Compiler Application (COMPILER)

version 4.3

Typeset in LaTeX from SGML source using the DOCBUILDER 3.3.2 Document System.

Contents

Compiler Reference Manual

Short Summaries

- Erlang Module **compile** [page 2] – Erlang Compiler

compile

The following functions are exported:

- `file(File)`
  [page 2] Compile a file
- `file(File, Options) -> CompRet`
  [page 2] Compile a file
- `forms(Forms)`
  [page 6] Compile a list of forms
- `forms(Forms, Options) -> CompRet`
  [page 6] Compile a list of forms
- `format_error(ErrorDescriptor) -> chars()`
  [page 6] Format an error descriptor

compile

Erlang 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.

Here follows first all elements of `Options` that in some way control the behavior of the compiler.

`basic_validation` This option is fast way to test whether a module will compile successfully (mainly useful for code generators that want to verify the code they emit). No code will generated. If warnings are enabled, warnings generated by the `erl_lint` module (such as warnings for unused variables and functions) will be returned too.

Use the `strong_validation` option to generate all warnings that the compiler would generate.

`strong_validation` Similar to the `basic_validation` option, no code will be generated, but more compiler passes will be run to ensure also warnings generated by the optimization passes are generated (such as clauses that will not match or expressions that are guaranteed to fail with an exception at run-time).

`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}`

`debug_info` Include debug information in the compiled beam module. Examle of Erlang/OTP applications that can use the debug information are Debugger, Xref, and Cover.

Warning: Note that the source code can be reconstructed from the abstract code. If it is important to keep the source code secret, use the `debug_info_key` option (described next) to encrypt the debug information, or strip the debug information using the [beam_lib] module before shipping your code.

`{debug_info_key,KeyString}`

`{debug_info_key,{Mode,KeyString}}` Include debug information, but encrypt it, so that it cannot be accessed without supplying the key. (To give the `debug_info` option as well is allowed, but is not necessary.) Using this option is a good way to always have the debug information available during testing, yet protect the source code.

`Mode` is the type of crypto algorithm to be used for encrypting the debug information. The default mode, and currently the only, is `des3_cbc` (three rounds of DES). The `KeyString` will be scrambled (using `erlang:md5`) to generate the actual keys used for `des3_cbc`. It is recommended that the key string contains at least 32 characters, and that both upper and lower case letters as well as digits and special characters are used.

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.

See the [beam_lib] module on how to register a key so that utilities such as Xref or the Debugger can access the debug information.

`encrypt_debug_info` Like the `debug_info_key` option above, except that the key will be read from an `.erlang.crypt` file. See the [beam_lib] module for syntax of the `.erlang.crypt` file.

`'P'` Produces a listing of the parsed code after preprocessing and parse transforms, in the file `<File>.P`. 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.

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

`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. When encountering an `-include` or `-include_dir` directive, the compiler searches for header files in the following directories:

1. ".", the current working directory of the file server;
2. the base name of the compiled file;
3. the directories specified using the `i` option. The directory specified last is searched first.

{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.

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

`ignore_try` `try` is a reserved keyword from the R9 release and may not be used as atom names or field names in records (unless single-quoted). To compile old code where `try` is used, the `ignore_try` option can be given.

`ignore_cond` `cond` is a reserved keyword starting with the R9 release and may not be used as atom names or field names in records (unless single-quoted). To compile old code where `cond` is used, the `ignore_cond` option can be given.

`strict_record_tests` By default (for historical reasons), the generated code the `Record#record_tag.field` operation will not check that the tuple `Record` indeed is a record having the tag `record_tag`. Use the `strict_record_tests` option to emit code that verifies that the first element of the tuple is the record tag and that the size of the tuple is the expected size. Currently, the tests will not be emitted for record operations in guards.

If the test fails, a `badmatch` exception will be generated, as opposed to the `badrecord` exception that all other record operations generate when they fail. (A future release of the compiler probably will emit code that generates a `badrecord` exception.)

The performance penalty for using this option is very slight or none at all if there are further uses of the same record in the same function. The compiler in most cases optimizes away redundant record tests, so that only the first use of a record in a function results in a record test. Note that all other record operations already test the record type.

`strict_record_tests` will probably be default in R11B.

If warnings are turned on (the `report_warnings` option described above), the following options control what type of warnings that will be generated. With the exception of {`warn_format,Verbosity`} all options below have two forms; one `warn_xxx` form to turn on the warning and one `nowarn_xxx` form to turn off the warning. In the description that follows, the form that is used to change the default value is listed.

{`warn_format, Verbosity`} Causes warnings to be emitted for malformed format strings as arguments to `io:format` and similar functions. `Verbosity` selects the amount of warnings: 0 = no warnings; 1 = warnings for invalid format strings and incorrect number of arguments; 2 = warnings also when the validity could not be checked (for example, when the format string argument is a variable). The default verbosity is 1. Verbosity 0 can also be selected by the option `nowarn_format`.

`nowarn_bif_clash` By default, a warning will be emitted when a module contains an exported function with the same name as an auto-imported BIF (such as `size/1`) AND there is a call to it without a qualifying module name. The reason is that the BIF will be called, not the function in the same module. The recommended way to eliminate that warning is to use a call with a module name - either `erlang` to call the BIF or `?MODULE` to call the function in the same module. The warning can also be turned off using `nowarn_bif_clash`, but that is not recommended.

`warn_export_vars` Causes warnings to be emitted for all implicitly exported variables referred to after the primitives where they were first defined. No warnings for exported variables unless they are referred to in some pattern, which is the default, can be selected by the option `nowarn_export_vars`.

`warn_shadow_vars` Causes warnings to be emitted for "fresh" variables in functional objects or list comprehensions with the same name as some already defined variable. The default is to warn for such variables. No warnings for shadowed variables can be selected by the option `nowarn_shadow_vars`.

`nowarn_unused_function` Turns off warnings for unused local functions. By default (`warn_unused_function`), warnings are emitted for all local functions that are not called directly or indirectly by an exported function. The compiler does not include unused local functions in the generated beam file, but the warning is still useful to keep the source code cleaner.

`warn_unused_import` Causes warnings to be emitted for unused imported functions. No warnings for imported functions, which is the default, can be selected by the option `nowarn_unused_import`.

`nowarn_unused_vars` By default, warnings are emitted for variables which are not used, with the exception of variables beginning with an underscore ("Prolog style warnings"). Use this option to turn off this kind of warnings.

Another class of warnings (introduced in the R10B release) are generated by the compiler during optimization and code generation. They warn about patterns that will never match (such as `a=b`), guards that will always evaluate to false, and expressions that will always fail (such as `atom+42`). Currently, those warnings cannot be disabled (except by disabling all warnings).

> **Warning:**
> Obviously, the absence of warnings does not mean that there are no remaining errors in the code.

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

For debugging of the compiler, or for pure curiosity, the intermediate code generated by each compiler pass can be inspected. A complete list of the options to produce list files can be printed by typing `compile:options()` at the Erlang shell prompt. The options will be printed in order that the passes are executed. If more than one listing 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 = BinRet | ErrRet
- BinRet = {ok,ModuleName,BinaryOrCode} | {ok,ModuleName,BinaryOrCode,Warnings}
- BinaryOrCode = binary() | term() <V>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. Options that would ordinarily produce a listing file, such as 'E', will instead cause the internal format for that compiler pass (an Erlang term; usually not a binary) to be returned instead of a binary.

`format_error(ErrorDescriptor) -> chars()`

Types:

- ErrorDescriptor = errordesc()

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

Default compiler options

The (host operating system) environment variable `ERL_COMPILER_OPTIONS` can be used to give default compiler options. Its value must be a valid Erlang term. If the value is a list, it will be used as is. If it is not a list, it will be put into a list. The list will be appended to any options given to `file/2` or `forms/2`.

Inlining

The compiler can now do function inlining within an Erlang module. Inlining means that a call to a function is replaced with the function body with the arguments replaced with the actual values. The semantics are preserved, except if exceptions are generated in the inlined code. Exceptions will be reported as occurring in the function the body was inlined into. Also, `function_clause` exceptions will be converted to similar `case_clause` exceptions.

When a function is inlined, the original function may be kept as a separate function as well, because there might still be calls to it. Therefore, inlining almost always increases code size.

Inlining does not necessarily improve running time. For instance, inlining may increase Beam stack usage which will probably be detrimental to performance for recursive functions.

Inlining is never default; it must be explicitly enabled with a compiler option or a '`-compile()`' attribute in the source module.

To enable inlining, use the '`inline`' option.

Example:

```
-compile(inline).
```

The '`{inline_size,Size}`' option controls how large functions that are allowed to be inlined. Default is 24, which will keep the size of the inlined code roughly the same as the un-inlined version (only relatively small functions will be inlined).

Example:

```
%% Aggressive inlining - will increase code size.
-compile(inline).
-compile({inline_size,100}).
```

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.

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(3)], [erl_id_trans(3)], [erl_lint(3)], [beam_lib(3)]

Index of Modules and Functions

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