Errortrace: Debugging and Profiling

Version 9.4.0.5

November 9, 2026

Errortrace is a stack-trace-on-exceptions, profiler, and coverage tool for Racket. It is not a complete debugger; DrRacket provides more. Meanwhile, using Errortrace might be better than Racket's limited stack-trace reporting.

Contents

1 Quick Instructions **3**

2 Installing Errortrace **4**

3 Using Errortrace **5**

3.1 Instrumentation and Profiling 5

3.2 Coverage 7

3.3 Other Errortrace Bindings 8

4 Errortrace Library **10**

5 Re-using Errortrace Stack Tracing **12**

6 Programmatically Extracting Stack-trace Information **19**

7 Errortrace Key **20**

1 Quick Instructions

First, throw away `".zo"` versions of your program—at least for the modules or files that should be instrumented for error reporting or profiling.

Then,

- If your program has a module file ⟨*prog*⟩, run it with

  `racket -l errortrace -t` ⟨*prog*⟩

- If your program has a module file ⟨*prog*⟩ with a submodule ⟨*sub*⟩, run it with

  `racket -l racket/init -l errortrace -e '(require (submod` ⟨*prog*⟩ ⟨*sub*⟩`))'`

- If your program is a non-module top-level sequence of definitions and expressions, you can instead add

  ```
  (require errortrace)
  ```

  to the beginning of the program, or start Racket with the `-l` option before the arguments to load your program:

  `racket -l racket/init -l errortrace -f` ⟨*prog*⟩

- If you have no main program and you want to use Racket interactively, include the `-i` flag before `-l`:

  ```
  racket -i -l errortrace
  ```

  The order of the flags is important.

- To instrument the contents of a collection or package, compile it with:

  ```
  raco setup -j 1 --mode errortrace ...
  ```

  Note that because `--mode` doesn't support parallel builds, `-j 1` is required unless you are building packages on a single-core machine

After starting `errortrace` in one of these ways, when an exception occurs, the exception handler prints something like a stack trace with most recent contexts first.

The `errortrace` module is strange: Don't import it into another module. Instead, the `errortrace` module is meant to be invoked from the top-level, so that it can install an evaluation handler, exception handler, etc.

To reuse parts of the code of `errortrace`, import `errortrace/errortrace-lib`. That library contains all of the bindings described here, but does not set the compilation handler or the error display handler.

2 Installing Errortrace

Invoking the `errortrace` module sets the compilation handler to instrument Racket source code. It also sets the error display handler to report source information for an exception, and it sets the `use-compiled-file-paths` parameter to trigger the use of Errortrace-specific `".zo"` files.

NOTE: `errortrace` has no effect on code loaded as compiled byte code (i.e., from a `".zo"` file) or native code (i.e., from a `".dll"`, `".so"` or `".dylib"` file). You can use the `--mode errortrace` flag to `raco setup` to create `".zo"` files with Errortrace information.

Explicitly requiring `errortrace` within a module is generally a bad idea, since `errortrace` sets various parameters.

3 Using Errortrace

```
(require errortrace)
```
package: errortrace-lib

See §1 “Quick Instructions” for information on starting with errortrace. This chapter provides information on the configuration of errortrace after it is loaded and installed.

Don’t import errortrace into another module and expect it to work on that module. Instead, the errortrace module is meant to be invoked from the top-level (as described in §1 “Quick Instructions”) so it can install handlers. The functions documented in this chapter then can be used at the top-level. The functions also can be accessed by importing errortrace/errortrace-lib, which does not install any handlers.

As a language name, errortrace chains to another language that is specified immediately after errortrace, but instruments the module for debugging in the same way as if errortrace is required before loading the module from source. Using the errortrace meta-language is one way to ensure that debugging instrumentation is present when the module is compiled.

3.1 Instrumentation and Profiling

By default, errortrace only instruments for stack-trace-on-exception. Profiling and coverage need to be enabled separately.

```
(instrumenting-enabled) → boolean?
(instrumenting-enabled on?) → void?
  on? : any/c
```

A parameter that determines whether tracing instrumentation is enabled, #t by default. Affects only the way that source code is compiled, not the way that exception information is reported. The instrumentation for storing exception information slows most programs by a factor of 2 or 3.

```
(profiling-enabled) → boolean?
(profiling-enabled on?) → void?
  on? : any/c
```

Errortrace’s profiling instrumentation is #f by default. To use it, you also need to ensure that instrumenting-enabled is on.

Also, profiling only records information about the time taken on the thread that compiled the code (more precisely, the thread that instruments the code via the errortrace-compile-handler).

```
(profiling-record-enabled) → boolean?
(profiling-record-enabled on?) → void?
  on? : any/c
```

Enables/disables the recording of profiling info for the instrumented code. The default is `#t`.

Profiling information is accumulated in a hash table. If a procedure is redefined, new profiling information is accumulated for the new version of the procedure, but the old information is also preserved.

Depending of the source program, profiling usually induces a factor of 2 to 4 slowdown, in addition to any slowdown from the exception-information instrumentation.

```
(output-profile-results paths? sort-time?) → void?
  paths? : any/c
  sort-time? : any/c
```

Gets the current profile results using `get-profile-results` and displays them. It optionally shows paths information (if it is recorded), and sorts by either time or call counts.

```
(get-profile-results [thd]) → list?
  thd : thread? = (current-thread)
```

Returns a list of lists that contain all profiling information accumulated so far (for the thread *thd*):

- the number of times a procedure was called.
- the number of milliseconds consumed by the procedure's body across all calls (including the time consumed by any nested non-tail call within the procedure, but not including time consumed by a tail-call from the procedure).
- an inferred name (or `#f`) for the procedure.
- the procedure's source in the form of a syntax object (which might, in turn, provide a source location file and position).
- optionally, a list of unique call paths (i.e. stack traces) recorded if `profile-paths-enabled` is set to `#t`. Each call path is a pair of
  - a count (the number of times the path occurred), and
  - a list containing two-element lists. Each two-element list contains
    - the calling procedure's name or source expression, and
    - the calling procedure's source file or `#f`.

Collecting this information is relatively expensive.

```
(profile-paths-enabled) → boolean?
(profile-paths-enabled on?) → void?
  on? : any/c
```

Enables/disables collecting path information for profiling. The default is `#f`, but setting the parameter to `#t` immediately affects all procedures instrumented for profiling information.

```
(clear-profile-results) → void?
```

Clears accumulated profile results for the current thread.

3.2 Coverage

Errortrace can produce coverage information in two flavors: both count the number of times each expression in the source was used during execution. The first flavor uses a simple approach, where each expression is counted when executed; the second one uses the same annotations that the profiler uses, so only function bodies are counted. To see the difference between the two approaches, try this program:

```
(define (foo x) (if x 1 2))
(equal? (foo #t) 1)
```

The first approach will produce exact results, but it is more expensive; use it when you want to know how covered your code is (when the expected counts are small). The second approach produces coarser results (which, in the above case, will miss the `2` expression), but is less expensive; use it when you want to use the counts for profiling (when the expected counts are large).

```
(coverage-counts-enabled) → boolean?
(coverage-counts-enabled on?) → void?
  on? : any/c
(execute-counts-enabled) → boolean?
(execute-counts-enabled on?) → void?
  on? : any/c
```

Parameters that determine if the first (exact coverage) or second (profiler-based coverage) are enabled. Remember that setting `instrumenting-enabled` to `#f` also disables both.

```
(get-coverage) → (listof (list/c any/c natural? natural?))
                 (listof (list/c any/c natural? natural?))
```

Returns a snapshot of the state of the test coverage expressions. The first result is all of the expressions that are being monitored and the second result is all of the monitored expressions that have been covered.

```
(get-execute-counts) → (list (cons/c syntax? number?))
```

Returns a list of pairs, one for each instrumented expression. The first element of the pair is a `syntax?` object (usually containing source location information) for the original expression, and the second element of the pair is the number of times that the expression has been evaluated. This list is snapshot of the current state of the computation.

```
(annotate-covered-file  filename-path
                        [display-string]) → void?
  filename-path : path-string?
  display-string : (or/c string? #f) = #f
(annotate-executed-file  filename-path
                         [display-string]) → void?
  filename-path : path-string?
  display-string : (or/c string? #t #f) = "^.,"
```

Writes the named file to the `current-output-port`, inserting an additional line between each source line to reflect execution counts (as reported by `get-coverage-counts` or `get-execute-counts`). The optional *display-string* is used for the annotation: the first character is used for expressions that were visited 0 times, the second character for 1 time, ..., and the last character for expressions that were visited more times. It can also be `#f` for a minimal display, `"#."`, or, in the case of `annotate-executed-file`, `#t` for a maximal display, `"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"`.

```
(test-coverage-info) → hasheq?
(test-coverage-info ht) → void?
  ht : hasheq?
```

The hash-table in this parameter is used to store the profile results.

3.3 Other Errortrace Bindings

The `errortrace` module also exports:

```
(print-error-trace output-port exn) → void?
  output-port : output-port?
  exn : exn?
```

The `print-error-trace` procedure takes a port and exception and prints the Errortrace-collected debugging information contained in the exception. It is used by the exception handler installed by Errortrace.

```
(error-context-display-depth) → integer?
(error-context-display-depth d) → void?
  d : integer?
```

The error-context-display-depth parameter controls how much context Errortrace's exception handler displays. The default value is 10,000.

4 Errortrace Library

```
(require errortrace/errortrace-lib)
                  package: errortrace-lib
```

The errortrace/errortrace-lib module exports all of the exports of errortrace, plus a few more. It does not install any handlers.

The additional exports are as follows:

```
(errortrace-compile-handler stx
                            immediate-eval?)
 → compiled-expression?
  stx : any/c
  immediate-eval? : any/c
```

Compiles *stx* using the compilation handler that was active when the errortrace/errortrace-lib module was executed, but first instruments the code for Errortrace information. The code is instrumented only if

```
(list (namespace-module-registry (current-namespace))
      (namespace-base-phase (current-namespace)))
```

is the same as when the errortrace/errortrace-lib module was executed. This procedure is suitable for use as a compilation handler via current-compile.

```
(make-errortrace-compile-handler)
 → (-> any/c any/c compiled-expression)
```

Produces a compile handler that is like errortrace-compile-handler, except that the code that it produces is instrumented if the value of

```
(namespace-module-registry (current-namespace))
```

is the same as when the original thunk is invoked.

In addition, when the thunk is invoked, it uses namespace-attach-module to attach the errortrace/errortrace-key module and the '#%kernel module to the current-namespace.

```
(errortrace-error-display-handler string
                                  exn)   → void?
  string : string?
  exn : exn?
```

Displays information about the exception; this procedure is suitable for use as an error display handler.

```
(errortrace-annotate stx) → any/c
  stx : any/c
```

Macro-expands and instruments the given top-level form. If the form reads as a module whose first subexpression has the same shape as `(void '#:errortrace-dont-annotate)`, then no instrumentation is applied. See the signature element errortrace-annotate (of `stacktrace/errortrace-annotate^`) for more detail.

This annotation function is used by errortrace-compile-handler.

Changed in version 1.5 of package `errortrace-lib`: Generalized predicate for skippable modules to those using `'#:errortrace-dont-annotate`

```
(annotate-top stx phase-level) → any/c
  stx : any/c
  phase-level : exact-integer?
```

Like errortrace-annotate, but given an explicit phase level for *stx*; `(namespace-base-phase)` is typically the right value for the *phase-level* argument.

Unlike errortrace-annotate, there no special case for a module named errortrace-key or with `'#:errortrace-dont-annotate`. Also, if *stx* is a module declaration, it is not enriched with imports to explicitly load Errortrace run-time support.

5 Re-using Errortrace Stack Tracing

```
(require errortrace/stacktrace)
```
package: errortrace-lib

The errortrace collection also includes a errortrace/stacktrace library. It exports the stacktrace@ unit (plus a few generalized variants), its import signature stacktrace-imports^ (plus signatures for generalizations), and its export signature stacktrace^.

```
stacktrace@ : unit?
```

Imports stacktrace-imports^ and exports stacktrace^.

```
stacktrace/annotator@ : unit?
```

Imports stacktrace/annotator-imports^ and exports stacktrace^.

```
stacktrace/filter@ : unit?
```

Imports stacktrace-imports^ and stacktrace-filter^ and exports stacktrace^.

Added in version 1.2 of package errortrace-lib.

```
stacktrace/annotator/filter@ : unit?
```

Imports stacktrace/annotator-imports^ and stacktrace-filter^ and exports stacktrace^.

Added in version 1.2 of package errortrace-lib.

```
stacktrace/errortrace-annotate@ : unit?
```

Imports stacktrace/annotator-imports^ and exports stacktrace/errortrace-annotate^.

Added in version 1.3 of package errortrace-lib.

```
stacktrace/filter/errortrace-annotate@ : unit?
```

Imports stacktrace/imports^ and stacktrace-filter^ and exports stacktrace/errortrace-annotate^.

Added in version 1.3 of package errortrace-lib.

```
stacktrace/errortrace-annotate/key-module-name@ : unit?
```

Imports `stacktrace/annotator-imports^` and `key-module-name^` and exports `stacktrace/errortrace-annotate^`.

Added in version 1.4 of package `errortrace-lib`.

`stacktrace^` : signature

```
(annotate stx phase-level) → syntax?
  stx : syntax?
  phase-level : exact-nonnegative-integer?
(annotate-top stx phase-level) → syntax?
  stx : syntax?
  phase-level : exact-nonnegative-integer?
```

Annotate expressions with errortrace information. The `annotate-top` function should be called with a top-level expression, and `annotate` should be called with a nested expression (e.g., by `initialize-profile-point`). The *phase-level* argument indicates the phase level of the expression, typically `(namespace-base-phase)` for a top-level expression.

```
(make-st-mark stx phase-level) → (or/c #f st-mark?)
  stx : syntax?
  phase-level : exact-nonnegative-integer?
(st-mark-source st-mark) → syntax?
  st-mark : st-mark?
(st-mark-bindings st-mark) → list?
  st-mark : st-mark?
```

The `st-mark-source` and `st-mark-bindings` functions extract information from a particular kind of value. The value must be created by `make-st-mark` (the shape of the value is guaranteed to be writable and not to be `#f`, but otherwise unspecified). The `make-st-mark` function returns `#f` when there is no source location information in the syntax object. The `st-mark-source` extracts the value originally provided to the expression-maker, and `st-mark-bindings` returns local binding information (if available) as a list of two element (syntax? any/c) lists. The `st-mark-bindings` function is currently hardwired to return `null`.

`stacktrace/errortrace-annotate^` : signature

Extends the support for annotation to better support use in the `current-compile` handler and to add explicit requires for errortrace's runtime support.

Added in version 1.3 of package `errortrace-lib`.

```
(errortrace-annotate  stx
                      [in-compile-handler?]) → syntax?
  stx : syntax?
  in-compile-handler? : boolean? = #t
```

Adds the property `'errortrace:annotate` to everywhere inside *stx*, and expands it. If *stx* is a module (but not named `errortrace-key` module nor a cross-phase persistent module nor a module whose first subexpression is `(void '#:errortrace-dont-annotate)`), calls `annotate-top` with the expanded code and inserts appropriate requires to the `errortrace-key` module.

If *in-compile-handler?* is true, also calls `namespace-require` to load `errortrace-key`.

See also `original-stx` and `expanded-stx`.

```
stacktrace-imports^ : signature
```

```
(with-mark source-stx dest-stx phase) → any/c
  source-stx : any/c
  dest-stx : any/c
  phase : exact-nonnegative-integer?
```

Called by `annotate` and `annotate-top` to wrap expressions with `with-continuation-mark`. The first argument is the source expression, the second argument is the expression to be wrapped, and the last is the phase level of the expression.

```
(test-coverage-enabled) → boolean?
(test-coverage-enabled on?) → void?
  on? : any/c
```

Determines if the test coverage annotation is inserted into the code. This parameter controls how compilation happens—it does not affect the dynamic behavior of the already compiled code. If the parameter is set, code generated by `test-covered` are inserted into the code (and `initialize-test-coverage-point` is called during compilation). If not, no calls to `test-covered` code are inserted.

```
(test-covered stx) → (or/c syntax? (-> void?) #f)
  stx : any/c
```

This is called during compilation of the program with an expression for each point in the program that was passed to `initialize-test-coverage-point`.

If the result is `#f`, this program point is not instrumented. If the result is syntax, it is inserted into the code, and if it is a thunk, the thunk is inserted into the code in an application (using the thunk directly, as a 3D value). In either case, the syntax or the thunk should register that the relevant point was covered.

Note: using a thunk tends to be slow. Current uses in the Racket code will create a mutable pair in `initialize-test-coverage-point`, and `test-covered` returns syntax that will set its mcar. (This makes the resulting overhead about 3 times smaller.)

```
(initialize-test-coverage-point stx) → void?
  stx : any/c
```

During compilation of the program, this function is called with each subexpression of the program. The argument is the syntax of this program point, which is usually used as a key to identify this program point.

```
profile-key : any/c
```

Only used for profiling paths.

```
(profiling-enabled) → boolean?
(profiling-enabled on?) → void?
  on? : any/c
```

Determines if profiling information is currently collected (affects the behavior of compiling the code—does not affect running code). If this always returns `#f`, the other profiling functions are never called.

```
(initialize-profile-point key name stx) → void?
  key : any/c
  name : (or/c syntax? false/c)
  stx : any/c
```

Called as the program is compiled for each profiling point that might be encountered during the program's execution. The first argument is a key identifying this code. The second argument is the inferred name at this point and the final argument is the syntax of this expression.

```
(register-profile-start key) → (or/c number? false/c)
  key : any/c
```

Called when some profiled code is about to be executed. If the result is a number, it is expected to be the current number of milliseconds. *key* is unique to this fragment of code—it is the same key passed to `initialize-profile-point` for this code fragment.

```
(register-profile-done key start) → void?
  key : any/c
  start : (or/c number? false/c)
```

This function is called when some profiled code is finished executing.

Note that `register-profile-start` and `register-profile-done` can be called in a nested manner; in this case, the result of `register-profile-start` should be `#f`.

```
stacktrace/annotator-imports^ : signature
```

Like `stacktrace-imports^`, but providing more control over the annotation function for test cases. The only difference between the two signatures is `test-coverage-enabled`, `initialize-test-coverage-point`, and `test-covered` are replaced by `test-coverage-point`.

```
(test-coverage-point body expr phase) → syntax?
  body : syntax?
  expr : syntax?
  phase : exact-integer?
```

Initializes the test coverage point for *expr* and returns the syntax that will cover it. *body* is the body that should be run after the coverage for *expr* has been recorded as covered. *body* and *expr* may not be the same. For example *expr* may not have appeared in an expression position. *phase* is the phase level at which *expr* appeared.

```
(with-mark source-stx dest-stx phase) → any/c
  source-stx : any/c
  dest-stx : any/c
  phase : exact-nonnegative-integer?
```

Same as in `stacktrace-imports^`.

```
profile-key : any/c
```

Same as in `stacktrace-imports^`.

```
(profiling-enabled) → boolean?
(profiling-enabled on?) → void?
  on? : any/c
```

Same as in `stacktrace-imports^`.

```
(initialize-profile-point key name stx) → void?
  key : any/c
  name : (or/c syntax? false/c)
  stx : any/c
```

Same as in stacktrace-imports^.

```
(register-profile-start key) → (or/c number? false/c)
  key : any/c
```

Same as in stacktrace-imports^.

```
(register-profile-done key start) → void?
  key : any/c
  start : (or/c number? false/c)
```

Same as stacktrace-imports^.

key-module-name^ : signature

key-module-name : module-path?

References to this module are inserted into each compile, such that references to its exported identifiers can be inserted by with-mark (as in stacktrace-imports^ and stacktrace/annotator-imports^).

Added in version 1.4 of package errortrace-lib.

stacktrace-filter^ : signature

The function defined for stacktrace-filter^ provides additional control over the expressions that are instrumented for debugging an profiling.

Added in version 1.2 of package errortrace-lib.

```
(should-annotate? stx phase) → any/c
  stx : syntax?
  phase : exact-nonnegative-integer?
```

Determines whether the annotate function from stacktrace/filter@ or stacktrace/annotator/filter@ adds debugging and/or test-coverage annotation to the immediate expression represented by *stx*. Annotation is added only when should-annotate? returns a true value, but subexpressions of *stx* will be checked and potentially annotated independent of the result for *stx*.

The default filter function used by stacktrace@ and stacktrace/annotator@ annotates an expression if it has a source location according to syntax-source.

When used via the errortrace meta-language or when requireing the errortrace module or when using errortrace via the "Debugging" option in DrRacket, the should-annotate? function checks to make sure that the syntax object has a source location and has the syntax property 'errortrace:annotate.

```
(original-stx) → (or/c syntax? #f)
(original-stx stx) → void?
  stx : (or/c syntax? #f)
 = #f
```

A parameter set by errortrace-annotate when it is called (to be used by with-mark, etc) to its input.

```
(expanded-stx) → (or/c syntax? #f)
(expanded-stx stx) → void?
  stx : (or/c syntax? #f)
 = #f
```

A parameter set by errortrace-annotate when it is called (to be used by with-mark, test-covered, etc) to the expanded version of its input.

6 Programmatically Extracting Stack-trace Information

```
(require errortrace/marks-to-context)
```

```
(errortrace-continuation-mark-set->context)
 → (or/c #f (-> continuation-mark-set? (listof srcloc?)))
(errortrace-continuation-mark-set->context proc) → void?
  proc : (or/c #f (-> continuation-mark-set? (listof srcloc?)))
 = (λ (x) #f)
```

Extracts the list of source locations recorded by `with-mark` from the given continuation marks, if it is a procedure. This parameter is expected to be set by an instantiation of the units documented above, e.g., by requiring the `errortrace` library or by enabling Debugging in DrRacket's language configuration dialog.

Added in version 1.6.

7 Errortrace Key

```
(require errortrace/errortrace-key)
                    package: errortrace-lib
```

This module depends only on `'#%kernel`.

```
errortrace-key : symbol?
```

A key used by errortrace via `with-continuation-mark` to record stack information.