].

```
1> F = sofs:family([{a,[]}, {b,[1]}, {c,[2,3]}]),
R = sofs:family_to_relation(F),
sofs:to_external(R).
[{b,1},{c,2},{c,3}]
```

family_union(Family1) -> Family2

Types:

Family1 = Family2 = *family()*

If `Family1` is a *family* 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].

```
1> F1 = sofs:from_term([{a,[[1,2],[2,3]]},{b,[[]]}]),
F2 = sofs:family_union(F1),
sofs:to_external(F2).
[{a,[1,2,3]},{b,[]}]
```

`family_union(F)` is equivalent to `family_projection(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 map i, otherwise `Family1`[i] or `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_union(F1, F2),
sofs:to_external(F3).
[{a,[1,2]},{b,[3,4,5]},{c,[5,6,7,8]},{d,[9,10]}]
```

field(BinRel) -> Set

Types:

BinRel = *binary_relation()*

Set = *a_set()*

Returns the *field* of the binary relation `BinRel`.

```
1> R = sofs:relation([{1,a},{1,b},{2,b},{2,c}]),
S = sofs:field(R),
sofs:to_external(S).
[1,2,a,b,c]
```

`field(R)` is equivalent to `union(domain(R), range(R))`.

from_external(ExternalSet, Type) -> AnySet

Types:

ExternalSet = *external_set()*

AnySet = *anyset()*

Type = *type()*

Creates a set from the *external set* `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 list `ListOfSets`.

```
1> S1 = sofs:relation([{a,1},{b,2}]),
S2 = sofs:relation([{x,3},{y,4}]),
S = sofs:from_sets([S1,S2]),
sofs:to_external(S).
[[{a,1},{b,2}],[{x,3},{y,4}]]
```

from_sets(TupleOfSets) -> Ordset

Types:

Ordset = *ordset()*

TupleOfSets = *tuple_of*(*anyset()*)

Returns the *ordered set* containing the sets of the non-empty tuple `TupleOfSets`.

from_term(Term) -> AnySet

from_term(Term, Type) -> AnySet

Types:

AnySet = *anyset()*

Term = term()

Type = *type()*

Creates an element of *Sets* by traversing term `Term`, sorting lists, removing duplicates, and deriving or verifying a *valid type* for the so obtained external set. An explicitly specified *type* `Type` can be used to limit the depth of the traversal; an atomic type stops the traversal, as shown by the following example where `"foo"` and `{"foo"}` are left unmodified:

```
1> S = sofs:from_term([{{"foo"},[1,1]},{"foo",[2,2]}],
[{atom,[atom]}]),
sofs:to_external(S).
[{{"foo"},[1]},{"foo",[2]}]
```

`from_term` can be used for creating atomic or ordered sets. The only purpose of such a set is that of later building unordered sets, as all functions in this module that **do** anything operate on unordered sets. Creating unordered sets from a collection of ordered sets can 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. The following example shows that a set can be built "layer by layer":

```
1> A = sofs:from_term(a),
S = sofs:set([1,2,3]),
P1 = sofs:from_sets({A,S}),
P2 = sofs:from_term({b,[6,5,4]}),
Ss = sofs:from_sets([P1,P2]),
sofs:to_external(Ss).
[{a,[1,2,3]},{b,[4,5,6]}]
```

Other functions that create sets are *`from_external/2`* and *`from_sets/1`*. Special cases of `from_term/2` are *`a_function/1,2`*, *`empty_set/0`*, *`family/1,2`*, *`relation/1,2`*, and *`set/1,2`*.

image(BinRel, Set1) -> Set2

Types:

BinRel = *binary_relation()*

Set1 = Set2 = *a_set()*

Returns the *image* of set `Set1` under the binary relation `BinRel`.

```
1> R = sofs:relation([{1,a},{2,b},{2,c},{3,d}]),
S1 = sofs:set([1,2]),
S2 = sofs:image(R, S1),
sofs:to_external(S2).
[a,b,c]
```

intersection(SetOfSets) -> Set

Types:

Set = *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.

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 *family* `Family`.

Intersecting an empty family exits the process with a `badarg` message.

```
1> F = sofs:family([{a,[0,2,4]},{b,[0,1,2]},{c,[2,3]}]),
S = sofs:intersection_of_family(F),
sofs:to_external(S).
[2]
```

inverse(Function1) -> Function2

Types:

Function1 = Function2 = *a_function()*

Returns the *inverse* of function `Function1`.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,c}]),
R2 = sofs:inverse(R1),
sofs:to_external(R2).
[{a,1},{b,2},{c,3}]
```

inverse_image(BinRel, Set1) -> Set2

Types:

BinRel = *binary_relation()*

Set1 = Set2 = *a_set()*

Returns the *inverse image* of `Set1` under the binary relation `BinRel`.

```
1> R = sofs:relation([{1,a},{2,b},{2,c},{3,d}]),
S1 = sofs:set([c,d,e]),
S2 = sofs:inverse_image(R, S1),
sofs:to_external(S2).
[2,3]
```

is_a_function(BinRel) -> Bool

Types:

Bool = boolean()
BinRel = *binary_relation()*

Returns `true` if the binary relation `BinRel` is a *function* or the untyped empty set, otherwise `false`.

is_disjoint(Set1, Set2) -> Bool

Types:

Bool = boolean()
Set1 = Set2 = *a_set()*

Returns `true` if `Set1` and `Set2` are *disjoint*, otherwise `false`.

is_empty_set(AnySet) -> Bool

Types:

AnySet = *anyset()*
Bool = boolean()

Returns `true` if `AnySet` is an empty unordered set, otherwise `false`.

is_equal(AnySet1, AnySet2) -> Bool

Types:

AnySet1 = AnySet2 = *anyset()*
Bool = boolean()

Returns `true` if `AnySet1` and `AnySet2` are *equal*, otherwise `false`. The following example shows that `==/2` is used when comparing sets for equality:

```
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, otherwise `false`.

is_subset(Set1, Set2) -> Bool

Types:

Bool = boolean()
Set1 = Set2 = *a_set()*

Returns `true` if `Set1` is a *subset* of `Set2`, otherwise `false`.

is_type(Term) -> Bool

Types:

Bool = boolean()
Term = term()

Returns `true` if 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`.

```
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` gives an element in `Set2`, the element belongs to `Set3`, otherwise the element belongs to `Set4`.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,c}]),
S = sofs:set([2,4,6]),
{R2,R3} = sofs:partition(1, R1, S),
{sofs:to_external(R2),sofs:to_external(R3)}.
{[{2,b}],[{1,a},{3,c}]}
```

`partition(F, S1, S2)` is equivalent to `{restriction(F, S1, S2), drestriction(F, S1, S2)}`.

partition_family(SetFun, Set) -> Family

Types:

Family = *family()*
SetFun = *set_fun()*
Set = *a_set()*

Returns *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).
[{{a,a},[{a,a,a,a}]},{{a,b},[{a,a,b,b},{a,b,b,b}]}]
```

product(TupleOfSets) -> Relation

Types:

Relation = *relation()*

TupleOfSets = *tuple_of(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).
[{a,1,x},{a,1,y},{a,2,x},{a,2,y},{b,1,x},{b,1,y},{b,2,x},{b,2,y}]
```

product(Set1, Set2) -> BinRel

Types:

BinRel = *binary_relation()*

Set1 = Set2 = *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).
[{1,a},{1,b},{2,a},{2,b}]
```

`product(S1, S2)` is equivalent to `product({S1, S2})`.

projection(SetFun, Set1) -> Set2

Types:

SetFun = *set_fun()*

Set1 = Set2 = *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 tuple size is N, is used as type of the relation. If no type is explicitly specified, the size of the first tuple of `Tuples` is used if there is such a tuple. `relation([])` is equivalent to `relation([], 2)`.

relation_to_family(BinRel) -> Family

Types:

Family = *family()*

BinRel = *binary_relation()*

Returns *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`.

```
1> R = sofs:relation([{b,1},{c,2},{c,3}]),
F = sofs:relation_to_family(R),
sofs:to_external(F).
[{b,[1]},{c,[2,3]}]
```

relative_product(ListOfBinRels) -> BinRel2

relative_product(ListOfBinRels, BinRel1) -> BinRel2

Types:

ListOfBinRels = [BinRel, ...]

BinRel = BinRel1 = BinRel2 = *binary_relation()*

If `ListOfBinRels` is a non-empty list [R[1], ..., R[n]] of binary relations and `BinRel1` is a binary relation, then `BinRel2` is the *relative product* of the ordered set (R[i], ..., R[n]) and `BinRel1`.

If `BinRel1` is omitted, the relation of equality between the elements of the *Cartesian product* of the ranges of R[i], range R[1] × ... × range R[n], is used instead (intuitively, nothing is "lost").

```
1> TR = sofs:relation([{1,a},{1,aa},{2,b}]),
R1 = sofs:relation([{1,u},{2,v},{3,c}]),
R2 = sofs:relative_product([TR, R1]),
sofs:to_external(R2).
[{1,{a,u}},{1,{aa,u}},{2,{b,v}}]
```

Notice that `relative_product([R1], R2)` is different from `relative_product(R1, R2)`; the list of one element is not identified with the element itself.

relative_product(BinRel1, BinRel2) -> BinRel3

Types:

BinRel1 = BinRel2 = BinRel3 = *binary_relation()*

Returns the *relative product* of the binary relations `BinRel1` and `BinRel2`.

relative_product1(BinRel1, BinRel2) -> BinRel3

Types:

BinRel1 = BinRel2 = BinRel3 = *binary_relation()*

Returns the *relative product* of the *converse* of the binary relation `BinRel1` and the binary relation `BinRel2`.

```
1> R1 = sofs:relation([{1,a},{1,aa},{2,b}]),
R2 = sofs:relation([{1,u},{2,v},{3,c}]),
R3 = sofs:relative_product1(R1, R2),
sofs:to_external(R3).
[{a,u},{aa,u},{b,v}]
```

`relative_product1(R1, R2)` is equivalent to `relative_product(converse(R1), R2)`.

restriction(BinRel1, Set) -> BinRel2

Types:

BinRel1 = BinRel2 = *binary_relation()*
Set = *a_set()*

Returns the *restriction* of the binary relation `BinRel1` to `Set`.

```
1> R1 = sofs:relation([{1,a},{2,b},{3,c}]),
S = sofs:set([1,2,4]),
R2 = sofs:restriction(R1, S),
sofs:to_external(R2).
[{1,a},{2,b}]
```

restriction(SetFun, Set1, Set2) -> Set3

Types:

SetFun = *set_fun()*
Set1 = Set2 = Set3 = *a_set()*

Returns a subset of `Set1` containing those elements that gives an element in `Set2` as the result of applying `SetFun`.

```
1> S1 = sofs:relation([{1,a},{2,b},{3,c}]),
S2 = sofs:set([b,c,d]),
S3 = sofs:restriction(2, S1, S2),
sofs:to_external(S3).
[{2,b},{3,c}]
```

`set(Terms) -> 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 specified, `[atom]` is used as the set type.

`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.

```
1> R1 = sofs:relation([{a,1},{b,2}]),
R2 = sofs:relation([{x,1},{x,2},{y,3}]),
S1 = sofs:from_sets([R1,R2]),
S2 = sofs:specification(fun sofs:is_a_function/1, S1),
sofs:to_external(S2).
[[{a,1},{b,2}]]
```

`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,b]
3> sofs:to_external(sofs:substitution(1,sofs:relation(L))).
[{{a,1},a},{{b,2},b}]
4> SetFun = {external, fun({A,_}=E) -> {E,A} end},
sofs:to_external(sofs:projection(SetFun,sofs:relation(L))).
[{{a,1},a},{{b,2},b}]
```

The relation of equality between the elements of {a,b,c}:

```
1> I = sofs:substitution(fun(A) -> A end, sofs:set([a,b,c])),
sofs:to_external(I).
[{a,a},{b,b},{c,c}]
```

Let `SetOfSets` be a set of sets and `BinRel` a binary relation. The function that maps each element `Set` of `SetOfSets` onto the *image* of `Set` under `BinRel` is returned by the following function:

```
images(SetOfSets, BinRel) ->
   Fun = fun(Set) -> sofs:image(BinRel, Set) end,
   sofs:substitution(Fun, SetOfSets).
```

External unordered sets are represented as sorted lists. So, creating the image of a set under a relation R can traverse all elements of R (to that comes the sorting of results, the image). In *`image/2`*, `BinRel` is traversed once for each element of `SetOfSets`, which can take too long. The following efficient function can be used instead under the assumption that the image of each element of `SetOfSets` under `BinRel` is non-empty:

```
images2(SetOfSets, BinRel) ->
   CR = sofs:canonical_relation(SetOfSets),
   R = sofs:relative_product1(CR, BinRel),
   sofs:relation_to_family(R).
```

`symdiff(Set1, Set2) -> Set3`

Types:

`Set1 = Set2 = Set3 = a_set()`

Returns the *symmetric difference* (or the Boolean sum) of `Set1` and `Set2`.

```
1> S1 = sofs:set([1,2,3]),
S2 = sofs:set([2,3,4]),
P = sofs:symdiff(S1, S2),
sofs:to_external(P).
[1,4]
```

`symmetric_partition(Set1, Set2) -> {Set3, Set4, Set5}`

Types:

`Set1 = Set2 = Set3 = Set4 = Set5 = 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`.

`to_external(AnySet) -> ExternalSet`

Types:

`ExternalSet = external_set()`
`AnySet = anyset()`

Returns the *external set* of an atomic, ordered, or unordered set.

`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 *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 provides functions for string processing.

A string in this module is represented by *`unicode:chardata()`*, that is, a list of codepoints, binaries with UTF-8-encoded codepoints (**UTF-8 binaries**), or a mix of the two.

```
"abcd"               is a valid string
<<"abcd">>           is a valid string
["abcd"]             is a valid string
<<"abc..åäö"/utf8>>  is a valid string
<<"abc..åäö">>       is NOT a valid string,
                     but a binary with Latin-1-encoded codepoints
[<<"abc">>, "..åäö"] is a valid string
[atom]               is NOT a valid string
```

This module operates on grapheme clusters. A **grapheme cluster** is a user-perceived character, which can be represented by several codepoints.

```
"å"  [229] or [97, 778]
"e#"  [101, 778]
```

The string length of "ß#e#" is 3, even though it is represented by the codepoints `[223,8593,101,778]` or the UTF-8 binary `<<195,159,226,134,145,101,204,138>>`.

Grapheme clusters for codepoints of class `prepend` and non-modern (or decomposed) Hangul is not handled for performance reasons in *`find/3`*, *`replace/3`*, *`split/2`*, *`split/2`* and *`trim/3`*.

Splitting and appending strings is to be done on grapheme clusters borders. There is no verification that the results of appending strings are valid or normalized.

Most of the functions expect all input to be normalized to one form, see for example *`unicode:characters_to_nfc_list/1`*.

Language or locale specific handling of input is not considered in any function.

The functions can crash for non-valid input strings. For example, the functions expect UTF-8 binaries but not all functions verify that all binaries are encoded correctly.

Unless otherwise specified the return value type is the same as the input type. That is, binary input returns binary output, list input returns a list output, and mixed input can return a mixed output.

```
1> string:trim("  sarah  ").
"sarah"
2> string:trim(<<"  sarah  ">>).
<<"sarah">>
3> string:lexemes("foo bar", " ").
["foo","bar"]
4> string:lexemes(<<"foo bar">>, " ").
[<<"foo">>,<<"bar">>]
```

This module has been reworked in Erlang/OTP 20 to handle *`unicode:chardata()`* and operate on grapheme clusters. The *`old functions`* that only work on Latin-1 lists as input are still available but should not be used. They will be deprecated in Erlang/OTP 21.

Data Types

direction() = leading | trailing

grapheme_cluster() = char() | [char()]

A user-perceived character, consisting of one or more codepoints.

Exports

casefold(String :: *unicode:chardata()*) -> *unicode:chardata()*

Converts `String` to a case-agnostic comparable string. Function `casefold/1` is preferred over `lowercase/1` when two strings are to be compared for equality. See also *`equal/4`*.

Example:

```
1> string:casefold("# and # SHARP S").
"# and ss sharp s"
```

chomp(String :: *unicode:chardata()*) -> *unicode:chardata()*

Returns a string where any trailing `\n` or `\r\n` have been removed from `String`.

Example:

```
182> string:chomp(<<"\nHello\n\n">>).
<<"\nHello">>
183> string:chomp("\nHello\r\r\n").
"\nHello\r"
```

equal(A, B) -> boolean()

equal(A, B, IgnoreCase) -> boolean()

equal(A, B, IgnoreCase, Norm) -> boolean()

Types:

A = B = *unicode:chardata()*

IgnoreCase = boolean()

Norm = none | nfc | nfd | nfkc | nfkd

Returns `true` if `A` and `B` are equal, otherwise `false`.

If `IgnoreCase` is `true` the function does *`casefold`*ing on the fly before the equality test.

If `Norm` is not `none` the function applies normalization on the fly before the equality test. There are four available normalization forms: *`nfc`*, *`nfd`*, *`nfkc`*, and *`nfkd`*.

By default, `IgnoreCase` is `false` and `Norm` is `none`.

Example:

```
1> string:equal("ääö", <<"ääö"/utf8>>).
true
2> string:equal("ääö", unicode:characters_to_nfd_binary("ääö")).
false
3> string:equal("ääö", unicode:characters_to_nfd_binary("ÄÄÖ"), true, nfc).
true
```

find(String, SearchPattern) -> *unicode:chardata()* | nomatch
find(String, SearchPattern, Dir) -> *unicode:chardata()* | nomatch

Types:

String = SearchPattern = *unicode:chardata()*
Dir = *direction()*

Removes anything before `SearchPattern` in `String` and returns the remainder of the string or `nomatch` if `SearchPattern` is not found. `Dir`, which can be `leading` or `trailing`, indicates from which direction characters are to be searched.

By default, `Dir` is `leading`.

Example:

```
1> string:find("ab..cd..ef", ".").
"..cd..ef"
2> string:find(<<"ab..cd..ef">>, "..", trailing).
<<"..ef">>
3> string:find(<<"ab..cd..ef">>, "x", leading).
nomatch
4> string:find("ab..cd..ef", "x", trailing).
nomatch
```

is_empty(String :: *unicode:chardata()*) -> boolean()

Returns `true` if `String` is the empty string, otherwise `false`.

Example:

```
1> string:is_empty("foo").
false
2> string:is_empty(["",<<>>]).
true
```

length(String :: *unicode:chardata()*) -> integer() >= 0

Returns the number of grapheme clusters in `String`.

Example:

```
1> string:length("ß#e#").
3
2> string:length(<<195,159,226,134,145,101,204,138>>).
3
```

lexemes(String :: *unicode:chardata()*,
SeparatorList :: [*grapheme_cluster()*]) ->
[*unicode:chardata()*]

Returns a list of lexemes in `String`, separated by the grapheme clusters in `SeparatorList`.

Notice that, as shown in this example, two or more adjacent separator graphemes clusters in `String` are treated as one. That is, there are no empty strings in the resulting list of lexemes. See also *`split/3`* which returns empty strings.

Notice that `[$\r,$\n]` is one grapheme cluster.

Example:

```
1> string:lexemes("abc de#fxxghix jkl\r\nfoo", "x e" ++ [[$\r,$\n]]).
["abc","de#f","ghi","jkl","foo"]
2> string:lexemes(<<"abc de#fxxghix jkl\r\nfoo"/utf8>>, "x e" ++ [$\r,$\n]).
[<<"abc">>,<<"de#f"/utf8>>,<<"ghi">>,<<"jkl\r\nfoo">>]
```

lowercase(String :: *unicode:chardata()*) -> *unicode:chardata()*

Converts `String` to lowercase.

Notice that function *`casefold/1`* should be used when converting a string to be tested for equality.

Example:

```
2> string:lowercase(string:uppercase("Micha#")).
"micha#"
```

next_codepoint(String :: *unicode:chardata()*) ->
maybe_improper_list(char(), *unicode:chardata()*) |
{error, *unicode:chardata()*}

Returns the first codepoint in `String` and the rest of `String` in the tail. Returns an empty list if `String` is empty or an `{error, String}` tuple if the next byte is invalid.

Example:

```
1> string:next_codepoint(unicode:characters_to_binary("e#fg")).
[101|<<"#fg"/utf8>>]
```

next_grapheme(String :: *unicode:chardata()*) ->
maybe_improper_list(*grapheme_cluster()*,
unicode:chardata()) |
{error, *unicode:chardata()*}

Returns the first grapheme cluster in `String` and the rest of `String` in the tail. Returns an empty list if `String` is empty or an `{error, String}` tuple if the next byte is invalid.

Example:

```
1> string:next_grapheme(unicode:characters_to_binary("e#fg")).
["e#"|<<"fg">>]
```

nth_lexeme(String, N, SeparatorList) -> *unicode:chardata()*

Types:

String = *unicode:chardata()*
N = integer() >= 0
SeparatorList = [*grapheme_cluster()*]

Returns lexeme number `N` in `String`, where lexemes are separated by the grapheme clusters in `SeparatorList`.

Example:

```
1> string:nth_lexeme("abc.de#f.ghiejkl", 3, ".e").
"ghi"
```

pad(String, Length) -> *unicode:charlist()*
pad(String, Length, Dir) -> *unicode:charlist()*
pad(String, Length, Dir, Char) -> *unicode:charlist()*

Types:

String = *unicode:chardata()*
Length = integer()
Dir = *direction()* | both
Char = *grapheme_cluster()*

Pads `String` to `Length` with grapheme cluster `Char`. `Dir`, which can be `leading`, `trailing`, or `both`, indicates where the padding should be added.

By default, `Char` is `$\s` and `Dir` is `trailing`.

Example:

```
1> string:pad(<<"He#llö"/utf8>>, 8).
[<<72,101,204,138,108,108,195,182>>,32,32,32]
2> io:format("'~ts'~n",[string:pad("He#llö", 8, leading)]).
'   He#llö'
3> io:format("'~ts'~n",[string:pad("He#llö", 8, both)]).
' He#llö  '
```

prefix(String :: *unicode:chardata()*, Prefix :: *unicode:chardata()*) ->
nomatch | *unicode:chardata()*

If `Prefix` is the prefix of `String`, removes it and returns the remainder of `String`, otherwise returns `nomatch`.

Example:

```
1> string:prefix(<<"prefix of string">>, "pre").
<<"fix of string">>
2> string:prefix("pre", "prefix").
nomatch
```

replace(String, SearchPattern, Replacement) ->
[*unicode:chardata()*]
replace(String, SearchPattern, Replacement, Where) ->
[*unicode:chardata()*]

Types:

String = SearchPattern = Replacement = *unicode:chardata()*
Where = *direction()* | all

Replaces `SearchPattern` in `String` with `Replacement`. `Where`, default `leading`, indicates whether the `leading`, the `trailing` or `all` encounters of `SearchPattern` are to be replaced.

Can be implemented as:

```
lists:join(Replacement, split(String, SearchPattern, Where)).
```

Example:

```
1> string:replace(<<"ab..cd..ef">>, "..", "*").
[<<"ab">>,"*",<<"cd..ef">>]
2> string:replace(<<"ab..cd..ef">>, "..", "*", all).
[<<"ab">>,"*",<<"cd">>,"*",<<"ef">>]
```

reverse(String :: *unicode:chardata()*) -> [*grapheme_cluster()*]

Returns the reverse list of the grapheme clusters in `String`.

Example:

```
1> Reverse = string:reverse(unicode:characters_to_nfd_binary("ÅÄÖ")).
[[79,776],[65,776],[65,778]]
2> io:format("~ts~n",[Reverse]).
O#A#A#
```

slice(String, Start) -> Slice
slice(String, Start, Length) -> Slice

Types:

String = *unicode:chardata()*
Start = integer() >= 0
Length = infinity | integer() >= 0
Slice = *unicode:chardata()*

Returns a substring of `String` of at most `Length` grapheme clusters, starting at position `Start`.

By default, `Length` is `infinity`.

Example:

```
1> string:slice(<<"He#llö Wörld"/utf8>>, 4).
<<"ö Wörld"/utf8>>
2> string:slice(["He#llö ", <<"Wörld"/utf8>>], 4,4).
"ö Wö"
3> string:slice(["He#llö ", <<"Wörld"/utf8>>], 4,50).
"ö Wörld"
```

split(String, SearchPattern) -> [*unicode:chardata()*]
split(String, SearchPattern, Where) -> [*unicode:chardata()*]

Types:

String = SearchPattern = *unicode:chardata()*
Where = *direction()* | all

Splits `String` where `SearchPattern` is encountered and return the remaining parts. `Where`, default `leading`, indicates whether the `leading`, the `trailing` or `all` encounters of `SearchPattern` will split `String`.

Example:

```
0> string:split("ab..bc..cd", "..").
["ab","bc..cd"]
1> string:split(<<"ab..bc..cd">>, "..", trailing).
[<<"ab..bc">>,<<"cd">>]
2> string:split(<<"ab..bc....cd">>, "..", all).
[<<"ab">>,<<"bc">>,<<>>,<<"cd">>]
```

take(String, Characters) -> {Leading, Trailing}
take(String, Characters, Complement) -> {Leading, Trailing}
take(String, Characters, Complement, Dir) -> {Leading, Trailing}

Types:

String = *unicode:chardata()*
Characters = [*grapheme_cluster()*]
Complement = boolean()
Dir = *direction()*
Leading = Trailing = *unicode:chardata()*

Takes characters from `String` as long as the characters are members of set `Characters` or the complement of set `Characters`. `Dir`, which can be `leading` or `trailing`, indicates from which direction characters are to be taken.

Example:

```
5> string:take("abc0z123", lists:seq($a,$z)).
{"abc","0z123"}
6> string:take(<<"abc0z123">>, lists:seq($0,$9), true, leading).
{<<"abc">>,<<"0z123">>}
7> string:take("abc0z123", lists:seq($0,$9), false, trailing).
{"abc0z","123"}
8> string:take(<<"abc0z123">>, lists:seq($a,$z), true, trailing).
{<<"abc0z">>,<<"123">>}
```

titlecase(String :: *unicode:chardata()*) -> *unicode:chardata()*

Converts `String` to titlecase.

Example:

```
1> string:titlecase("ß is a SHARP s").
"Ss is a SHARP s"
```

to_float(String) -> {Float, Rest} | {error, Reason}

Types:

String = *unicode:chardata()*
Float = float()
Rest = *unicode:chardata()*
Reason = no_float | badarg

Argument `String` is expected to start with a valid text represented float (the digits are ASCII values). Remaining characters in the string after the float are returned in `Rest`.

Example:

```
> {F1,Fs} = string:to_float("1.0-1.0e-1"),
> {F2,[]} = string:to_float(Fs),
> F1+F2.
0.9
> string:to_float("3/2=1.5").
{error,no_float}
> string:to_float("-1.5eX").
{-1.5,"eX"}
```

to_integer(String) -> {Int, Rest} | {error, Reason}

Types:

String = *unicode:chardata()*
Int = integer()
Rest = *unicode:chardata()*
Reason = no_integer | badarg

Argument `String` is expected to start with a valid text represented integer (the digits are ASCII values). Remaining characters in the string after the integer are returned in `Rest`.

Example:

```
> {I1,Is} = string:to_integer("33+22"),
> {I2,[]} = string:to_integer(Is),
> I1-I2.
11
> string:to_integer("0.5").
{0,".5"}
> string:to_integer("x=2").
{error,no_integer}
```

to_graphemes(String :: *unicode:chardata()*) -> [*grapheme_cluster()*]

Converts `String` to a list of grapheme clusters.

Example:

```
1> string:to_graphemes("ß#e#").
[223,8593,[101,778]]
2> string:to_graphemes(<<"ß#e#"/utf8>>).
[223,8593,[101,778]]
```

trim(String) -> *unicode:chardata()*
trim(String, Dir) -> *unicode:chardata()*
trim(String, Dir, Characters) -> *unicode:chardata()*

Types:

String = *unicode:chardata()*
Dir = *direction()* | both
Characters = [*grapheme_cluster()*]

Returns a string, where leading or trailing, or both, `Characters` have been removed. `Dir` which can be `leading`, `trailing`, or `both`, indicates from which direction characters are to be removed.

Default `Characters` are the set of nonbreakable whitespace codepoints, defined as Pattern_White_Space in **Unicode Standard Annex #31**. `By default, Dir` is `both`.

Notice that `[$\r,$\n]` is one grapheme cluster according to the Unicode Standard.

Example:

```
1> string:trim("\t  Hello  \n").
"Hello"
2> string:trim(<<"\t  Hello  \n">>, leading).
<<"Hello  \n">>
3> string:trim(<<".Hello.\n">>, trailing, "\n.").
<<".Hello">>
```

`uppercase(String :: unicode:chardata()) -> unicode:chardata()`

Converts `String` to uppercase.

See also *`titlecase/1`*.

Example:

```
1> string:uppercase("Micha#").
"MICHA#"
```

Obsolete API functions

Here follows the function of the old API. These functions only work on a list of Latin-1 characters.

> **Note:**
>
> The functions are kept for backward compatibility, but are not recommended. They will be deprecated in Erlang/OTP 21.
>
> Any undocumented functions in `string` are not to be used.

Exports

`centre(String, Number) -> Centered`

`centre(String, Number, Character) -> Centered`

Types:

```
String = Centered = string()
Number = integer() >= 0
Character = char()
```

Returns a string, where `String` is centered in the string and surrounded by blanks or `Character`. The resulting string has length `Number`.

This function is *obsolete*. Use *`pad/3`*.

`chars(Character, Number) -> String`

`chars(Character, Number, Tail) -> String`

Types:

```
Character = char()
Number = integer() >= 0
Tail = String = string()
```

Returns a string consisting of `Number` characters `Character`. Optionally, the string can end with string `Tail`.

This function is *obsolete*. Use *`lists:duplicate/2`*.

`chr(String, Character) -> Index`

Types:

```
String = string()
Character = char()
Index = integer() >= 0
```

Returns the index of the first occurrence of `Character` in `String`. Returns `0` if `Character` does not occur.

This function is *obsolete*. Use *`find/2`*.

`concat(String1, String2) -> String3`

Types:

```
String1 = String2 = String3 = string()
```

Concatenates `String1` and `String2` to form a new string `String3`, which is returned.

This function is *obsolete*. Use `[String1, String2]` as `Data` argument, and call *`unicode:characters_to_list/2`* or *`unicode:characters_to_binary/2`* to flatten the output.

`copies(String, Number) -> Copies`

Types:

```
String = Copies = string()
Number = integer() >= 0
```

Returns a string containing `String` repeated `Number` times.

This function is *obsolete*. Use *`lists:duplicate/2`*.

`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 not from `Chars`.

This function is *obsolete*. Use *`take/3`*.

Example:

```
> string:cspan("\t    abcdef", " \t").
0
```

`join(StringList, Separator) -> String`

Types:

StringList = [string()]
Separator = String = string()

Returns a string with the elements of `StringList` separated by the string in `Separator`.

This function is *obsolete*. Use *`lists:join/2`*.

Example:

```
> join(["one", "two", "three"], ", ").
"one, two, three"
```

left(String, Number) -> Left
left(String, Number, Character) -> Left

Types:

String = Left = string()
Number = integer() >= 0
Character = char()

Returns `String` with the length adjusted in accordance with `Number`. The left margin is fixed. If `length(String)` < `Number`, then `String` is padded with blanks or `Character`s.

This function is *obsolete*. Use *`pad/2`* or *`pad/3`*.

Example:

```
> string:left("Hello",10,$.).
"Hello....."
```

len(String) -> Length

Types:

String = string()
Length = integer() >= 0

Returns the number of characters in `String`.

This function is *obsolete*. Use *`length/1`*.

rchr(String, Character) -> Index

Types:

String = string()
Character = char()
Index = integer() >= 0

Returns the index of the last occurrence of `Character` in `String`. Returns `0` if `Character` does not occur.

This function is *obsolete*. Use *`find/3`*.

right(String, Number) -> Right
right(String, Number, Character) -> Right

Types:

String = Right = string()
Number = integer() >= 0
Character = char()

Returns `String` with the length adjusted in accordance with `Number`. The right margin is fixed. If the length of `(String)` < `Number`, then `String` is padded with blanks or `Character`s.

This function is *obsolete*. Use *`pad/3`*.

Example:

```
> string:right("Hello", 10, $.).
".....Hello"
```

rstr(String, SubString) -> Index

Types:

String = SubString = string()
Index = integer() >= 0

Returns the position where the last occurrence of `SubString` begins in `String`. Returns `0` if `SubString` does not exist in `String`.

This function is *obsolete*. Use *`find/3`*.

Example:

```
> string:rstr(" Hello Hello World World ", "Hello World").
8
```

span(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 `Chars`.

This function is *obsolete*. Use *`take/2`*.

Example:

```
> string:span("\t    abcdef", " \t").
5
```

str(String, SubString) -> Index

Types:

String = SubString = string()
Index = integer() >= 0

Returns the position where the first occurrence of `SubString` begins in `String`. Returns `0` if `SubString` does not exist in `String`.

This function is *obsolete*. Use *`find/2`*.

Example:

```
> string:str(" Hello Hello World World ", "Hello World").
8
```

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 or trailing, or both, blanks or a number of `Character` have been removed. `Direction`, which can be `left`, `right`, or `both`, indicates from which direction blanks are to be removed. `strip/1` is equivalent to `strip(String, both)`.

This function is *obsolete*. Use *`trim/3`*.

Example:

```
> string:strip("...Hello.....", both, $.).
"Hello"
```

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 position `Start` to the end of the string, or to and including position `Stop`.

This function is *obsolete*. Use *`slice/3`*.

Example:

```
sub_string("Hello World", 4, 8).
"lo Wo"
```

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 position `Start`, and ending at the end of the string or at length `Length`.

This function is *obsolete*. Use *`slice/3`*.

Example:

```
> substr("Hello World", 4, 5).
"lo Wo"
```

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 `Character`s.

This function is *obsolete*. Use *`nth_lexeme/3`*.

Example:

```
> string:sub_word(" Hello old boy !",3,$o).
"ld b"
```

`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 specified string or character is case-converted. Notice that the supported character set is ISO/IEC 8859-1 (also called Latin 1); all values outside this set are unchanged

This function is *obsolete* use *`lowercase/1`*, *`uppercase/1`*, *`titlecase/1`* or *`casefold/1`*.

`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`.

Example:

```
> tokens("abc defxxghix jkl", "x ").
["abc", "def", "ghi", "jkl"]
```

Notice that, as shown in this example, two or more adjacent separator characters in `String` are treated as one. That is, there are no empty strings in the resulting list of tokens.

This function is *obsolete*. Use *`lexemes/2`*.

`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`.

This function is *obsolete*. Use *`lexemes/2`*.

Example:

```
> words(" Hello old boy!", $o).
4
```

Notes

Some of the general string functions can seem to overlap each other. The reason is that this string package is the combination of two earlier packages and all functions of both packages have been retained.

supervisor

Erlang module

This behavior module provides a supervisor, a process that 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_server*, or *gen_statem* behaviors. A supervisor implemented using this module has a standard set of interface functions and include functionality for tracing and error reporting. Supervisors are used to build a hierarchical process structure called a supervision tree, a nice way to structure a fault-tolerant application. For more information, see *Supervisor Behaviour* in OTP Design Principles.

A supervisor expects the definition of which child processes to supervise to be specified in a callback module exporting a predefined set of functions.

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

Supervision Principles

The supervisor is responsible for starting, stopping, and monitoring its child processes. The basic idea of a supervisor is that it must keep its child processes alive by restarting them when necessary.

The children of a supervisor are 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.

The supervisor properties are defined by the supervisor flags. The type definition for the supervisor flags is as follows:

```
sup_flags() = #{strategy => strategy(),         % optional
                intensity => non_neg_integer(), % optional
                period => pos_integer()}        % optional
```

A supervisor can have one of the following **restart strategies** specified with the `strategy` key in the above map:

- `one_for_one` - If one child process terminates and is to be restarted, only that child process is affected. This is the default restart strategy.
- `one_for_all` - If one child process terminates and is to be restarted, all other child processes are terminated and then all child processes are restarted.
- `rest_for_one` - If one child process terminates and is to be restarted, the 'rest' of the child processes (that is, 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, that is, running the same code.

  Functions *`delete_child/2`* and *`restart_child/2`* are invalid for `simple_one_for_one` supervisors and return `{error,simple_one_for_one}` if the specified supervisor uses this restart strategy.

  Function *`terminate_child/2`* can be used for children under `simple_one_for_one` supervisors by specifying the child's `pid()` as the second argument. If instead the child specification identifier is used, `terminate_child/2` return `{error,simple_one_for_one}`.

  As a `simple_one_for_one` supervisor can have many children, it shuts them all down asynchronously. This means that the children do their cleanup in parallel, and therefore the order in which they are stopped is not defined.

To prevent a supervisor from getting into an infinite loop of child process terminations and restarts, a **maximum restart intensity** is defined using two integer values specified with keys `intensity` and `period` in the above map.

Assuming the values `MaxR` for `intensity` and `MaxT` for `period`, then, if more than `MaxR` restarts occur within `MaxT` seconds, the supervisor terminates all child processes and then itself. The termination reason for the supervisor itself in that case will be `shutdown`. `intensity` defaults to `1` and `period` defaults to `5`.

The type definition of a child specification is as follows:

```
child_spec() = #{id => child_id(),       % mandatory
                 start => mfargs(),      % mandatory
                 restart => restart(),   % optional
                 shutdown => shutdown(), % optional
                 type => worker(),       % optional
                 modules => modules()}   % optional
```

The old tuple format is kept for backwards compatibility, see *child_spec()*, but the map is preferred.

- `id` is used to identify the child specification internally by the supervisor.

  The `id` key is mandatory.

  Notice that this identifier on occations has been called "name". As far as possible, the terms "identifier" or "id" are now used but to keep backward compatibility, some occurences of "name" can still be found, for example in error messages.
- `start` defines the function call used to start the child process. It must be a module-function-arguments tuple `{M,F,A}` used as `apply(M,F,A)`.

  The start function **must create and link to** the child process, and must return `{ok,Child}` or `{ok,Child,Info}`, where `Child` is the pid of the child process and `Info` any term that 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 is kept by the supervisor (unless it is a temporary child) but the non-existing child process is ignored.

  If something goes wrong, the function can also return an error tuple `{error,Error}`.

  Notice that the `start_link` functions of the different behavior modules fulfill the above requirements.

  The `start` key is mandatory.
- `restart` defines when a terminated child process must be restarted. A `permanent` child process is always restarted. A `temporary` child process is never 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). A `transient` child process is restarted only if it terminates abnormally, that is, with another exit reason than `normal`, `shutdown`, or `{shutdown,Term}`.

  The `restart` key is optional. If it is not specified, it defaults to `permanent`.
- `shutdown` defines how a child process must be terminated. `brutal_kill` means that the child process is unconditionally terminated using `exit(Child,kill)`. An integer time-out value means that the supervisor tells the child process to terminate by calling `exit(Child,shutdown)` and then wait for an exit signal with reason `shutdown` back from the child process. If no exit signal is received within the specified number of milliseconds, the child process is unconditionally terminated using `exit(Child,kill)`.

  If the child process is another supervisor, the shutdown time is to be set to `infinity` to give the subtree ample time to shut down. It is also allowed to set it to `infinity`, if the child process is a worker.

> **Warning:**
>
> Be careful when setting the shutdown time 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.

Notice that all child processes implemented using the standard OTP behavior modules automatically adhere to the shutdown protocol.

The `shutdown` key is optional. If it is not specified, it defaults to `5000` if the child is of type `worker` and it defaults to `infinity` if the child is of type `supervisor`.

- `type` specifies if the child process is a supervisor or a worker.

  The `type` key is optional. If it is not specified, it defaults to `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, if the child process is a `supervisor`, `gen_server` or, `gen_statem`, this is to be a list with one element `[Module]`, where `Module` is the callback module. If the child process is an event manager (`gen_event`) with a dynamic set of callback modules, value `dynamic` must be used. For more information about release handling, see *Release Handling* in OTP Design Principles.

  The `modules` key is optional. If it is not specified, it defaults to `[M]`, where `M` comes from the child's start `{M,F,A}`.
- 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(),
      start := mfargs(),
      restart => restart(),
      shutdown => shutdown(),
      type => worker(),
      modules => modules()} |
    {Id :: child_id(),
     StartFunc :: mfargs(),
     Restart :: restart(),
     Shutdown :: shutdown(),
     Type :: worker(),
     Modules :: modules()}
```

The tuple format is kept for backward compatibility only. A map is preferred; see more details *above*.

```
mfargs() =
    {M :: module(), F :: atom(), A :: [term()] | undefined}
```

Value `undefined` for `A` (the argument list) is only to be used internally in `supervisor`. If the restart type of the child is `temporary`, the process is never to be restarted and therefore there is no need to store the real argument list. Value `undefined` is then stored instead.

```
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_flags() =
    #{strategy => strategy(),
      intensity => integer() >= 0,
      period => integer() >= 1} |
    {RestartStrategy :: strategy(),
     Intensity :: integer() >= 0,
     Period :: integer() >= 1}
```

The tuple format is kept for backward compatibility only. A map is preferred; see more details *above*.

```
sup_ref() =
    (Name :: atom()) |
    {Name :: atom(), Node :: node()} |
    {global, Name :: atom()} |
    {via, Module :: module(), Name :: any()} |
    pid()
worker() = worker | supervisor
```

Exports

check_childspecs(ChildSpecs) -> Result

Types:

```
ChildSpecs = [child_spec()]
Result = ok | {error, Error :: term()}
```

Takes a list of child specification as argument and returns `ok` if all of them are syntactically correct, otherwise `{error,Error}`.

count_children(SupRef) -> PropListOfCounts

Types:

```
SupRef = sup_ref()
PropListOfCounts = [Count]
Count =
    {specs, ChildSpecCount :: integer() >= 0} |
    {active, ActiveProcessCount :: integer() >= 0} |
    {supervisors, ChildSupervisorCount :: integer() >= 0} |
    {workers, ChildWorkerCount :: integer() >= 0}
```

Returns a property list (see *`proplists`*) containing the counts for each of the following elements of the supervisor's child specifications and managed processes:

- `specs` - The total count of children, dead or alive.
- `active` - The count of all actively running child processes managed by this supervisor. For a `simple_one_for_one` supervisors, no check is done to ensure that each child process is still alive, although the result provided here is likely to be very accurate unless the supervisor is heavily overloaded.

- `supervisors` - The count of all children marked as `child_type = supervisor` in the specification list, regardless if the child process is still alive.
- `workers` - The count of all children marked as `child_type = worker` in the specification list, regardless if the child process is still alive.

For a description of `SupRef`, see *`start_child/2`*.

`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 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.

For a description of `SupRef`, see *`start_child/2`*.

If successful, the function returns `ok`. If the child specification identified by `Id` exists but the corresponding child process is running or is 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}`.

`get_childspec(SupRef, Id) -> Result`

Types:

```
SupRef = sup_ref()
Id = pid() | child_id()
Result = {ok, child_spec()} | {error, Error}
Error = not_found
```

Returns the child specification map for the child identified by `Id` under supervisor `SupRef`. The returned map contains all keys, both mandatory and optional.

For a description of `SupRef`, see *`start_child/2`*.

`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 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.

Notice that for temporary children, the child specification is automatically deleted when the child terminates; thus, it is not possible to restart such children.

For a description of `SupRef`, see *`start_child/2`*.

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.

`start_child(SupRef, ChildSpec) -> startchild_ret()`

Types:

```
SupRef = sup_ref()
ChildSpec = child_spec() | (List :: [term()])
startchild_ret() =
    {ok, Child :: child()} |
    {ok, Child :: child(), Info :: term()} |
    {error, startchild_err()}
startchild_err() =
    already_present | {already_started, Child :: child()} | term()
```

Dynamically adds a child specification to supervisor `SupRef`, which starts the corresponding child process.

`SupRef` can be any of the following:

- The pid
- `Name`, if the supervisor is locally registered
- `{Name,Node}`, if the supervisor is locally registered at another node
- `{global,Name}`, if the supervisor is globally registered
- `{via,Module,Name}`, if the supervisor is registered through an alternative process registry

`ChildSpec` must be a valid child specification (unless the supervisor is a `simple_one_for_one` supervisor; see below). The child process is started by using the start function as defined in the child specification.

For a `simple_one_for_one` supervisor, the child specification defined in `Module:init/1` is used, and `ChildSpec` must instead be an arbitrary list of terms `List`. The child process is then started by appending `List` to the existing start function arguments, that is, 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 identifier, `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 are 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 (unless the supervisor is a `simple_one_for_one` supervisor, see below), the pid is set to `undefined`, and the function returns `{ok,undefined}`.

For a `simple_one_for_one` supervisor, when a child process start function returns `ignore`, the functions returns `{ok,undefined}` and no child is added to the supervisor.

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.

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()} |
    {via, Module :: module(), Name :: any()}
```

Creates a supervisor process as part of a supervision tree. For example, the function ensures 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 intensity, and child processes. To ensure a synchronized startup 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 must export the functions `register_name/2`, `unregister_name/1`, and `send/2`, which must 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 any term that is passed as the argument to `Module:init/1`.

- If the supervisor and its child processes are successfully created (that is, if all child process start functions return `{ok,Child}`, `{ok,Child,Info}`, or `ignore`), the function returns `{ok,Pid}`, where `Pid` is the pid of the supervisor.
- If there already exists a process with the specified `SupName`, the function returns `{error, {already_started,Pid}}`, where `Pid` is the pid of that process.
- If `Module:init/1` returns `ignore`, this function returns `ignore` as well, and the supervisor terminates with reason `normal`.
- If `Module:init/1` fails or returns an incorrect value, this function returns `{error,Term}`, where `Term` is a term with information about the error, and the supervisor terminates with reason `Term`.
- If any child process start function fails or returns an error tuple or an erroneous value, the supervisor first terminates all already started child processes with reason `shutdown` and then terminate itself and returns `{error, {shutdown, Reason}}`.

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 supervisor `SupRef` to terminate the specified child.

If the supervisor is not `simple_one_for_one`, `Id` must be the child specification identifier. The process, if any, is terminated and, unless it is a temporary child, the child specification is kept by the supervisor. The child process can 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` cannot be used for these children.

If the supervisor is `simple_one_for_one`, `Id` must be the `pid()` of the child process. If the specified process is alive, but is not a child of the specified supervisor, the function returns `{error,not_found}`. If the child specification identifier is specified instead of a `pid()`, the function returns `{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}`.

For a description of `SupRef`, see *`start_child/2`*.

`which_children(SupRef) -> [{Id, Child, Type, Modules}]`

Types:

```
SupRef = sup_ref()
Id = child_id() | undefined
Child = child() | restarting
Type = worker()
Modules = modules()
```

Returns a newly created list with information about all child specifications and child processes belonging to supervisor `SupRef`.

Notice that calling this function when supervising many childrens under low memory conditions can cause an out of memory exception.

For a description of `SupRef`, see *`start_child/2`*.

The following information is given for each child specification/process:

- `Id` - As defined in the child specification or `undefined` for a `simple_one_for_one` supervisor.
- `Child` - The pid of the corresponding child process, the atom `restarting` if the process is about to be restarted, or `undefined` if there is no such process.
- `Type` - As defined in the child specification.
- `Modules` - As defined in the child specification.

Callback Functions

The following function must be exported from a `supervisor` callback module.

Exports

`Module:init(Args) -> Result`

Types:

`Args = term()`

`Result = {ok,{SupFlags,[ChildSpec]}} | ignore`

`SupFlags = sup_flags()`

`ChildSpec = child_spec()`

Whenever a supervisor is started using *`start_link/2,3`*, this function is called by the new process to find out about restart strategy, maximum restart intensity, and child specifications.

`Args` is the `Args` argument provided to the start function.

`SupFlags` is the supervisor flags defining the restart strategy and maximum restart intensity for the supervisor. `[ChildSpec]` is a list of valid child specifications defining which child processes the supervisor must start and monitor. See the discussion in section *`Supervision Principles`* earlier.

Notice 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 child specification identifier is ignored.) No child process is then started during the initialization phase, but all children are assumed to be started dynamically using *`start_child/2`*.

The function can also return `ignore`.

Notice that this function can also be called as a part of a code upgrade procedure. Therefore, the function is not to have any side effects. For more information about code upgrade of supervisors, see section *Changing a Supervisor* in OTP Design Principles.

See Also

`gen_event(3)`, *`gen_statem(3)`*, *`gen_server(3)`*, *`sys(3)`*

supervisor_bridge

Erlang module

This behavior module provides a supervisor bridge, a process that 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. For more information, see *Supervisor Behaviour* in OTP Design Principles.

A supervisor bridge assumes the functions for starting and stopping the subsystem to be located in a callback module exporting a predefined set of functions.

The *sys(3)* module can be used for debugging a supervisor bridge.

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

Exports

start_link(Module, Args) -> Result

start_link(SupBridgeName, Module, Args) -> Result

Types:

```
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 startup 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 is to export functions `register_name/2`, `unregister_name/1`, and `send/2`, which are to 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.

`Module` is the name of the callback module.

`Args` is an arbitrary term that is passed as the argument to `Module:init/1`.

- If the supervisor bridge and the subsystem are successfully started, the function returns `{ok,Pid}`, where `Pid` is is the pid of the supervisor bridge.
- If there already exists a process with the specified `SupBridgeName`, 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 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`.

Callback Functions

The following functions must 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 *`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 is to 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 terminates with reason `Reason` as well. If later the supervisor bridge is stopped by its supervisor with reason `Reason`, it calls `Module:terminate(Reason,State)` to terminate.

If the initialization fails, the function is to 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 is to be the opposite of `Module:init/1` and stop the subsystem and do any necessary cleaning up. The return value is ignored.

`Reason` is `shutdown` if the supervisor bridge is terminated by its supervisor. If the supervisor bridge terminates because a a linked process (apart from the main process of the subsystem) has terminated with reason `Term`, then `Reason` becomes `Term`.

`State` is taken from the return value of `Module:init/1`.

See Also

`supervisor(3)`, *`sys(3)`*

sys

Erlang module

This module contains functions for sending system messages used by programs, and messages used for debugging purposes.

Functions used for implementation of processes are also expected to understand system messages, such as debug messages and code change. These functions must be used to implement the use of system messages for a process; either directly, or through standard behaviors, such as *`gen_server`*.

The default time-out is 5000 ms, unless otherwise specified. `timeout` defines the time to wait for the process to respond to a request. If the process does not respond, the function evaluates `exit({timeout, {M, F, A}})`.

The functions make references to a debug structure. The debug structure is a list of `dbg_opt()`, which is an internal data type used by function *`handle_system_msg/6`*. No debugging is performed if it is an empty list.

System Messages

Processes that are not implemented as one of the standard behaviors must still understand system messages. The following three messages must be understood:

- Plain system messages. These are received as `{system, From, Msg}`. The content and meaning of this message are not interpreted by the receiving process module. When a system message is received, function *`handle_system_msg/6`* is called to handle the request.
- Shutdown messages. If the process traps exits, it must be able to handle a shutdown request from its parent, the supervisor. The message `{'EXIT', Parent, Reason}` from the parent is an order to terminate. The process must terminate when this message is received, normally with the same `Reason` as `Parent`.
- If the modules used to implement the process change dynamically during runtime, the process must understand one more message. An example is the *`gen_event`* processes. The message is `{_Label, {From, Ref}, get_modules}`. The reply to this message is `From ! {Ref, Modules}`, where `Modules` is a list of the currently active modules in the process.

  This message is used by the release handler to find which processes that execute a certain module. The process can later be suspended and ordered to perform a code change for one of its modules.

System Events

When debugging a process with the functions of this module, the process generates **system_events**, which are then treated in the debug function. For example, `trace` formats the system events to the terminal.

Three predefined system events 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

```
name() =
    pid() | atom() | {global, term()} | {via, module(), term()}
system_event() =
    {in, Msg :: term()} |
    {in, Msg :: term(), From :: term()} |
    {out, Msg :: term(), To :: term()} |
```

```
    term()
dbg_opt()
```

See the introduction of this manual page.

```
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

```
change_code(Name, Module, OldVsn, Extra) -> ok | {error, Reason}
change_code(Name, Module, OldVsn, Extra, Timeout) ->
               ok | {error, Reason}
```

Types:

```
Name = name()
Module = module()
OldVsn = undefined | term()
Extra = term()
Timeout = timeout()
Reason = term()
```

Tells the process to change code. The process must be suspended to handle this message. Argument `Extra` is reserved for each process to use as its own. Function `Module:system_code_change/4` is called. `OldVsn` is the old version of the `Module`.

```
get_state(Name) -> State
get_state(Name, Timeout) -> State
```

Types:

```
Name = name()
Timeout = timeout()
State = term()
```

Gets the state of the process.

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 the state from the return values of *`get_status/1`* or *`get_status/2`* while debugging.

The value of `State` varies for different types of processes, as follows:

- For a *`gen_server`* process, the returned `State` is the state of the callback module.

- For a *gen_statem* process, State is the tuple {CurrentState,CurrentData}.
- For a *gen_event* process, State is a list of tuples, where each tuple corresponds to an event handler registered in the process and contains {Module, Id, HandlerState}, as follows:

  Module

  The module name of the event handler.

  Id

  The ID of the handler (which is false if it was registered without an ID).

  HandlerState

  The state of the handler.

If the callback module exports a function system_get_state/1, it is 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 function *Module:system_get_state/1* is expected to extract the state of the callback module from it. Function system_get_state/1 must return {ok, State}, where State is the state of the callback module.

If the callback module does not export a system_get_state/1 function, get_state/1,2 assumes that the Misc value is the state of the callback module 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.

Function system_get_state/1 is primarily useful for user-defined behaviors and modules that implement OTP *special processes*. The gen_server, gen_statem, and gen_event OTP behavior modules export this function, so callback modules for those behaviors need not to supply their own.

For more information about a process, including its state, see *get_status/1* and *get_status/2*.

get_status(Name) -> Status

get_status(Name, Timeout) -> Status

Types:

```
Name = name()
Timeout = timeout()
Status =
    {status, Pid :: pid(), {module, Module :: module()}, [SItem]}
SItem =
    (PDict :: [{Key :: term(), Value :: term()}]) |
    (SysState :: running | suspended) |
    (Parent :: pid()) |
    (Dbg :: [dbg_opt()]) |
    (Misc :: term())
```

Gets the status of the process.

The value of Misc varies for different types of processes, for example:

- A *gen_server* process returns the state of the callback module.
- A *gen_statem* process returns information, such as its current state name and state data.
- A *gen_event* process returns information about each of its registered handlers.

Callback modules for gen_server, gen_statem, and gen_event can also change the value of Misc by exporting a function format_status/2, which contributes module-specific information.

For details, see *gen_server:format_status/2*, *gen_statem:format_status/2*, and *gen_event:format_status/2*.

install(Name, FuncSpec) -> ok
install(Name, FuncSpec, Timeout) -> ok

Types:

Name = *name()*
FuncSpec = {Func, FuncState}
Func = *dbg_fun()*
FuncState = term()
Timeout = timeout()

Enables installation of alternative debug functions. 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. For example, turning on low-level tracing.

`Func` is called whenever a system event is generated. This function is to return `done`, or a new `Func` state. In the first case, the function is removed. It is also removed if the function fails.

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 (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 *`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 text 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 *`handle_debug/4`*). The file is opened with encoding UTF-8.

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 are installed explicitly with function *install/2,3*, for example, triggers.

remove(Name, Func) -> ok
remove(Name, Func, Timeout) -> ok

Types:

Name = *name()*
Func = *dbg_fun()*
Timeout = timeout()

Removes an installed debug function from the process. `Func` must be the same as previously installed.

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 are not to be called from normal code. They are provided for convenience, allowing developers to avoid having to create their own custom state replacement functions.

Function `StateFun` provides a new state for the process. Argument `State` and the `NewState` return value of `StateFun` vary for different types of processes as follows:

- For a *gen_server* process, `State` is the state of the callback module and `NewState` is a new instance of that state.
- For a *gen_statem* process, `State` is the tuple `{CurrentState,CurrentData}`, and `NewState` is a similar tuple, which can contain a new current state, new state data, or both.
- For a *gen_event* process, `State` is the tuple `{Module, Id, HandlerState}` as follows:

  `Module`

  The module name of the event handler.

  `Id`

  The ID of the handler (which is `false` if it was registered without an ID).

  `HandlerState`

  The state of the handler.

  `NewState` is a similar tuple where `Module` and `Id` are to have the same values as in `State`, but the value of `HandlerState` can be different. Returning a `NewState`, whose `Module` or `Id` values differ from those of

State, leaves the state of the event handler 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 can return its State argument.

If a StateFun function crashes or throws an exception, the original state of the process is unchanged for gen_server, and gen_statem processes. 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 is 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 new state of the callback module, obtained by calling StateFun, and NewMisc is a possibly new value used to replace the original Misc (required as Misc often contains the state of the callback module within it).

If the callback module does not export a system_replace_state/2 function, *replace_state/2,3* assumes that Misc is the state of the callback module, 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 function system_replace_state/2 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}}.

Function system_replace_state/2 is primarily useful for user-defined behaviors and modules that implement OTP *special processes*. The OTP behavior modules gen_server, gen_statem, and gen_event export this function, so callback modules for those behaviors need not to supply their own.

```
resume(Name) -> ok
resume(Name, Timeout) -> ok
```

Types:

```
Name = name()
Timeout = timeout()
```

Resumes a suspended process.

```
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:date_time()
    Timeout = timeout()
```

Enables or disables the collection of statistics. If `Flag` is `get`, the statistical collection is returned.

```
suspend(Name) -> ok
suspend(Name, Timeout) -> ok
```

Types:

```
    Name = name()
    Timeout = timeout()
```

Suspends the process. When the process is suspended, it only responds to other system messages, but not other messages.

```
terminate(Name, Reason) -> ok
terminate(Name, Reason, Timeout) -> ok
```

Types:

```
    Name = name()
    Reason = term()
    Timeout = timeout()
```

Orders the process to terminate with the specified `Reason`. The termination is done asynchronously, so it is not guaranteed that the process is terminated when the function returns.

```
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 *`handle_debug/4`*).

Process Implementation Functions

The following functions are used when implementing a special process. This is an ordinary process, which does not use a standard behavior, but a process that 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()
```

Can be used by a process that initiates a debug structure from a list of options. The values of argument `Opt` are the same as for the corresponding functions.

```
get_debug(Item, Debug, Default) -> term()
```

Types:

```
Item = log | statistics
Debug = [dbg_opt()]
Default = term()
```

Gets the data associated with a debug option. `Default` is returned if `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, called as `FormFunc(Device, Event, Extra)` to print the events, which is necessary if tracing is activated. `Extra` is any extra information that the process needs in the format function, for example, the process name.

```
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 to take care of system messages. The process receives a `{system, From, Msg}` message and passes `Msg` and `From` to this function.

This function **never** returns. It calls either of the following functions:

- `Module:system_continue(Parent, NDebug, Misc)`, where the process continues the execution.
- `Module:system_terminate(Reason, Parent, Debug, Misc)`, if the process is to terminate.

`Module` must export the following:

- system_continue/3
- system_terminate/4
- system_code_change/4
- system_get_state/1
- system_replace_state/2

Argument Misc 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*.

Module:system_code_change(Misc, Module, OldVsn, Extra) -> {ok, NMisc}

Types:

Misc = term()
OldVsn = undefined | term()
Module = atom()
Extra = term()
NMisc = term()

Called from *handle_system_msg/6* when the process is to perform a code change. The code change is used when the internal data structure has changed. This function converts argument Misc to the new data structure. OldVsn is attribute **vsn** of the old version of the Module. If no such attribute is defined, the atom undefined is sent.

Module:system_continue(Parent, Debug, Misc) -> none()

Types:

Parent = pid()
Debug = [*dbg_opt()*]
Misc = term()

Called from *handle_system_msg/6* when the process is to continue its execution (for example, after it has been suspended). This function never returns.

Module:system_get_state(Misc) -> {ok, State}

Types:

Misc = term()
State = term()

Called from *handle_system_msg/6* when the process is to return a term that reflects its current state. State is the value returned by *get_state/2*.

Module:system_replace_state(StateFun, Misc) -> {ok, NState, NMisc}

Types:

StateFun = fun((State :: term()) -> NState)
Misc = term()

```
NState = term()
NMisc = term()
```

Called from *handle_system_msg/6* when the process is to replace its current state. `NState` is the value returned by *replace_state/3*.

Module:system_terminate(Reason, Parent, Debug, Misc) -> none()

Types:

```
Reason = term()
Parent = pid()
Debug = [dbg_opt()]
Misc = term()
```

Called from *handle_system_msg/6* when the process is to terminate. For example, this function is called when the process is suspended and its parent orders shutdown. It gives the process a chance to do a cleanup. This function never returns.

timer

Erlang module

This module provides useful functions related to time. Unless otherwise stated, time is always measured in **milliseconds**. All timer functions return immediately, regardless of work done by another process.

Successful evaluations of the timer functions give return values containing a timer reference, denoted `TRef`. By using *`cancel/1`*, the returned reference can be used to cancel any requested action. A `TRef` is an Erlang term, which contents must not be changed.

The time-outs are not exact, but are **at least** as long as requested.

Data Types

`time() = integer() >= 0`

Time in milliseconds.

`tref()`

A timer reference.

Exports

```
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` milliseconds.

Returns `{ok, TRef}` or `{error, Reason}`.

```
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}`.

cancel(TRef) -> {ok, cancel} | {error, Reason}

Types:

TRef = *tref()*

Reason = term()

Cancels a previously requested time-out. `TRef` is a unique timer reference returned by the related timer function.

Returns `{ok, cancel}`, or `{error, Reason}` when `TRef` is not a timer reference.

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/2` is the same as `exit_after(Time, self(), Reason1)`.

`exit_after/3` sends an exit signal with reason `Reason1` to pid `Pid`. Returns `{ok, TRef}` or `{error, Reason2}`.

hms(Hours, Minutes, Seconds) -> MilliSeconds

Types:

Hours = Minutes = Seconds = MilliSeconds = integer() >= 0

Returns the number of milliseconds in `Hours + Minutes + Seconds`.

hours(Hours) -> MilliSeconds

Types:

Hours = MilliSeconds = integer() >= 0

Returns the number of milliseconds in `Hours`.

kill_after(Time) -> {ok, TRef} | {error, Reason2}

kill_after(Time, Pid) -> {ok, TRef} | {error, Reason2}

Types:

Time = *time()*

Pid = pid() | (RegName :: atom())

TRef = *tref()*

Reason2 = term()

`kill_after/1` is the same as `exit_after(Time, self(), kill)`.

`kill_after/2` is the same as `exit_after(Time, Pid, kill)`.

minutes(Minutes) -> MilliSeconds

Types:

Minutes = MilliSeconds = integer() >= 0

Returns the number of milliseconds in `Minutes`.

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` are time-stamp tuples on the same format as returned from *`erlang:timestamp/0`* or *`os:timestamp/0`*.

seconds(Seconds) -> MilliSeconds

Types:

Seconds = MilliSeconds = integer() >= 0

Returns the number of milliseconds in `Seconds`.

send_after(Time, Message) -> {ok, TRef} | {error, Reason}
send_after(Time, Pid, Message) -> {ok, TRef} | {error, Reason}

Types:

Time = *time()*
Pid = pid() | (RegName :: atom())
Message = term()
TRef = *tref()*
Reason = term()

`send_after/3`

Evaluates `Pid ! Message` after `Time` milliseconds. (`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)`.

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` milliseconds. (`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)`.

`sleep(Time) -> ok`

Types:

`Time = timeout()`

Suspends the process calling this function for `Time` milliseconds and then returns `ok`, or suspends the process forever if `Time` is the atom `infinity`. Naturally, this function does **not** return immediately.

`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 is to be started explicitly. Use configuration parameters for *Kernel* for this.

`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 the same as `tc/3`.

`tc/1`

Evaluates `Fun()`. Otherwise the same as `tc/2`.

Examples

Example 1

The following example shows how to print "Hello World!" in 5 seconds:

```
1> timer:apply_after(5000, io, format, ["~nHello World!~n", []]).
{ok,TRef}
Hello World!
```

Example 2

The following example shows a process performing a certain action, and if this action is not completed within a certain limit, 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),
...
```

Notes

A timer can always be removed by calling `cancel/1`.

An interval timer, that is, a timer created by evaluating any of the functions `apply_interval/4`, `send_interval/3`, and `send_interval/2` is linked to the process to which the timer performs its task.

A one-shot timer, that is, 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 time-out, or if it is explicitly removed by a call to `cancel/1`.

unicode

Erlang module

This module contains functions for converting between different character representations. It converts between ISO Latin-1 characters and Unicode characters, 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 code point for the character.

Other Unicode encodings than integers representing code points 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. ISO Latin-1 encoding is supported both for backward compatibility and for communication with external entities not supporting Unicode character sets.

Programs should always operate on a normalized form and compare canonical-equivalent Unicode characters as equal. All characters should thus be normalized to one form once on the system borders. One of the following functions can convert characters to their normalized forms *`characters_to_nfc_list/1`*, *`characters_to_nfc_binary/1`*, *`characters_to_nfd_list/1`* or *`characters_to_nfd_binary/1`*. For general text *`characters_to_nfc_list/1`* or *`characters_to_nfc_binary/1`* is preferred, and for identifiers one of the compatibility normalization functions, such as *`characters_to_nfkc_list/1`*, is preferred for security reasons. The normalization functions where introduced in OTP 20. Additional information on normalization can be found in the **Unicode FAQ**.

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 (that is, 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 a valid ISO Latin-1 character (0-255).

```
latin1_chardata() = latin1_charlist() | latin1_binary()
```

Same as `iodata()`.

```
latin1_charlist() =
    maybe_improper_list(latin1_char() |
                        latin1_binary() |
                        latin1_charlist(),
                        latin1_binary() | [])
```

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
```

Checks for a UTF Byte Order Mark (BOM) in the beginning of a binary. If the supplied binary `Bin` begins with a valid BOM for either UTF-8, UTF-16, or UTF-32, the function returns the encoding identified along with the BOM length in bytes.

If no BOM is found, the function returns `{latin1,0}`.

```
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 a binary instead of a Unicode list.

`InEncoding` defines how input is to be interpreted if binaries are present in `Data`

`OutEncoding` defines in what format output is to be generated.

Options:

`unicode`

An alias for `utf8`, as this is the preferred encoding for Unicode characters in binaries.

`utf16`

An alias for `{utf16,big}`.

`utf32`

An alias for `{utf32,big}`.

The atoms `big` and `little` denote big- or little-endian encoding.

Errors and exceptions occur as in *`characters_to_list/2`*, but the second element in tuple `error` or `incomplete` is a `binary()` and not a `list()`.

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 can have characters encoded as one of the following:

- ISO Latin-1 (0-255, one character per byte). Here, case parameter `InEncoding` is to be specified as `latin1`.
- One of the UTF-encodings, which is specified as parameter `InEncoding`.

Note that integers in the list always represent code points regardless of `InEncoding` passed. If `InEncoding` `latin1` is passed, only code points < 256 are allowed; otherwise, all valid unicode code points are allowed.

If `InEncoding` is `latin1`, parameter `Data` corresponds to the `iodata()` type, but for `unicode`, parameter `Data` can contain integers > 255 (Unicode characters beyond the ISO Latin-1 range), which makes it invalid as `iodata()`.

The purpose of the function is mainly 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.

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 atoms `big` and `little` denote big- or little-endian encoding.

If the data cannot be converted, either because of illegal Unicode/ISO Latin-1 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 or mixed list, or both, not necessarily of the same depth as the original data. The error occurs when traversing the list and whatever is left to decode is 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 > 255 is found in the lists.

  If `InEncoding` is of a Unicode type, an error occurs whenever either of the following is found:

  - An integer > 16#10FFFF (the maximum Unicode character)
  - An integer in the range 16#D800 to 16#DFFF (invalid range reserved for UTF-16 surrogate pairs)
- Incorrect UTF encoding.

  If `InEncoding` is one of the UTF types, the bytes in any binaries must be valid in that encoding.

  Errors can occur for various reasons, including the following:

  - "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.

- 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 can occur if bytes are read from a file in chunks or if binaries in other ways are split on non-UTF character boundaries. An `incomplete` tuple is then 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 character 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 specified as input without errors occurring.

Example:

```
decode_data(Data) ->
   case unicode:characters_to_list(Data,unicode) of
      {incomplete,Encoded, Rest} ->
            More = get_some_more_data(),
            Encoded ++ decode_data([Rest, More]);
      {error,Encoded,Rest} ->
            handle_error(Encoded,Rest);
      List ->
            List
   end.
```

However, bit strings that are not whole bytes are not allowed, so a UTF character must be split along 8-bit boundaries to ever be decoded.

A `badarg` exception is thrown for the following cases:

- Any parameters are of the wrong type.
- The list structure is invalid (a number as tail).
- The binaries do not contain whole bytes (bit strings).

`characters_to_nfc_list(CD :: `*`chardata()`*`) ->`
`                        [char()] | {error, [char()], `*`chardata()`*`}`

Converts a possibly deep list of characters and binaries into a Normalized Form of canonical equivalent Composed characters according to the Unicode standard.

Any binaries in the input must be encoded with utf8 encoding.

The result is a list of characters.

```
3> unicode:characters_to_nfc_list([<<"abc..a">>,[778],$a,[776],$o,[776]]).
"abc..åäö"
```

`characters_to_nfc_binary(CD :: `*`chardata()`*`) ->`
`                          `*`unicode_binary()`*` |`
`                          {error, `*`unicode_binary(), chardata()`*`}`

Converts a possibly deep list of characters and binaries into a Normalized Form of canonical equivalent Composed characters according to the Unicode standard.

Any binaries in the input must be encoded with utf8 encoding.

The result is an utf8 encoded binary.

```
4> unicode:characters_to_nfc_binary([<<"abc..a">>,[778],$a,[776],$o,[776]]).
<<"abc..ääö"/utf8>>
```

characters_to_nfd_list(CD :: *chardata()*) -> [char()] | {error, [char()], *chardata()*}

Converts a possibly deep list of characters and binaries into a Normalized Form of canonical equivalent Decomposed characters according to the Unicode standard.

Any binaries in the input must be encoded with utf8 encoding.

The result is a list of characters.

```
1> unicode:characters_to_nfd_list("abc..ääö").
[97,98,99,46,46,97,778,97,776,111,776]
```

characters_to_nfd_binary(CD :: *chardata()*) -> *unicode_binary()* | {error, *unicode_binary()*, *chardata()*}

Converts a possibly deep list of characters and binaries into a Normalized Form of canonical equivalent Decomposed characters according to the Unicode standard.

Any binaries in the input must be encoded with utf8 encoding.

The result is an utf8 encoded binary.

```
2> unicode:characters_to_nfd_binary("abc..ääö").
<<97,98,99,46,46,97,204,138,97,204,136,111,204,136>>
```

characters_to_nfkc_list(CD :: *chardata()*) -> [char()] | {error, [char()], *chardata()*}

Converts a possibly deep list of characters and binaries into a Normalized Form of compatibly equivalent Composed characters according to the Unicode standard.

Any binaries in the input must be encoded with utf8 encoding.

The result is a list of characters.

```
3> unicode:characters_to_nfkc_list([<<"abc..a">>,[778],$a,[776],$o,[776],[65299,65298]]).
"abc..ääö32"
```

characters_to_nfkc_binary(CD :: *chardata()*) -> *unicode_binary()* | {error, *unicode_binary()*, *chardata()*}

Converts a possibly deep list of characters and binaries into a Normalized Form of compatibly equivalent Composed characters according to the Unicode standard.

Any binaries in the input must be encoded with utf8 encoding.

The result is an utf8 encoded binary.

```
4> unicode:characters_to_nfkc_binary([<<"abc..a">>,[778],$a,[776],$o,[776],[65299,65298]]).
<<"abc..ääö32"/utf8>>
```

characters_to_nfkd_list(CD :: *chardata()*) ->
[char()] |
{error, [char()], *chardata()*}

Converts a possibly deep list of characters and binaries into a Normalized Form of compatibly equivalent Decomposed characters according to the Unicode standard.

Any binaries in the input must be encoded with utf8 encoding.

The result is a list of characters.

```
1> unicode:characters_to_nfkd_list(["abc..åäö",[65299,65298]]).
[97,98,99,46,46,97,778,97,776,111,776,51,50]
```

characters_to_nfkd_binary(CD :: *chardata()*) ->
***unicode_binary()* |**
{error, *unicode_binary()*, *chardata()*}

Converts a possibly deep list of characters and binaries into a Normalized Form of compatibly equivalent Decomposed characters according to the Unicode standard.

Any binaries in the input must be encoded with utf8 encoding.

The result is an utf8 encoded binary.

```
2> unicode:characters_to_nfkd_binary(["abc..åäö",[65299,65298]]).
<<97,98,99,46,46,97,204,138,97,204,136,111,204,136,51,50>>
```

encoding_to_bom(InEncoding) -> Bin

Types:

Bin = binary()

A `binary()` such that `byte_size(Bin) >= 4`.

InEncoding = *encoding()*

Creates 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 `latin1` encoding, as there is no BOM for ISO Latin-1.

Notice that the BOM for UTF-8 is seldom used, and it is really not a **byte order** mark. There are obviously no byte order issues with UTF-8, so the BOM is only there to differentiate UTF-8 encoding from other UTF formats.

win32reg

Erlang module

This module 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 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 subkeys to the right and backslash between keys. (Remember that backslashes must be doubled in Erlang strings.) Case is preserved but not significant.

For 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 this module as follows:

```
Abbreviation     Registry key
============     ============
hkcr             HKEY_CLASSES_ROOT
current_user     HKEY_CURRENT_USER
hkcu             HKEY_CURRENT_USER
local_machine    HKEY_LOCAL_MACHINE
hklm             HKEY_LOCAL_MACHINE
users            HKEY_USERS
hku              HKEY_USERS
current_config   HKEY_CURRENT_CONFIG
hkcc             HKEY_CURRENT_CONFIG
dyn_data         HKEY_DYN_DATA
hkdd             HKEY_DYN_DATA
```

The key above can be written as `"\\hklm\\software\\ericsson\\erlang\\5.0"`.

This module uses a current key. It works much like the current directory. From the current key, values can be fetched, subkeys can be listed, and so on.

Under a key, any number of named values can be stored. They have names, types, and data.

`win32reg` supports storing of the following types:

- `REG_DWORD`, which is an integer
- `REG_SZ`, which is a string
- `REG_BINARY`, which is a binary

Other types can be read, and are 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, for example, `%SystemRoot%Windows`. `SystemRoot` is an environment variable, and is to be replaced with its value. Function *`expand/1`* is provided so that environment variables surrounded by `%` can be expanded to their values.

For more information on the Windows registry, see 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`.

Notice that the current key is stored in the driver, and can be invalid (for example, 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()`. Notice 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 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 an environment variable, and is replaced by the value. Two consecutive % are replaced by one %.

A variable name that is not in the environment results in an error.

`format_error(ErrorId) -> ErrorString`

Types:

`ErrorId = atom()`
`ErrorString = string()`

Converts a POSIX error code to a string (by calling `erl_posix_msg:message/1`).

`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 is the root (`HKEY_CLASSES_ROOT`). Flag `read` 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 is used. The supported types are the following:

- `REG_DWORD` for integers
- `REG_SZ` for strings
- `REG_BINARY` for binaries

Other types cannot 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, as 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/2`*. Calls the Win32 API function `EnumRegValuesEx()`.

See Also

`erl_posix_msg`, The Windows 95 Registry (book from O'Reilly), Win32 Programmer's Reference (from Microsoft)

zip

Erlang module

This module archives and extracts files to and from a zip archive. The zip format is specified by the "ZIP Appnote.txt" file, available on the PKWARE web site **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 is to end with `.zip`. To abide to the convention, add `.zip` to the filename.

- To create zip archives, use function *`zip/2`* or *`zip/3`*. They are also available as `create/2,3`, to resemble the *`erl_tar`* module.
- To extract files from a zip archive, use function *`unzip/1`* or *`unzip/2`*. They are also available as `extract/1,2`, to resemble the *`erl_tar`* module.
- To fold a function over all files in a zip archive, use function *`foldl/3`*.
- To return a list of the files in a zip archive, use function *`list_dir/1`* or *`list_dir/2`*. They are also available as `table/1,2`, to resemble the *`erl_tar`* module.
- To print a list of files to the Erlang shell, use function *`t/1`* or *`tt/1`*.
- Sometimes it is desirable to open a zip archive, and to unzip files from it file by file, without having to reopen the archive. This can be done by functions *`zip_open/1,2`*, *`zip_get/1,2`*, *`zip_list_dir/1`*, and *`zip_close/1`*.

Limitations

- Zip64 archives are not supported.
- Password-protected and encrypted archives are not supported.
- Only the DEFLATE (zlib-compression) and the STORE (uncompressed data) zip methods are supported.
- The archive size is limited to 2 GB (32 bits).
- Comments for individual files are not supported when creating zip archives. The zip archive comment for the whole zip archive is supported.
- Changing a zip archive is not supported. To add or remove a file from an archive, the whole archive must be recreated.

Data Types

```
zip_comment() = #zip_comment{comment = string()}
```

The record `zip_comment` only contains the archive comment for a zip archive.

```
zip_file() =
    #zip_file{name = string(),
              info = file:file_info(),
              comment = string(),
              offset = integer() >= 0,
              comp_size = integer() >= 0}
```

The record `zip_file` contains the following fields:

`name`

The filename

`info`

File information as in *`file:read_file_info/1`* in Kernel

`comment`

The comment for the file in the zip archive

`offset`

The file offset in the zip archive (used internally)

`comp_size`

The size of the compressed file (the size of the uncompressed file is found in `info`)

filename() = *file:filename()*

The name of a zip file.

```
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*.

handle()

As returned by *zip_open/2*.

Exports

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()
```

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 information about the file.

`GetBin` returns the file contents.

Both `GetInfo` and `GetBin` must be called within the `Fun`. Their behavior is undefined if they are called outside the context of `Fun`.

The `Fun` must return a new accumulator, which is passed to the next call. `foldl/3` returns the final accumulator value. `Acc0` is returned if the archive is empty. It is not necessary to iterate over all files in the archive. The iteration can be ended prematurely in a controlled manner by throwing an exception.

Example:

```
> Name = "dummy.zip".
"dummy.zip"
> {ok, {Name, Bin}} = zip:create(Name, [{"foo", <<"FOO">>}, {"bar", <<"BAR">>}], [memory]).
{ok,{"dummy.zip",
     <<80,75,3,4,20,0,0,0,0,0,74,152,97,60,171,39,212,26,3,0,
       0,0,3,0,0,...>>}}
> {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}}]}
> {ok, {Name, Bin}} = zip:create(Name, lists:reverse(FileSpec), [memory]).
{ok,{"dummy.zip",
     <<80,75,3,4,20,0,0,0,0,0,74,152,97,60,171,39,212,26,3,0,
       0,0,3,0,0,...>>}}
> catch zip:foldl(fun("foo", _, B, _) -> throw(B()); (_,_,_,Acc) -> Acc end, [], {Name, Bin}).
<<"FOO">>
```

`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`

`list_dir/1` retrieves all filenames in the zip archive `Archive`.

`list_dir/2` provides options.

`table/1` and `table/2` are provided as synonyms to resemble the *`erl_tar`* module.

The result value is the tuple `{ok, List}`, where `List` contains the zip archive comment as the first element.

One option is available:

`cooked`

By default, this function opens 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 overrides the default and opens the zip file without option `raw`.

`t(Archive) -> ok`

Types:

```
Archive = file:name() | binary() | ZipHandle
ZipHandle = handle()
```

Prints all filenames in the zip archive `Archive` to the Erlang shell. (Similar to `tar t`.)

`tt(Archive) -> ok`

Types:

```
Archive = file:name() | binary() | ZipHandle
ZipHandle = handle()
```

Prints filenames and information about all files in the zip archive `Archive` to the Erlang shell. (Similar to `tar tv`.)

`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()}}
```

`unzip/1` extracts all files from a zip archive.

`unzip/2` provides options to extract some files, and more.

`extract/1` and `extract/2` are provided as synonyms to resemble module *`erl_tar`*.

If argument `Archive` is specified as a binary, the contents of the binary is assumed to be a zip archive, otherwise a filename.

Options:

`{file_list, FileList}`

By default, all files are extracted from the zip archive. With option `{file_list, FileList}`, function `unzip/2` only extracts the files whose names are included in `FileList`. The full paths, including the names of all subdirectories within the zip archive, must be specified.

`cooked`

By default, this function opens 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 overrides the default and opens the zip file without option `raw`. The same applies for the files extracted.

`keep_old_files`

By default, all files with the same name as files in the zip archive are overwritten. With option `keep_old_files` set, function `unzip/2` does not overwrite existing files. Notice that even with option `memory` specified, which means that no files are overwritten, existing files are excluded from the result.

`verbose`

Prints an informational message for each extracted file.

`memory`

Instead of extracting to the current directory, the result is given as a list of tuples `{Filename, Binary}`, where `Binary` is a binary containing the extracted data of file `Filename` in the zip archive.

`{cwd, CWD}`

Uses the specified directory as current directory. It is prepended to filenames when extracting them from the zip archive. (Acting like *`file:set_cwd/1`* in Kernel, but without changing the global `cwd` property.)

```
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()}
```

Creates a zip archive containing the files specified in `FileList`.

`create/2` and `create/3` are provided as synonyms to resemble module *`erl_tar`*.

`FileList` is a list of files, with paths relative to the current directory, which are stored with this path in the archive. Files can also be specified with data in binaries to create an archive directly from data.

Files are compressed using the DEFLATE compression, as described in the "Appnote.txt" file. However, files are stored without compression if they are already compressed. `zip/2` and `zip/3` check the file extension to determine if the file is to 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 control what types of files that are to be compressed by using options `{compress, What}` and `{uncompress, What}`. It is also possible to use many `compress` and `uncompress` options.

To trigger file compression, 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 extension `"gif"` are compressed.

Options:

`cooked`

By default, this function opens the zip file in mode `raw`, which is faster but does not allow a remote (Erlang) file server to be used. Adding `cooked` to the mode list overrides the default and opens the zip file without the `raw` option. The same applies for the files added.

`verbose`

Prints an informational message about each added file.

`memory`

The output is not to a file, but instead as a tuple `{FileName, binary()}`. The binary is a full zip archive with header and can be extracted with, for example, *`unzip/2`*.

`{comment, Comment}`

Adds a comment to the zip archive.

`{cwd, CWD}`

Uses the specified directory as current work directory (`cwd`). This is prepended to filenames when adding them, although not in the zip archive (acting like *`file:set_cwd/1`* in Kernel, but without changing the global `cwd` property.).

`{compress, What}`

Controls what types of files to be compressed. Defaults to `all`. The following values of `What` are allowed:

`all`

All files are compressed (as long as they pass the `uncompress` condition).

`[Extension]`

Only files with exactly these extensions are 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 to be uncompressed. Defaults to `[".Z", ".zip", ".zoo", ".arc", ".lzh", ".arj"]`. The following values of `What` are allowed:

`all`

No files are compressed.

`[Extension]`

Files with these extensions are uncompressed.

`{add,[Extension]}`

Adds these extensions to the list of uncompress extensions.

`{del,[Extension]}`

Deletes these extensions from the list of uncompress extensions.

`zip_close(ZipHandle) -> ok | {error, einval}`

Types:

`ZipHandle = handle()`

Closes a zip archive, previously opened with *`zip_open/1,2`*. All resources are closed, and the handle is not to be used after closing.

`zip_get(ZipHandle) -> {ok, [Result]} | {error, Reason}`
`zip_get(FileName, ZipHandle) -> {ok, Result} | {error, Reason}`

Types:

`FileName = file:name()`
`ZipHandle = handle()`
`Result = file:name() | {file:name(), binary()}`
`Reason = term()`

Extracts one or all files from an open archive.

The files are unzipped to memory or to file, depending on the options specified to function *`zip_open/1,2`* when opening the archive.

`zip_list_dir(ZipHandle) -> {ok, Result} | {error, Reason}`

Types:

`Result = [zip_comment() | zip_file()]`
`ZipHandle = handle()`
`Reason = term()`

Returns the file list of an open zip archive. The first returned element is the zip archive comment.

`zip_open(Archive) -> {ok, ZipHandle} | {error, Reason}`
`zip_open(Archive, Options) -> {ok, ZipHandle} | {error, Reason}`

Types:

```
Archive = file:name() | binary()
ZipHandle = handle()
Options = [Option]
Option = cooked | memory | {cwd, CWD :: file:filename()}
Reason = term()
```

Opens a zip archive, and reads and saves its directory. This means that later reading files from the archive is faster than unzipping files one at a time with `unzip/1,2`.

The archive must be closed with `zip_close/1`.

The `ZipHandle` is closed if the process that originally opened the archive dies.