# Compiler Application (COMPILER)

version 1.3

Robert Virding

1997-05-02

Typeset in LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

# Contents

1 COMPILER Reference Manual 1
1.1 compile (Module) . . . . . 2

# COMPILER Reference Manual

## Short Summaries

- Erlang Module compile [page 2] – Erlang Compiler

## compile

The following functions are exported:

- `file(File)`
  [page 2] Compiles a file
- `file(File, Options) -> CompRet`
  [page 2] Compiles a file
- `forms(Forms)`
  [page 3] Compiles a list of forms
- `forms(Forms, Options) -> CompRet`
  [page 4] Compiles a list of forms
- `format_error(ErrorDescriptor) -> string()`
  [page 4] Formats an error descriptor

# compile (Module)

This module provides an interface to the standard Erlang compiler. It can generate either a new file which contains the object code, or return a binary which can be loaded directly.

## Exports

`file(File)`

Is the same as `file(File, [verbose,report_errors,report_warnings])`.

`file(File, Options) -> CompRet`

Types:

- CompRet = ModRet | BinRet | ErrRet
- ModRet = {ok,ModuleName} | {ok,ModuleName,Warnings}
- BinRet = {ok,ModuleName,Binary} | {ok,ModuleName,Binary,Warnings}
- ErrRet = error | {error,Errors,Warnings}

Compiles the code in the file `File`, which is an Erlang source code file without the `.erl` extension. `Options` determine the behavior of the compiler.

Returns {`ok,ModuleName`} if successful, or `error` if there are errors. An object code file is created if the compilation succeeds with no errors.

The elements of `Options` can be selected as follows:

`binary` Causes the compiler to return the object code in a binary instead of creating an object file. If successful, the compiler returns {`ok,ModuleName,Binary`}

`'P'` Produces a listing of the parsed code after preprocessing and parse transforms, in the file <`File`>`.P`. No object file is produced.

`dexp` Produces a listing of the expanded code in the file <`File`>`.expand`. No object file is produced.

`dpe` Produces a listing of the code after partial evaluation, in the file <`File`>`.parteval`. No object file is produced.

`'E'` Produces a listing of the code after all source code transformations have been performed, in the file <`File`>`.E`. No object file is produced.

`dcg` For Beam, produces a listing of the intermediate code after code generation, in the file <`File`>`.codegen`. No object file is produced.

`dopt` For Beam, produces a listing of the intermediate code after optimization, in the file <`File`>`.optimize`. No object file is produced.

'S' Produces a listing of the assembler code in the file <File>.S. No object file is produced.

trace Produces slightly slower code that can be traced function by function with the use of the BIF erlang:trace/3.

report_errors/report_warnings Causes errors/warnings to be printed as they occur.

report This is a short form for both report_errors and report_warnings.

return_errors If this flag is set, then {error,ErrorList,WarningList} is returned when there are errors.

return_warnings If this flag is set, then an extra field containing WarningList is added to the tuples returned on success.

return This is a short form for both return_errors and return_warnings.

verbose Causes more verbose information from the compiler describing what it is doing.

{outdir,Dir} Sets a new directory for the object code. The current directory is used for output, except when a directory has been specified with this option.

export_all Causes all functions in the module to be exported.

{i,Dir} Add Dir to the list of directories to be searched when including a file.

{d,Macro}

{d,Macro,Value} Defines a macro Macro to have the value Value. The default is true).

{parse_transform,Module} Causes the parse transformation function Module:parse_transform/2 to be applied to the parsed code before the code is checked for errors.

jam The compiler will generate code for JAM.

beam The compiler will generate code for BEAM.

asm The input file is expected to be assembler code (default file suffix ".S"). Only works for BEAM. Note that the format of assembler files is not documented, and may change between releases - this option is primarily for internal debugging use.

Note that all the options except the include path can also be given in the file with a -compile([Option,...]). attribute.

The single-atom options beginning with a 'd', as well as 'P', 'E' and 'S' provide a way to look at the code as produced by certain passes in the compiler. In the list above, they are listed in the order in which the passes occur. If more than one such option is used, the one representing the earliest pass takes effect.

*Unrecognized options are ignored.*

Both WarningList and ErrorList have the following format:

[{FileName,[ErrorInfo]}].

ErrorInfo is described below. The file name has been included here as the compiler uses the Erlang pre-processor epp, which allows the code to be included in other files. For this reason, it is important to know to *which* file an error or warning line number refers.

forms(Forms)

Is the same as forms(File, [verbose,report_errors,report_warnings]).

`forms(Forms, Options) -> CompRet`

Types:

- Forms = [Form]
- CompRet = ModRet | BinRet | ErrRet
- ModRet = {ok,ModuleName} | {ok,ModuleName,Warnings}
- BinRet = {ok,ModuleName,Binary} | {ok,ModuleName,Binary,Warnings}
- ErrRet = error | {error,Errors,Warnings}

Analogous to `file/1`, but takes a list of forms (in the Erlang abstract format representation) as first argument. The option `binary` is implicit; i.e., no object code file is produced. If the options indicate that a listing file should be produced (e.g., 'E'), the module name is taken as the file name.

`format_error(ErrorDescriptor) -> string()`

Types:

- ErrorDescriptor = errordesc()

Uses an `ErrorDescriptor` and returns a string which describes the error. This function is usually called implicitly when an `ErrorInfo` structure is processed. See below.

## Parse Transformations

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

This type of activity is strongly discouraged.

## Error Information

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

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

A string describing the error is obtained with the following call:

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

## See Also

epp, erl_id_trans

## Bugs

If Erlang crashes during compilation, `<File>.ja#` files may be created in the current directory. Such files can be safely deleted.

# Index

Modules are typed in *this way*.
Functions are typed in `this way`.

*compile*
    `file/1`, 2
    `file/2`, 2
    `format_error/1`, 4
    `forms/1`, 3
    `forms/2`, 4

`file/1`
    *compile* , 2

`file/2`
    *compile* , 2

`format_error/1`
    *compile* , 4

`forms/1`
    *compile* , 3

`forms/2`
    *compile* , 4