TRY MANUAL

Contents

[in package TRY]

1 Links and Systems

Here is the official repository and the HTML documentation for the latest version.

- **[system]** "try"
  - *Version:* 0.0.8
  - *Description:* Try is an extensible test framework with equal support for interactive and non-interactive workflows.
  - *Long Description:* Try stays as close to normal Lisp evaluation rules as possible. Tests are functions that record the checks they perform as events. These events provide the means of customization of what to debug, print or rerun. There is a single fundamental check, the extensible `is` macro. Everything else is built on top.
  - *Licence:* MIT, see COPYING.
  - *Author:* Gábor Melis
  - *Mailto:* mega@retes.hu
  - *Homepage:* http://github.com/melisgl/try
  - *Bug tracker:* https://github.com/melisgl/try/issues
  - *Source control:* GIT
  - *Depends on:* alexandria, cl-ppcre, closer-mop, ieee-floats, mgl-pax, trivial-gray-streams, uiop

2 Tutorial

Try is a library for unit testing with equal support for interactive and non-interactive workflows. Tests are functions, and almost everything else is a condition, whose types feature prominently

in parameterization.

Try is what we get if we make tests functions and build a test framework on top of the condition system as Stefil did but also address the issue of rerunning and replaying, make the `is` check more capable, use the types of the condition hierarchy to parametrize what to debug, print, rerun, and finally document the whole thing.

Looking for Truth The `is` Macro is a replacement for `cl:assert` that can capture values of subforms to provide context to failures:

```
(is (= (1+ 5) 0))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS (= #1=(1+ 5) 0))
..   where
..     #1# = 6
```

This is a PAX transcript, output is prefixed with `".. "`. Readable and unreadable return values are prefixed with `"=> "` and `"==> "`, respectively.

Note the `#n#` syntax due to `*print-circle*`.

Checking Multiple Values `is` automatically captures values of arguments to functions like `1+` in the above example. Values of other interesting subforms can be explicitly captured. `is` supports capturing multiple values and can be taught how to deal with macros. The combination of these features allows `match-values` to be implementable as a tiny extension:

```
(is (match-values (values (1+ 5) "sdf")
      (= * 0)
      (string= * "sdf")))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS
..      (MATCH-VALUES #1=(VALUES (1+ 5) #2="sdf")
..        (= * 0)
..        (STRING= * "sdf")))
..   where
..     #1# == 6
..            #2#
```

In the body of `match-values`, `*` is bound to successive return values of some form, here `(values (1+ 5) "sdf")`. `match-values` comes with an automatic rewrite rule that captures the values of this form, which are printed above as `#1# == 6 #2#`. `is` is flexible enough that all other checks (`signals`, `signals-not`, `invokes-debugger`, `invokes-debugger-not`, `fails`, and `in-time`) are built on top of it.

Writing Tests Beyond `is` (a fancy `assert`), Try provides tests, which are Lisp functions that record their execution in `trial` objects. Let's define a test and run it:

```
(deftest should-work ()
  (is t))

(should-work)
.. SHOULD-WORK            ; TRIAL-START
..   · (IS T)             ; EXPECTED-RESULT-SUCCESS
.. · SHOULD-WORK ·1       ; EXPECTED-VERDICT-SUCCESS
..
==> #<TRIAL (SHOULD-WORK) EXPECTED-SUCCESS 0.000s ·1>
```

Try is driven by conditions, and the comments to the right show the type of the condition printed on that line. The · character marks successes.

We could have run our test with `(try 'should-work)` as well, which does pretty much the same thing except that it defaults to never entering the debugger, whereas calling a test function directly enters the debugger on events whose type matches the type in the variable `*debug*`.

```
(try 'should-work)
.. SHOULD-WORK
..   · (IS T)
.. · SHOULD-WORK ·1
..
==> #<TRIAL (SHOULD-WORK) EXPECTED-SUCCESS 0.000s ·1>
```

Test Suites Test suites are just tests that call other tests.

```
(deftest my-suite ()
  (should-work)
  (is (= (foo) 5)))

(defun foo ()
  4)

(try 'my-suite)
.. MY-SUITE                ; TRIAL-START
..   SHOULD-WORK           ; TRIAL-START
..     · (IS T)            ; EXPECTED-RESULT-SUCCESS
..   · SHOULD-WORK ·1      ; EXPECTED-VERDICT-SUCCESS
..   ☒ (IS (= #1=(FOO) 5))  ; UNEXPECTED-RESULT-FAILURE
..     where
..       #1# = 4
.. ☒ MY-SUITE ☒1 ·1        ; UNEXPECTED-VERDICT-FAILURE
..
==> #<TRIAL (MY-SUITE) UNEXPECTED-FAILURE 0.000s ☒1 ·1>
```

☒ marks `unexpected-failure`s. Note how the failure of `(is (= (foo) 5))` caused `my-suite` to fail as well. Finally, the `☒1` and the `·1` in the `trial`'s printed representation are the event counts.

Filtering Output To focus on the important bits, we can print only the `unexpected` events:

```
(try 'my-suite :print 'unexpected)
.. MY-SUITE
..   ☒ (IS (= #1=(FOO) 5))
..     where
..       #1# = 4
.. ☒ MY-SUITE ☒1 ·1
..
==> #<TRIAL (MY-SUITE) UNEXPECTED-FAILURE 0.000s ☒1 ·1>
```

Note that `should-work` is still run, and its check's success is counted as evidenced by `·1`. The above effect can also be achieved without running the tests again with `replay-events`.

Debugging Let's figure out what went wrong:

```
(my-suite)

;;; Here the debugger is invoked:
UNEXPECTED-FAILURE in check:
  (IS (= #1=(FOO) 5))
where
  #1# = 4
Restarts:
 0: [RECORD-EVENT] Record the event and continue.
 1: [FORCE-EXPECTED-SUCCESS] Change outcome to TRY:EXPECTED-RESULT-SUCCESS.
 2: [FORCE-UNEXPECTED-SUCCESS] Change outcome to TRY:UNEXPECTED-RESULT-SUCCESS.
 3: [FORCE-EXPECTED-FAILURE] Change outcome to TRY:EXPECTED-RESULT-FAILURE.
 4: [ABORT-CHECK] Change outcome to TRY:RESULT-ABORT*.
 5: [SKIP-CHECK] Change outcome to TRY:RESULT-SKIP.
 6: [RETRY-CHECK] Retry check.
 7: [ABORT-TRIAL] Record the event and abort trial TRY::MY-SUITE.
 8: [SKIP-TRIAL] Record the event and skip trial TRY::MY-SUITE.
 9: [RETRY-TRIAL] Record the event and retry trial TRY::MY-SUITE.
 10: [SET-TRY-DEBUG] Supply a new value for :DEBUG of TRY:TRY.
 11: [RETRY] Retry SLIME interactive evaluation request.
```

In the SLIME debugger, we press `v` on the frame of the call to `my-suite` to navigate to its definition, realize what the problem is and fix `foo`:

```
(defun foo ()
  5)
```

Now, we select the `retry-trial` restart, and on the retry `my-suite` passes. The full output is:

```
MY-SUITE
  SHOULD-WORK
    · (IS T)
  · SHOULD-WORK ·1
WARNING: redefining TRY::FOO in DEFUN
  ☒ (IS (= #1=(FOO) 5))
    where
      #1# = 4
MY-SUITE retry #1
```

```
  SHOULD-WORK
    · (IS T)
  · SHOULD-WORK ·1
  · (IS (= (FOO) 5))
· MY-SUITE ·2
```

Rerunning Stuff Instead of working interactively, one can fix the failing test and rerun it. Now, let's fix `my-suite` and rerun it:

```
(deftest my-suite ()
  (should-work)
  (is nil))

(try 'my-suite)
.. MY-SUITE
..   SHOULD-WORK
..     · (IS T)
..   · SHOULD-WORK ·1
..   ☒ (IS NIL)
.. ☒ MY-SUITE ☒1 ·1
..
==> #<TRIAL (MY-SUITE) UNEXPECTED-FAILURE 0.000s ☒1 ·1>

(deftest my-suite ()
  (should-work)
  (is t))

(try !)
.. MY-SUITE
..   - SHOULD-WORK
..   · (IS T)
.. · MY-SUITE ·1
..
==> #<TRIAL (MY-SUITE) EXPECTED-SUCCESS 0.004s ·1>
```

Here, `!` refers to the most recent `trial` returned by `try`. When a trial is passed to `try` or is `funcall`ed, trials in it that match the type in `try`'s `rerun` argument are rerun (here, `unexpected` by default). `should-work` and its check are `expected-success`es, hence they don't match `unexpected` and are not rerun.

Conditional Execution Conditional execution can be achieved simply by testing the `trial` object returned by Tests.

```
(deftest my-suite ()
  (when (passedp (should-work))
    (is t :msg "a test that depends on SHOULD-WORK")
    (when (is nil)
      (is nil :msg "never run"))))
```

Skipping Sometimes, we do not know up front that a test should not be executed. Calling `skip-trial` unwinds from the `current-trial` and marks it skipped.

```
(deftest my-suite ()
  (is t)
  (skip-trial)
  (is nil))

(my-suite)
==> #<TRIAL (MY-SUITE) SKIP 0.000s ·1>
```

In the above, `(is t)` was executed, but `(is nil)` was not.

```
(deftest known-broken ()
  (with-failure-expected (t)
    (is nil)))

(known-broken)
.. KNOWN-BROKEN
..    × (IS NIL)
.. · KNOWN-BROKEN ×1
..
==> #<TRIAL (KNOWN-BROKEN) EXPECTED-SUCCESS 0.000s ×1>
```

Expecting Outcomes × marks `expected-failure`s. `(with-skip (t) ...)` makes all check successes and failures `expected`, which are counted in their own `*categories*` by default but don't make the enclosing tests fail. Also see `with-expected-outcome`.

Running Tests on Definition With `*run-deftest-when*`, tests can be run in various `eval-when` situations. To run tests on evaluation, as in SLIME `C-M-x`, `slime-eval-defun`:

```
(setq *run-deftest-when* :execute)

(deftest some-test ()
  (is t))
.. SOME-TEST
..    · (IS T)
.. · SOME-TEST ·1
..
=> SOME-TEST

(setq *run-deftest-when* nil)
```

Fixtures There is no direct support for fixtures in Try because they are not needed with the ability of Rerunning Trials in context.

If one insists, macros like the following are easy to write.

```
(defvar *server* nil)
```

```
(defmacro with-xxx (&body body)
  `(flet ((,with-xxx-body ()
            ,@body))
     (if *server*
         (with-xxx-body)
         (with-server (make-expensive-server)
           (with-xxx-body)))))
```

Packages The suggested way of writing tests is to call test functions explicitly:

```
(defpackage :some-test-package
  (:use #:common-lisp #:try))
(in-package :some-test-package)

(deftest test-all ()
  (test-this)
  (test-that))

(deftest test-this ()
  (test-this/more))

(deftest test-this/more ()
  (is t))

(deftest test-that ()
  (is t))

(deftest not-called ()
  (is t))

(defun test ()
  (warn-on-tests-not-run ((find-package :some-test-package))
    (try 'test-all)))

(test)
.. TEST-ALL
..   TEST-THIS
..     TEST-THIS/MORE
..       · (IS T)
..     · TEST-THIS/MORE ·1
..   · TEST-THIS ·1
..   TEST-THAT
..     · (IS T)
..   · TEST-THAT ·1
.. · TEST-ALL ·2
.. WARNING: Test NOT-CALLED not run.
==> #<TRIAL (TEST-ALL) EXPECTED-SUCCESS 0.012s ·2>
```

Note how the `test` function uses `warn-on-tests-not-run` to catch any tests defined in `some-test-package` that were not run. Tests can be deleted by `fmakunbound`, `unintern`, or by redefining the function with `defun`. Tests defined in a given package can be listed with `list-`

`package-tests`.

This style allows higher level tests to establish the dynamic environment necessary for lower level tests.

3 Emacs Integration

The Elisp `mgl-try` interactive command runs `try` with some testable and displays its output in a Try buffer, which has major mode `lisp-mode` and minor modes `outline-mode` and `mgl-try-mode`. It is assumed that the Lisp is running under Slime.

Use `mgl-try-rerun` and `mgl-try-rerun-all` to rerun trials. They are especially convenient to rerun `try:!`, when deciding to inspect the results in a Try buffer for a trial that may not have been run via Emacs.

In an Emacs Try buffer, the following key bindings are available.

- Movement:
  - Cursor keys move freely.
  - `C-p` and `C-n` move between events.
  - `p` and `n` to move between `unexpected` events.
  - `P` and `N` move between events which are not `expected-successes`.
  - `<tab>` cycles visibility of the current heading's body.
  - `U` moves to the parent heading.
  - `q` is bound to `quit-window`.
- Calling tests:
  - `t` runs a test (defaults to the name of the innermost global test function that contains the current line) in the context associated with the Emacs buffer, which is similar to but distinct from `*rerun-context*`. With a prefix arg, the test is an Implicit `try` with no arguments. This is suitable for interactive debugging under the default settings.
  - `r` reruns the most recent trial conducted by Emacs (this is distinct from `try:!`). With a prefix argument, the test is called implicitly.
  - `R` is like `r`, but `*try-rerun*` and `try:*rerun*` are set to `t`, to ensure that all tests are rerun. With a prefix argument, the test is called implicitly.
- Visiting source locations:
  - `v` visits the source location of the enclosing global test function (see `t`).
  - `M-.` visits a test function as usual.

In general, since the major mode is `lisp-mode`, the usual key bindings are available.

3.1 Emacs Setup

Load `src/mgl-try.el` in Emacs.

If you installed Try with Quicklisp, the location of `mgl-try.el` may change with updates, and you may want to copy the current version of `mgl-try.el` to a stable location:

```
(try:install-try-elisp "~/quicklisp/")
```

Then, assuming the Elisp file is in the quicklisp directory, add something like this to your `.emacs`:

```
(load "~/quicklisp/mgl-try.el")
```

For easy access to the functionality of the keys `t`, `r` and `r` described in Emacs Integration, you may want to give them a global binding:

```
(global-set-key (kbd "s-t t") 'mgl-try)
(global-set-key (kbd "s-t r") 'mgl-try-rerun)
(global-set-key (kbd "s-t R") 'mgl-try-rerun-all)
```

The same with `use-package`:

```
(use-package mgl-try :load-path "~/quicklisp/"
  :after slime
  :demand t
  :bind (("s-t t" . mgl-try)
         ("s-t r" . mgl-try-rerun)
         ("s-t R" . mgl-try-rerun-all)))
```

- **[function]** **install-try-elisp** *target-dir*

  Install `mgl-try.el` distributed with this package in `target-dir`.

4 Events

Try is built around events implemented as `condition`s. Matching the types of events to `*debug*`, `*count*`, `*collect*`, `*rerun*`, `*print*`, and `*describe*` is what gives Try its flexibility.

4.1 Middle Layer of Events

The event hierarchy is fairly involved, so let's start with the middle layer because it is the smallest. The condition `event` has 4 disjoint subclasses:

- `trial-start`, starting a `trial` (by executing a test),
- `verdict`, the `outcome` of a `trial`,
- `result`, the `outcome` of a check, and
- `error*`, an unexpected `cl:error`(`0` `1`) or unadorned non-local exit.

```
(let (;; We don't want to debug nor print a backtrace for the error below.
      (*debug* nil)
```

```
      (*describe* nil))
  ;; signals TRIAL-START / VERDICT-ABORT* on entry / exit
  (with-test (demo)
    ;; signals EXPECTED-RESULT-SUCCESS
    (is t)
    ;; signals UNHANDLED-ERROR with a nested CL:ERROR
    (error "xxx")))
.. DEMO                          ; TRIAL-START
..   · (IS T)                   ; EXPECTED-RESULT-SUCCESS (·)
..   ⊟ "xxx" (SIMPLE-ERROR)     ; UNHANDLED-ERROR (⊟)
.. ⊟ DEMO ⊟1 ·1                 ; VERDICT-ABORT* (⊟)
..
==> #<TRIAL (WITH-TEST (DEMO)) ABORT* 0.004s ⊟1 ·1>
```

4.2 Concrete Events

The non-abstract condition classes of events that are actually signalled are called concrete.

Checks' `result`s and Trials' `verdict`s have six concrete subclasses each:

- `expected-result-success`, `unexpected-result-success`, `expected-result-failure`, `unexpected-result-failure`, `result-skip`, `result-abort*`
- `expected-verdict-success`, `unexpected-verdict-success`, `expected-verdict-failure`, `unexpected-verdict-failure`, `verdict-skip`, `verdict-abort*`

Breaking the symmetry between Checks and Trials, `trial-start` is a concrete event class, that marks the start of a `trial`.

`error*` is an abstract class with two concrete subclasses:

- `unhandled-error`, signalled when a `cl:error`(`0` `1`) reaches the handler set up by `deftest` or `with-test`, or when the debugger is invoked.
- `nlx`, signalled when no error was detected by the handler, but the trial finishes with a non-local exit.

These are the 15 concrete event classes.

- **[function]** **concrete-events-of-type** *type*

  The hierarchy of Events is hairy. Sometimes it's handy to list the Concrete Events that match a given type. We use this below in the documentation.

4.3 Event Glue

These condition classes group various bits of the Concrete Events and the Middle Layer of Events for ease of reference.

Concrete event classes except `trial-start` and `nlx` are subclasses of the hyphen-separated words constituting their name. For example, `unexpected-result-failure` inherits from `unexpected`, `result`, and `failure`, so it matches types such as `unexpected` or `(and unexpected result)`.

- **[condition]** **event**

  Common abstract superclass of all events in Try.

- **[condition]** **act** *event*

  events that produce evidence or determine the course of a `trial` are acts. All events are acts except `trial-start`.

```
(concrete-events-of-type '(not act))
=> (TRIAL-START)
```

`expected` and `unexpected` partition `act`.

- **[condition]** **expected** *act*

  Concrete condition classes with `expected` in their name are subclasses of `expected`. `skip` is also a subclass of `expected`.

```
(concrete-events-of-type 'expected)
=> (EXPECTED-RESULT-SUCCESS EXPECTED-RESULT-FAILURE RESULT-SKIP
    EXPECTED-VERDICT-SUCCESS EXPECTED-VERDICT-FAILURE VERDICT-SKIP)
```

- **[condition]** **unexpected** *act*

  Concrete condition classes with `unexpected` in their name are subclasses of `unexpected`. `abort*` is also a subclass of `unexpected`.

```
(concrete-events-of-type 'unexpected)
=> (UNEXPECTED-RESULT-SUCCESS UNEXPECTED-RESULT-FAILURE RESULT-ABORT*
    UNEXPECTED-VERDICT-SUCCESS UNEXPECTED-VERDICT-FAILURE
    VERDICT-ABORT* UNHANDLED-ERROR NLX)
```

`success`, `failure` and `dismissal` partition `act`.

- **[condition]** **success** *act*

  See Checks and Trial Verdicts for how `success` or `failure` is decided.

```
(concrete-events-of-type 'success)
=> (EXPECTED-RESULT-SUCCESS UNEXPECTED-RESULT-SUCCESS
    EXPECTED-VERDICT-SUCCESS UNEXPECTED-VERDICT-SUCCESS)
```

- **[condition]** **failure** *act*

  See `success`.

```
(concrete-events-of-type 'failure)
=> (EXPECTED-RESULT-FAILURE UNEXPECTED-RESULT-FAILURE
    EXPECTED-VERDICT-FAILURE UNEXPECTED-VERDICT-FAILURE)
```

- **[condition]** **dismissal** *act*

  The third possibility after `success` and `failure`. Either `skip` or `abort*`.

```
(concrete-events-of-type 'dismissal)
=> (RESULT-SKIP RESULT-ABORT* VERDICT-SKIP VERDICT-ABORT*
    UNHANDLED-ERROR NLX)
```

`abort*` and `skip` partition `dismissal`.

- **[condition]** **abort*** *unexpected dismissal*

```
(concrete-events-of-type 'abort*)
=> (RESULT-ABORT* VERDICT-ABORT* UNHANDLED-ERROR NLX)
```

- **[condition]** **skip** *expected dismissal*

```
(concrete-events-of-type 'skip)
=> (RESULT-SKIP VERDICT-SKIP)
```

- **[condition]** **leaf** *act*

  Events that do not mark a `trial`'s start (`trial-start`) or end (`verdict`) are `leaf` events. These are the leaves of the tree of nested trials delineated by their `trial-start` and `verdict` events.

```
(concrete-events-of-type 'leaf)
=> (EXPECTED-RESULT-SUCCESS UNEXPECTED-RESULT-SUCCESS
    EXPECTED-RESULT-FAILURE UNEXPECTED-RESULT-FAILURE RESULT-SKIP
    RESULT-ABORT* UNHANDLED-ERROR NLX)
```

  `leaf` `event`s are `result`s of Checks and also `error*`s.

```
(equal (concrete-events-of-type 'leaf)
       (concrete-events-of-type '(or result error*)))
=> T
```

  Equivalently, `leaf` is the complement of `trial-event`.

```
(equal (concrete-events-of-type 'leaf)
       (concrete-events-of-type '(not trial-event)))
=> T
```

The following types are shorthands.

- **[type]** **expected-success**

  A shorthand for `(and expected success)`.

- **[type]** **unexpected-success**

  A shorthand for `(and unexpected success)`.

- **[type]** **expected-failure**

  A shorthand for `(and expected failure)`.

- **[type]** **unexpected-failure**

A shorthand for `(and unexpected failure)`.

`pass` and `fail` partition `act`.

- **[type] pass**

  An `outcome` that's not an `abort*` or an `unexpected-failure`. `pass` is equivalent to `(not fail)`. `pass`es are signalled with `signal`.

```
(concrete-events-of-type 'pass)
=> (EXPECTED-RESULT-SUCCESS UNEXPECTED-RESULT-SUCCESS
    EXPECTED-RESULT-FAILURE RESULT-SKIP EXPECTED-VERDICT-SUCCESS
    UNEXPECTED-VERDICT-SUCCESS EXPECTED-VERDICT-FAILURE VERDICT-SKIP)
```

- **[type] fail**

  An `abort*` or an `unexpected-failure`. `fail` conditions are signalled with `error`. See `pass`.

```
(concrete-events-of-type 'fail)
=> (UNEXPECTED-RESULT-FAILURE RESULT-ABORT* UNEXPECTED-VERDICT-FAILURE
    VERDICT-ABORT* UNHANDLED-ERROR NLX)
```

4.4 Printing Events

- **[variable] *event-print-bindings*** *((*print-circle* t) (sb-ext:*print-circle-not-shared* nil))*

  Try var. `event`s are conditions signalled in code that may change printer variables such as `*print-circle*`, `*print-length*`, etc. To control how events are printed, the list of variable bindings in `*event-print-bindings*` is established whenever an `event` is printed as if with:

```
(progv (mapcar #'first *event-print-bindings*)
       (mapcar #'second *event-print-bindings*)
  ...)
```

  The default value ensures that shared structure is recognized (see Captures). If the `#n#` syntax feels cumbersome, then change this variable.

4.5 Event Restarts

Only `record-event` is applicable to all `event`s. See Check Restarts, Trial Restarts for more.

- **[function] record-event** *&optional condition*

  This restart is always the first restart available when an `event` is signalled running under `try` (i.e. there is a `current-trial`). `try` always invokes `record-event` when handling events.

4.6 Outcomes

- **[condition] outcome** *act*

An `outcome` is the resolution of either a `trial` or a check, corresponding to subclasses `verdict` and `result`.

```
(concrete-events-of-type '(not outcome))
=> (TRIAL-START UNHANDLED-ERROR NLX)
```

- **[macro] with-expected-outcome** *(expected-type) &body body*

  When an `outcome` is to be signalled, `expected-type` determines whether it's going to be `expected`. The concrete `outcome` classes are `{expected,unexpected}-{result,verdict}-{success,failure}` (see Events), of which `result` or `verdict` and `success` or `failure` are already known. If a `result failure` is to be signalled, then the moral equivalent of `(subtypep '(and result failure) expected-type)` is evaluated and depending on whether it's true, `expected-result-failure` or `unexpected-result-failure` is signalled.

  By default, `success` is expected. The following example shows how to expect both `success` and `failure` for `result`s, while requiring `verdict`s to succeed:

```
(let ((*debug* nil))
  (with-expected-outcome ('(or result (and verdict success)))
    (with-test (t1)
      (is nil))))
.. T1
..   × (IS NIL)
.. · T1 ×1
..
==> #<TRIAL (WITH-TEST (T1)) EXPECTED-SUCCESS 0.000s ×1>
```

  This is equivalent to `(with-failure-expected () ...)`. To make result failures expected but result successes unexpected:

```
(let ((*debug* nil))
  (with-expected-outcome ('(or (and result failure) (and verdict success)))
    (with-test (t1)
      (is t)
      (is nil))))
.. T1
..   ⊡ (IS T)
..   × (IS NIL)
.. · T1 ⊡1 ×1
..
==> #<TRIAL (WITH-TEST (T1)) EXPECTED-SUCCESS 0.000s ⊡1 ×1>
```

  This is equivalent to `(with-failure-expected ('failure) ...)`. The final example leaves result failures unexpected but makes both verdict successes and failures expected:

```
(let ((*debug* nil))
  (with-expected-outcome ('(or (and result success) verdict))
    (with-test (t1)
      (is nil))))
.. T1
..   ⊠ (IS NIL)
```

```
.. × T1 ☒1
..
==> #<TRIAL (WITH-TEST (T1)) EXPECTED-FAILURE 0.004s ☒1>
```

- **[macro]** **with-failure-expected** *(&optional (result-expected-type t) (verdict-expected-type 'success)) &body body*

  A convenience macro on top of `with-expected-outcome`, `with-failure-expected` expects `verdicts` to have `verdict-expected-type` and `results` to have `result-expected-type`. A simple `(with-failure-expected () ...)` makes all `result` `successes` and `failures` `expected`. `(with-failure-expected ('failure) ..)` expects `failures` only, and any `successes` will be `unexpected`.

- **[macro]** **with-skip** *(&optional (skip t)) &body body*

  `with-skip` skips checks and trials. It forces an immediate `skip-trial` whenever a trial is started (which turns into a `verdict-skip`) and makes checks (without intervening trials, of course) evaluate normally but signal `result-skip`. `skip` being `nil` cancels the effect of any enclosing `with-skip` with `skip` true.

4.6.1 Outcome Restarts

- **[function]** **force-expected-success** *&optional outcome*

  Handle the `outcome` being signalled, and signal an `expected-result-success` or `expected-verdict-success` for when the `outcome` is a `result` or a `verdict`, respectively.

- **[function]** **force-unexpected-success** *&optional outcome*

  Handle the `outcome` being signalled, and signal an `unexpected-result-success` or `unexpected-verdict-success` for when the `outcome` is a `result` or a `verdict`, respectively.

- **[function]** **force-expected-failure** *&optional outcome*

  Handle the `outcome` being signalled, and signal an `expected-result-failure` or `expected-verdict-failure` for when the `outcome` is a `result` or a `verdict`, respectively.

- **[function]** **force-unexpected-failure** *&optional outcome*

  Handle the `outcome` being signalled, and signal an `unexpected-result-failure` or `unexpected-verdict-failure` for when the `outcome` is a `result` or a `verdict`, respectively.

4.6.2 Checks

Checks are like `cl:assert`s, they check whether some condition holds and signal an `outcome`. The outcome signalled for checks is a subclass of `result`.

Take, for example, `(is (= x 5))`. Depending on whether `x` is indeed 5, some kind of `result` `success` or `failure` will be signalled. `with-expected-outcome` determines whether it's `expected` or `unexpected`, and we have one of `expected-result-success`, `unexpected-result-success`, `expected-result-failure`, `unexpected-result-failure` to signal. Furthermore, if `with-skip` is in effect, then `result-skip` is signalled.

The result is signalled with the function `signal` if it is a `pass`, else it's signalled with `error`. This distinction matters only if the event is not handled, which is never the case in a `trial`. However, standalone checks – those not enclosed by a trial – invoke the debugger on `result`s which are not of type `pass`.

The signalled `result` is not final until `record-event` is invoked on it, and it can be changed with the Outcome Restarts and the Check Restarts.

- **[condition]** **result** *leaf outcome*
- **[condition]** **expected-result-success** *expected result success*
- **[condition]** **unexpected-result-success** *unexpected result success*
- **[condition]** **expected-result-failure** *expected result failure*
- **[condition]** **unexpected-result-failure** *unexpected result failure*
- **[condition]** **result-skip** *result skip*
- **[condition]** **result-abort*** *result abort* dismissal*

Check Restarts

- **[function]** **abort-check** *&optional condition*

  Change the `outcome` of the check being signalled to `result-abort*`. `result-abort*`, being a `fail`, will cause the check to return `nil` if `record-event` is invoked on it.

- **[function]** **skip-check** *&optional condition*

  Change the `outcome` of the check being signalled to `result-skip`. `result-skip`, being a `pass`, will cause the check to return `t` if `continue(0 1)` or `record-event` is invoked on it.

- **[function]** **retry-check** *&optional condition*

  Initiate a non-local exit to go reevaluate the forms wrapped by the check without signalling an `outcome`.

4.6.3 Trials

- **[class]** **trial** *sb-mop:funcallable-standard-object*

  Trials are records of calls to tests (see Counting Events, Collecting Events). Their behaviour as funcallable instances is explained in Rerunning Trials.

There are three ways to acquire a `trial` object: by calling `current-trial`, through the lexical binding of the symbol that names the test, or through the return value of a test:

```
(deftest xxx ()
  (prin1 xxx))

(xxx)
.. #<TRIAL (XXX) RUNNING>
==> #<TRIAL (XXX) EXPECTED-SUCCESS 0.000s>
```

`with-trial` can also provide access to its `trial`:

```
(with-test (t0)
  (prin1 t0))
.. #<TRIAL (WITH-TEST (T0)) RUNNING>
==> #<TRIAL (WITH-TEST (T0)) EXPECTED-SUCCESS 0.000s>
```

`trial`s are not to be instantiated by client code.

- **[function]** **current-trial**

  `trial`s, like the calls to tests they stand for, nest. `current-trial` returns the innermost trial. If there is no currently running test, then an error is signalled. The returned trial is `runningp`.

Trial Events

- **[condition]** **trial-event** *event*

  A `trial-event` is either a `trial-start` or a `verdict`.

- **[reader]** **trial** *trial-event (:trial)*

- **[condition]** **trial-start** *trial-event*

  `trial-start` is signalled when a test function (see Tests) is entered and a `trial` is started. When this happens that trial is already the `current-trial`, and the Trial Restarts are available. It is also signalled when a trial is retried:

```
(let ((*print* nil)
      (n 0))
  (with-test ()
    (handler-bind ((trial-start
                    (lambda (c)
                      (format t "TRIAL-START for ~S retry#~S~%"
                              (test-name (trial c))
                              (n-retries (trial c))))))
      (with-test (this)
        (incf n)
        (when (< n 3)
          (retry-trial))))))
.. TRIAL-START for THIS retry#0
.. TRIAL-START for THIS retry#1
```

```
.. TRIAL-START for THIS retry#2
..
```

The matching of `trial-start` events is less straightforward than that of other `event`s.

- When a `trial-start` event matches the `collect` type (see Collecting Events), its `trial` is collected.
- Similarly, when a `trial-start` matches the `print` type (see Printing Events), it is printed immediately, and its trial's `verdict` will be printed too regardless of whether it matches `print`. If `trial-start` does not match `print`, it may still be printed if for example `*print-parent*` requires it.
- When a `trial-start` matches the `rerun` type (see Rerunning Trials), its `trial` may be rerun.
- Also, see `with-skip`.

- **[condition]** **verdict** *trial-event outcome*

A `verdict` is the `outcome` of a `trial`. It is one of `{expected,unexpected}-verdict-{success,failure}`, `verdict-skip` and `verdict-abort*`. Regarding how the verdict type is determined, see Trial Verdicts.

Verdicts are signalled while their `trial` is still the `current-trial`, and Trial Restarts are still available.

```
(try (lambda ()
       (handler-bind (((and verdict failure) #'retry-trial))
         (with-test (this)
           (is (zerop (random 2)))))))
.. (TRY #<FUNCTION (LAMBDA ()) {53038ADB}>)
..   THIS
..     ☒ (IS (ZEROP #1=(RANDOM 2)))
..       where
..         #1# = 1
..   THIS retry #1
..     · (IS (ZEROP (RANDOM 2)))
..   · THIS ·1
.. · (TRY #<FUNCTION (LAMBDA ()) {53038ADB}>) ·1
..
==> #<TRIAL (TRY #<FUNCTION (LAMBDA ()) {53038ADB}>) EXPECTED-SUCCESS 0.000s
↪  ·1>
```

- **[condition]** **expected-verdict-success** *expected verdict success*
- **[condition]** **unexpected-verdict-success** *unexpected verdict success*
- **[condition]** **expected-verdict-failure** *expected verdict failure*
- **[condition]** **unexpected-verdict-failure** *unexpected verdict failure*
- **[condition]** **verdict-skip** *verdict skip*

- **[condition]** **verdict-abort*** *verdict abort* dismissal*

Trial Verdicts When a trial has finished, a `verdict` is signalled. The verdict's type is determined as follows.

- It is a `verdict-skip` if
  - `skip-trial` was called on the trial, or
  - `abort-trial`, `skip-trial`, or `retry-trial` was called on an enclosing trial, and these were not overruled by a later `abort-trial` or `retry-trial` on the trial.
- It is a `verdict-abort*` if `abort-trial` was called on the trial, and it wasn't overruled by a later `skip-trial` or `retry-trial`.
- If all children (including those not collected in `children`) of the trial `pass`, then the verdict will be a `success`, else it will be a `failure`.
- Subject to the `with-expected-outcome` in effect, `{expected,unexpected}-verdict-{success,failure}` is the type of the verdict which will be signalled.

The verdict of this type is signalled, but its type can be changed by the Outcome Restarts or the Trial Restarts before `record-event` is invoked on it.

- **[reader]** **verdict** *trial (= nil)*

  The `verdict` `event` signalled when this `trial` finished or `nil` if it has not finished yet.

- **[function]** **runningp** *trial*

  See if the function call associated with `trial` has not returned yet. Trials that are not running have a `verdict` and are said to be finished.

- **[function]** **passedp** *trial*

  See if `trial` has finished and its `verdict` is a `pass`.

- **[function]** **failedp** *trial*

  See if `trial` has finished and its `verdict` is a `fail`.

Trial Restarts There are three restarts available for manipulating running trials: `abort-trial`, `skip-trial`, and `retry-trial`. They may be invoked programmatically or from the debugger. `abort-trial` is also invoked by `try` when encountering `unhandled-error`.

The functions below invoke one of these restarts associated with a `trial`. It is an error to call them on trials that are not `runningp`, but they may be called on trials other than the `current-trial`. In that case, any intervening trials are skipped.

```
;; Skipped trials are marked with '-' in the output.
(with-test (outer)
  (with-test (inner)
    (is t)
    (skip-trial nil outer)))
.. OUTER
```

```
..   INNER
..     · (IS T)
..   - INNER ·1
.. - OUTER ·1
..
==> #<TRIAL (WITH-TEST (OUTER)) SKIP 0.000s ·1>
```

Furthermore, all three restarts initiate a non-local exit to return from the trial. If during the unwinding of the stack, the non-local-exit is cancelled (see cancelled non-local exit), the appropriate restart will be invoked upon returning from the trial. In the following example, the non-local exit from a skip is cancelled by a `throw`.

```
(with-test (some-test)
  (catch 'foo
    (unwind-protect
         (skip-trial)
      (throw 'foo nil)))
  (is t :msg "check after skip"))
.. SOME-TEST
..   · check after skip
.. - SOME-TEST ·1
..
==> #<TRIAL (WITH-TEST (SOME-TEST)) SKIP 0.000s ·1>
```

In the next example, the non-local exit from a skip is cancelled by an `error(0 1)`, which triggers an `abort-trial`.

```
(let ((*debug* nil)
      (*describe* nil))
  (with-test (foo)
    (unwind-protect
         (skip-trial)
      (error "xxx"))))
.. FOO
..   ⊟ "xxx" (SIMPLE-ERROR)
.. ⊟ FOO ⊟1
..
==> #<TRIAL (WITH-TEST (FOO)) ABORT* 0.000s ⊟1>
```

All three restarts may be invoked on any `event`, including the trial's own `trial-start` and `verdict`. If their `condition` argument is an `event` (`retry-trial` has a special case here), they also record it (as in `record-event`) to ensure that when they handle an `event` in the debugger or programmatically that event is not dropped.

- **[function] abort-trial** *&optional condition (trial (current-trial))*

  Invoke the `abort-trial` restart of a `runningp trial`.

  When `condition` is a `verdict` for `trial`, `abort-trial` signals a new verdict of type `verdict-abort*`. This behaviour is similar to that of `abort-check`. Else, the `abort-trial` restart may record `condition`, then it initiates a non-local exit to return from the test function with `verdict-abort*`. If during the unwinding `skip-trial` or `retry-trial`

is called, then the abort is cancelled.

Since `abort*` is an `unexpected event`, `abort-trial` is rarely used programmatically. Signalling any error in a trial that's not caught before the trial's handler catches it will get turned into an `unhandled-error`, and `try` will invoke `abort-trial` with it. Thus, instead of invoking `abort-trial` directly, signalling an error will often suffice.

- **[function]** **skip-trial** *&optional condition (trial (current-trial))*

  Invoke the `skip-trial` restart of a `runningp trial`.

  When `condition` is a `verdict` for `trial`, `skip-trial` signals a new verdict of type `verdict-skip`. This behaviour is similar to that of `skip-check`. Else, the `skip-trial` restart may record `condition`, then it initiates a non-local exit to return from the test function with `verdict-skip`. If during the unwinding `abort-trial` or `retry-trial` is called, then the skip is cancelled.

```
(with-test (skipped)
  (handler-bind ((unexpected-result-failure #'skip-trial))
    (is nil)))
.. SKIPPED
..   ☒ (IS NIL)
.. - SKIPPED ☒1
..
==> #<TRIAL (WITH-TEST (SKIPPED)) SKIP 0.000s ☒1>
```

  Invoking `skip-trial` on the `trial`'s own `trial-start` skips the trial being started.

```
(let ((*print* '(or outcome leaf)))
  (with-test (parent)
    (handler-bind ((trial-start #'skip-trial))
      (with-test (child)
        (is nil)))))
.. PARENT
..   - CHILD
.. · PARENT
..
```

- **[function]** **retry-trial** *&optional condition (trial (current-trial))*

  Invoke the `retry-trial` restart of `runningp trial`. The `retry-trial` restart may record `condition`, then it initiates a non-local exit to go back to the beginning of the test function. If the non-local exit completes, then

  - (`n-retries trial`) is incremented,
  - collected results and trials are cleared (see Collecting Events),
  - counts are zeroed (see Counting Events), and
  - `trial-start` is signalled again.

  If during the unwinding `abort-trial` or `skip-trial` is called, then the retry is cancelled.

`condition` (which may be `nil`) is recorded if it is an `event` but not the `verdict` of `trial`, and the `record-event` restart is available.

- **[reader]** **n-retries** *trial (:n-retries = 0)*

  The number of times this `trial` has been retried. See `retry-trial`.

4.7 Errors

- **[condition]** **error*** *abort* leaf*

  Either `unhandled-error` or `nlx`, `error*` causes or represents abnormal termination of a `trial`. `abort-trial` can be called with `error*`s, but there is little need for explicitly doing so as `record-event`, which `try` invokes, takes care of this.

- **[reader]** **test-name** *error* (:test-name)*

- **[condition]** **unhandled-error** *error**

  Signalled when a `cl:error` condition reaches the handlers set up by `deftest` or `with-test`, or when their `*debugger-hook*` is invoked with a condition that's not an `event`.

  Note that if `nested-condition` (the original `cl:error`) has restarts associated with it, these are not going to be associated with its `unhandled-error` condition, which may restrict debugger the list of available restarts in the debugger.

- **[reader]** **nested-condition** *unhandled-error (:condition = 'nil)*

- **[reader]** **backtrace-of** *unhandled-error (:backtrace = 'nil)*

- **[reader]** **debugger-invoked-p** *unhandled-error (:debugger-invoked-p = 'nil)*

- **[variable]** ***gather-backtrace*** *t*

  Try var. Capturing the backtrace can be expensive. `*gather-backtrace*` controls whether `unhandled-error`s shall have their `backtrace-of` populated. Also, see `*print-backtrace*`.

- **[condition]** **nlx** *error**

  Representing a non-local exit of unknown origin, this is signalled if a `trial` does not return normally although it should have because it was not dismissed (see `dismissal`, `skip-trial`, `abort-trial`). In this case, there is no `cl:error(0 1)` associated with the event.

4.8 Categories

Categories determine how event types are printed and events of what types are counted together.

The default value of `*categories*` is

```
((abort*             :marker "⊟")
 (unexpected-failure :marker "⊠")
 (unexpected-success :marker "⊡")
```

```
(skip               :marker "-")
(expected-failure   :marker "×")
(expected-success   :marker "·"))
```

which says that all concrete events that are of type abort* (i.e. result-abort*, verdict-abort*, unhandled-error, and nlx) are to be marked with "⊟" when printed (see Printing Events). Also, the six types define six counters for Counting Events. Note that unexpected events have the same marker as their expected counterpart but squared.

- **[variable]** ***categories*** "- *see above* -"

  Try var. A list of elements like (type &key marker). When Printing Events, Concrete Events are printed with the marker of the first matching type. When Counting Events, the counts associated with all matching types are incremented.

- **[function]** **fancy-std-categories**

  Returns the default value of *categories* (see Categories), which contains some fancy Unicode characters.

- **[function]** **ascii-std-categories**

  Returns a value suitable for *categories*, which uses only ASCII characters for the markers.

```
'((abort*             :marker "!")
  (unexpected-failure :marker "F")
  (unexpected-success :marker ":")
  (skip               :marker "-")
  (expected-failure   :marker "f")
  (expected-success   :marker "."))
```

5 The is Macro

is is the fundamental one among Checks, on which all the others are built, and it is a replacement for cl:assert that can capture values of subforms to provide context to failures:

```
(is (= (1+ 5) 0))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS (= #1=(1+ 5) 0))
..   where
..     #1# = 6
```

is automatically captures values of arguments to functions like 1+ in the above example. Values of other interesting subforms can be explicitly requested to be captured. is supports capturing multiple values and can be taught how to deal with macros. The combination of these features allows match-values to be implementable as a tiny extension:

```
(is (match-values (values (1+ 5) "sdf")
      (= * 0)
```

```
      (string= * "sdf")))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS
..      (MATCH-VALUES #1=(VALUES (1+ 5) #2="sdf")
..        (= * 0)
..        (STRING= * "sdf")))
..   where
..     #1# == 6
..            #2#
```

`is` is flexible enough that all other checks (`signals`, `signals-not`, `invokes-debugger`, `invokes-debugger-not`, `fails`, and `in-time`) are built on top of it.

- **[macro]** **is** *form &key msg ctx (capture t) (print-captures t) (retry t)*

  If `form` returns `nil`, signal a `result` `failure`. Else, signal a `result` `success`. `is` returns normally if

  - the `record-event` restart is invoked (available when in a trial), or
  - the `continue` restart is invoked (available when not in a trial), or
  - the condition signalled last (after Outcome Restarts) is a `pass`, and it is not handled.

  If `is` returns normally after signalling an `outcome`, it returns `t` if the last condition signalled was a `success`, and `nil` otherwise.

  - `msg` and `ctx` are Format Specifier Forms. `msg` is always evaluated (as a format specifier form), and it shall print a description of the check being made, stating what the desired outcome is. The default `msg` is the whole `is` form.

    `ctx` is only evaluated if `form` evaluates to `nil`. It shall provide contextual information about the failure.

    ```
    (is (equal (prin1-to-string 'hello) "hello")
        :msg "Symbols are replacements for strings."
        :ctx ("*PACKAGE* is ~S and *PRINT-CASE* is ~S~%"
              *package* *print-case*))
    .. debugger invoked on UNEXPECTED-RESULT-FAILURE:
    ..   UNEXPECTED-FAILURE in check:
    ..     Symbols are replacements for strings.
    ..   where
    ..     (PRIN1-TO-STRING 'HELLO) = "HELLO"
    ..   *PACKAGE* is #<PACKAGE "TRY"> and *PRINT-CASE* is :UPCASE
    ..
    ```

  - If `capture` is true, the value(s) of some subforms of `form` may be automatically recorded in the condition and also made available for `ctx` via `*is-captures*`. See Captures for more.
  - If `print-captures` is true, the captures made are printed when the `result` condition is displayed in the debugger or `*describe*`d (see Printing Events). This is the `where`

`(PRIN1-TO-STRING 'HELLO) ="HELLO"` part above. If `print-captures` is `nil`, the captures are still available in `*is-captures*` for writing custom `ctx` messages.

- If `retry` is true, then the `retry-check` restart evaluates `form` again and signals a new `result`. If `retry` is `nil`, then the `retry-check` restart returns `:retry`, which allows complex checks such as `signals` to implement their own retry mechanism.

- **[variable]** ***is-form***

  `is` binds this to its `form` argument for `ctx` and `msg`.

- **[variable]** ***is-captures***

  During the evaluation of its `ctx` argument, `is` binds `*is-captures*` to the list of captures made. The list is ordered by the time of capture.

5.1 Format Specifier Form

A format specifier form is a Lisp form, typically an argument to a macro, standing for the `format-control` and `format-args` arguments to the `format` function.

It may be a constant string:

```
(is nil :msg "FORMAT-CONTROL~%with no args.")
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     FORMAT-CONTROL
..     with no args.
```

It may be a list whose first element is a constant string, and the rest are the format arguments to be evaluated:

```
(is nil :msg ("Implicit LIST ~A." "form"))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     Implicit LIST form.
```

Or it may be a form that evaluates to a list like `(format-control &rest format-args)`:

```
(is nil :msg (list "Full ~A." "form"))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     Full form.
```

Finally, it may evaluate to `nil`, in which case some context specific default is implied.

- **[function]** **canonicalize-format-specifier-form** *form*

  Ensure that the format specifier form `form` is in its full form.

5.2 Captures

During the evaluation of the `form` argument of `is`, evaluation of any form (e.g. a subform of `form`) may be recorded, which are called captures.

5.2.1 Automatic Captures

`is` automatically captures some subforms of `form` that are likely to be informative. In particular, if `form` is a function call, then non-constant arguments are automatically captured:

```
(is (= 3 (1+ 2) (- 4 3)))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS (= 3 #1=(1+ 2) #2=(- 4 3)))
..   where
..     #1# = 3
..     #2# = 1
```

By default, automatic captures are not made for subforms deeper in `form`, except for when `form` is a call to `null`, `endp` and `not`:

```
(is (null (find (1+ 1) '(1 2 3))))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS (NULL #1=(FIND #2=(1+ 1) '(1 2 3))))
..   where
..     #2# = 2
..     #1# = 2
```

```
(is (endp (member (1+ 1) '(1 2 3))))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS (ENDP #1=(MEMBER #2=(1+ 1) '(1 . #3=(2 3)))))
..   where
..     #2# = 2
..     #1# = #3#
```

Note that the argument of `not` is not captured as it is assumed to be `nil` or `t`. If that's not true, use `null`.

```
(is (not (equal (1+ 5) 6)))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS (NOT (EQUAL #1=(1+ 5) 6)))
..   where
..     #1# = 6
```

Other automatic captures are discussed with the relevant functionality such as `match-values`.

Writing Automatic Capture Rules

- **[structure] sub**

  A `sub` (short for substitution) says that in the original form `is` is checking, a `subform` was substituted (by `substitute-is-form`) with `var` (if `valuesp` is `nil`) or with (`values-list var`) if `valuesp` is true. Conversely, `var` is to be bound to the evaluated `new-form` if `valuesp` is `nil`, and to (`multiple-value-list new-form`) if `valuesp`. `new-form` is often

`eq` to `subform`, but it may be different, which is the case when further substitutions are made within a substitution.

- **[function]** **make-sub** *var subform new-form valuesp*
- **[structure-accessor]** **sub-var** *sub*
- **[structure-accessor]** **sub-subform** *sub*
- **[structure-accessor]** **sub-new-form** *sub*
- **[structure-accessor]** **sub-valuesp** *sub*
- **[generic-function]** **substitute-is-list-form** *first form env*

  In the list `form`, whose `car` is `first`, substitute subexpressions of interest with a `gensym` and return the new form. As the second value, return a list of `sub`s.

  For example, consider `(is (find (foo) list))`. When `substitute-is-list-form` is invoked on `(find (foo) list)`, it substitutes each argument of `find` with a variable, returning the new form `(find temp1 temp2)` and the list of two substitutions `((temp2 (foo) (foo) nil) (temp3 list list nil))`. This allows the original form to be rewritten as

```
(let* ((temp1 (foo))
       (temp2 list))
  (find temp1 temp2))
```

  `temp1` and `temp2` may then be reported in the `outcome` condition signalled by `is` like this:

```
  The following check failed:
    (is (find #1=(foo) #2=list))
  where
    #1# = <return-value-of-foo>
    #2# = <value-of-variable-list>
```

5.2.2 Explicit Captures

In addition to automatic captures, which are prescribed by rewriting rules (see Writing Automatic Capture Rules), explicit, ad-hoc captures can also be made.

```
(is (let ((x 1))
      (= (capture x) 2)))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS
..      (LET ((X 1))
..        (= (CAPTURE X) 2)))
..   where
..     X = 1
```

If `capture` showing up in the form that `is` prints is undesirable, then `%` may be used instead:

```
(is (let ((x 1))
      (= (% x) 2)))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS
..      (LET ((X 1))
..        (= X 2)))
..   where
..     X = 1
```

Multiple values may be captured with `capture-values` and its secretive counterpart `%%`:

```
(is (= (%% (values 1 2)) 2))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS (= #1=(VALUES 1 2) 2))
..   where
..     #1# == 1
..            2
```

where printing `==` instead of `=` indicates that this is a multiple value capture.

- **[macro]** **capture** *form*

  Evaluate `form`, record its primary return value if within the dynamic extent of an `is` evaluation, and finally return that value. If `capture` is used within the lexical scope of `is`, then `capture` itself will show up in the form that the default `msg` prints. Thus it is recommended to use the equivalent `macrolet` `%` in the lexical scope as `%` is removed before printing.

- **[macro]** **capture-values** *form*

  Like `capture-values`, but records and return all values returned by `form`. It is recommended to use the equivalent `macrolet` `%%` in the lexical scope as `%%` is removed before printing.

- **[macrolet]** **%** *form*

  An alias for `capture` in the lexical scope of `is`. Removed from the `is` form when printed.

- **[macrolet]** **%%** *form*

  An alias for `capture-values` in the lexical scope of `is`. Removed from the `is` form when printed.

6 Check Library

In the following, various checks built on top of `is` are described. Many of them share a number of arguments, which are described here.

- `on-return` is a boolean that determines whether the check in a macro that wraps `body` is made when `body` returns normally.

- `on-nlx` is a boolean that determines whether the check in a macro that wraps `body` is made when `body` performs a non-local exit.
- `msg` and `ctx` are Format Specifier Forms as in `is`.
- `name` may be provided so that it is printed (with `prin1`) instead of `body` in `msg`.

6.1 Checking Conditions

The macros `signals`, `signals-not`, `invokes-debugger`, and `invokes-debugger-not` all check whether a condition of a given type, possibly also matching a predicate, was signalled. In addition to those already described in Check Library, these macros share a number of arguments.

Matching conditions are those that are of type `condition-type` (not evaluated) and satisfy the predicate `pred`.

When `pred` is `nil`, it always matches. When it is a string, then it matches if it is a substring of the printed representation of the condition being handled (by `princ` under `with-standard-io-syntax`). When it is a function, it matches if it returns true when called with the condition as its argument.

The check is performed in the cleanup form of an `unwind-protect` around `body`. If the `current-trial` is performing an `abort-trial`, `skip-trial` or `retry-trial`, then `result-skip` is signalled.

`handler` is called when a matching condition is found. It can be a function, `t`, or `nil`. When it is a function, it is called from the condition handler (`signals` and `signals-not`) or the debugger hook (`invokes-debugger` and `invokes-debugger-not`) with the matching condition. `handler` may perform a non-local exit. When `handler` is `t`, the matching condition is handled by performing a non-local exit to just outside `body`. If the exit completes, `body` is treated as if it had returned normally, and `on-return` is consulted. When `handler` is `nil`, no additional action is performed when a matching condition is found.

The default `ctx` describes the result of the matching process in terms of `*condition-matched-p*` and `*best-matching-condition*`.

- **[variable]** ***condition-matched-p***

  When a check described in Checking Conditions signals its `outcome`, this variable is bound to a boolean value to indicate whether a condition that matched `condition-type` and `pred` was found.

- **[variable]** ***best-matching-condition***

  Bound when a check described in Checking Conditions signals its `outcome`. If `*condition-matched-p*`, then it is the most recent condition that matched both `condition-type` and `pred`. Else, it is the most recent condition that matched `condition-type` or `nil` if no such conditions were detected.

- **[macro]** **signals** *(condition-type &key pred (handler t) (on-return t) (on-nlx t) name msg ctx) &body body*

Check that body signals a condition of condition-type (not evaluated) that matches pred. To detect matching conditions, signals sets up a handler-bind. Thus it can only see what body does not handle. The arguments are described in Checking Conditions.

```
(signals (error)
  (error "xxx"))
=> NIL
```

The following example shows a failure where condition-type matches but pred does not.

```
(signals (error :pred "non-matching")
  (error "xxx"))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (ERROR "xxx") signals a condition of type ERROR that matches
..     "non-matching".
..   The predicate did not match "xxx".
```

- **[macro]** **signals-not** *(condition-type &key pred (handler t) (on-return t) (on-nlx t) name msg ctx) &body body*

  Check that body does not signal a condition of condition-type (not evaluated) that matches pred. To detect matching conditions, signals-not sets up a handler-bind. Thus, it can only see what body does not handle. The arguments are described in Checking Conditions.

- **[macro]** **invokes-debugger** *(condition-type &key pred (handler t) (on-return t) (on-nlx t) name msg ctx) &body body*

  Check that body enters the debugger with a condition of condition-type (not evaluated) that matches pred. To detect matching conditions, invokes-debugger sets up a *debugger-hook*. Thus, if *debugger-hook* is changed by body, it may not detect the condition. The arguments are described in Checking Conditions.

  Note that in a trial (see current-trial), all error(0 1)s are handled, and a *debugger-hook* is set up (see unhandled-error). Thus, invoking the debugger would normally cause the trial to abort.

```
(invokes-debugger (error :pred "xxx")
  (handler-bind ((error #'invoke-debugger))
    (error "xxx")))
=> NIL
```

- **[macro]** **invokes-debugger-not** *(condition-type &key pred (handler t) (on-return t) (on-nlx t) name msg ctx) &body body*

  Check that body does not enter the debugger with a condition of condition-type (not evaluated) that matches pred. To detect matching conditions, invokes-debugger-not sets up a *debugger-hook*. Thus, if *debugger-hook* is changed by body, it may not detect the condition. The arguments are described in Checking Conditions.

6.2 Miscellaneous Checks

- **[macro]** **fails** *(&key name msg ctx) &body body*

  Check that `body` performs a non-local exit but do not cancel it (see cancelled non-local exit). See Check Library for the descriptions of the other arguments.

  In the following example, `fails` signals a `success`.

```
(catch 'foo
  (fails ()
    (throw 'foo 7)))
=> 7
```

  Next, `fails` signals an `unexpected-failure` because `body` returns normally.

```
(fails ()
  (print 'hey))
..
.. HEY
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (PRINT 'HEY) does not return normally.
```

  Note that there is no `fails-not` as `with-test` fills that role.

- **[macro]** **in-time** *(seconds &key (on-return t) (on-nlx t) name msg ctx) &body body*

  Check that `body` finishes in `seconds`. See Check Library for the descriptions of the other arguments.

```
(in-time (1)
  (sleep 2))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (SLEEP 2) finishes within 1s.
..   Took 2.000s.
```

  `retry-check` restarts timing.

- **[variable]** ***in-time-elapsed-seconds***

  Bound to the number of seconds passed during the evaluation of `body` when `in-time` signals its `outcome`.

6.3 Check Utilities

These utilities are not checks (which signal `outcome`s) but simple functions and macros that may be useful for writing `is` checks.

- **[macro]** **on-values** *form &body body*

  `on-values` evaluates `form` and transforms its return values one by one based on forms in `body`. The Nth value is replaced by the return value of the Nth form of `body` evaluated with

`*` bound to the Nth value. If the number of values exceeds the number of transformation forms in `body` then the excess values are returned as is.

```
(on-values (values 1 "abc" 7)
  (1+ *)
  (length *))
=> 2
=> 3
=> 7
```

If the number of values is less than the number of transformation forms, then in later transformation forms `*` is bound to `nil`.

```
(on-values (values)
  *
  *)
=> NIL
=> NIL
```

The first forms in `body` may be options. Options must precede transformation forms. With `:truncate t`, the excess values are discarded.

```
(on-values (values 1 "abc" 7)
  (:truncate t)
  (1+ *)
  (length *))
=> 2
=> 3
```

The `:on-length-mismatch` option may be `nil` or a function of a single argument. If the number of values and the number of transformation forms are different, then this function is called to transform the list of values. `:truncate` is handled before `:on-length-mismatch`.

```
(on-values 1
  (:on-length-mismatch (lambda (values)
                         (if (= (length values) 1)
                             (append values '("abc"))
                             values)))
  (1+ *)
  *)
=> 2
=> "abc"
```

If the same option is specified multiple times, the first one is in effect.

- **[macro]** **match-values** *form &body body*

  `match-values` returns true iff all return values of `form` satisfy the predicates given by `body`, which are described in `on-values`. The `:on-length-mismatch` and `:truncate` options of `on-values` are supported. If no `:on-length-mismatch` option is specified, then `match-values` returns `nil` on length mismatch.

```
;; no values
(is (match-values (values)))
;; single value success
(is (match-values 1
      (= * 1)))
;; success with different types
(is (match-values (values 1 "sdf")
      (= * 1)
      (string= * "sdf")))
;; too few values
(is (not (match-values 1
           (= * 1)
           (string= * "sdf"))))
;; too many values
(is (not (match-values (values 1 "sdf" 3)
           (= * 1)
           (string= * "sdf"))))
;; too many values, but truncated
(is (match-values (values 1 "sdf" 3)
      (:truncate t)
      (= * 1)
      (string= * "sdf")))
```

- **[function]** **mismatch%** *sequence1 sequence2 &key from-end (test #'eql) (start1 0) end1 (start2 0) end2 key max-prefix-length max-suffix-length*

  Like `cl:mismatch` but `capture`s and returns the common prefix and the mismatched suffixes. The `test-not` argument is deprecated by the `clhs` and is not supported. In addition, if `max-prefix-length` and `max-suffix-length` are non-`nil`, they must be non-negative integers, and they limit the number of elements in the prefix and the suffixes.

```
(is (null (mismatch% '(1 2 3) '(1 2 4 5))))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS (NULL #1=(MISMATCH% '(1 2 3) '(1 2 4 5))))
..   where
..     COMMON-PREFIX = (1 2)
..     MISMATCHED-SUFFIX-1 = (3)
..     MISMATCHED-SUFFIX-2 = (4 5)
..     #1# = 2
```

```
(is (null (mismatch% "Hello, World!"
                     "Hello, world!")))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS (NULL #1=(MISMATCH% "Hello, World!" "Hello, world!")))
..   where
..     COMMON-PREFIX = "Hello, "
..     MISMATCHED-SUFFIX-1 = "World!"
..     MISMATCHED-SUFFIX-2 = "world!"
..     #1# = 7
```

- **[function]** **different-elements** *sequence1 sequence2 &key (pred #'eql) (missing :missing)*

  Return the different elements under `pred` in the given sequences as a list of `(:index <index> <e1> <e2>)` elements, where `e1` and `e2` are elements of `sequence1` and `sequence2` at `<index>`, respectively, and they may be `missing` if the corresponding sequence is too short.

```
(is (endp (different-elements '(1 2 3) '(1 b 3 d))))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS (ENDP #1=(DIFFERENT-ELEMENTS '(1 2 3) '(1 B 3 D))))
..   where
..     #1# = ((:INDEX 1 2 B) (:INDEX 3 :MISSING D))
```

- **[function]** **same-set-p** *list1 list2 &key key (test #'eql)*

  See if `list1` and `list2` represent the same set. See `cl:set-difference` for a description of the `key` and `test` arguments.

```
(try:is (try:same-set-p '(1) '(2)))
.. debugger invoked on UNEXPECTED-RESULT-FAILURE:
..   UNEXPECTED-FAILURE in check:
..     (IS (SAME-SET-P '(1) '(2)))
..   where
..     ONLY-IN-1 = (1)
..     ONLY-IN-2 = (2)
```

- **[macro]** **with-shuffling** *nil &body body*

  Execute the forms that make up the list of forms `body` in random order and return `nil`. This may be useful to prevent writing tests that accidentally depend on the order in which subtests are called.

```
(loop repeat 3 do
  (with-shuffling ()
    (prin1 1)
    (prin1 2)))
.. 122112
=> NIL
```

6.3.1 Comparing Floats

Float comparisons following https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/.

- **[function]** **float-≅** *x y &key (max-diff-in-value *max-diff-in-value*) (max-diff-in-ulp *max-diff-in-ulp*)*

  Return whether two numbers, `x` and `y`, are approximately equal either according to `max-diff-in-value` or `max-diff-in-ulp`.

  If the absolute value of the difference of two floats is not greater than `max-diff-in-value`, then they are considered equal.

If two floats are of the same sign and the number of representable floats (ULP, unit in the last place) between them is less than `max-diff-in-ulp`, then they are considered equal.

If neither `x` nor `y` is a float, then the comparison is done with `=`. If one of them is a `double-float`, then the other is converted to a double float, and the comparison takes place in double-float space. Else, both are converted to `single-float` and the comparison takes place in single-float space.

- **[variable]** ***max-diff-in-value*** *1.0e-16*

  The default value of the `max-diff-in-value` argument of `float-≅`.

- **[variable]** ***max-diff-in-ulp*** *2*

  The default value of the `max-diff-in-ulp` argument of `float-≅`.

- **[function]** **float-≲** *x y &key (max-diff-in-value *max-diff-in-value*) (max-diff-in-ulp *max-diff-in-ulp*)*

  Return whether `x` is approximately less than `y`. Equivalent to `<`, but it also allows for approximate equality according to `float-≅`.

- **[function]** **float-≳** *x y &key (max-diff-in-value *max-diff-in-value*) (max-diff-in-ulp *max-diff-in-ulp*)*

  Return whether `x` is approximately greater than `y`. Equivalent to `>`, but it also allows for approximate equality according to `float-≅`.

7 Tests

In Try, tests are Lisp functions that record their execution in `trial` objects. `trial`s are to tests what function call traces are to functions. In more detail, tests

- create a `trial` object and signal a `trial-start` event upon entry to the function,
- signal a `verdict` condition before returning normally or via a non-local exit,
- return the `trial` object as the first value,
- return explicitly returned values as the second, third, and so on values.

See `deftest` and `with-test` for more precise descriptions.

- **[macro]** **deftest** *name lambda-list &body body*

  `deftest` is a wrapper around `defun` to define global test functions. See `defun` for a description of `name`, `lambda-list`, and `body`. The behaviour common with `with-test` is described in Tests.

```
(deftest my-test ()
  (write-string "hey"))
=> MY-TEST

(test-bound-p 'my-test)
```

```
=> T

(my-test)
.. hey
==> #<TRIAL (MY-TEST) EXPECTED-SUCCESS 0.000s>
```

Although the common case is for tests to have no arguments, `deftest` supports general function lambda lists. Within a global test,

- `name` is bound to the `trial` object
- the first return value is the trial
- values are not returned implicitly
- values returned with an explicit `return-from` are returned as values after the trial

```
(deftest my-test ()
  (prin1 my-test)
  (return-from my-test (values 2 3)))

(my-test)
.. #<TRIAL (MY-TEST) RUNNING>
==> #<TRIAL (MY-TEST) EXPECTED-SUCCESS 0.000s>
=> 2
=> 3
```

- **[variable]** ***run-deftest-when*** *nil*

  This may be any of `:compile-toplevel`, `:load-toplevel`, `:execute`, or a list thereof. The value of `*run-deftest-when*` determines in what `eval-when` situation to call the test function immediately after it has been defined with `deftest`.

  For interactive development, it may be convenient to set it to `:execute` and have the test run when the `deftest` is evaluated (maybe with Slime `C-M-x`, `slime-eval-defun`). Or set it to `:compile-toplevel`, and have it rerun on Slime `C-c C-c`, `slime-compile-defun`.

  If the test has required arguments, an argument list is prompted for and read from `*query-io*`.

- **[function]** **test-bound-p** *symbol*

  See if `symbol` names a global test (i.e. a test defined with `deftest`). If since the execution of `deftest`, the symbol has been `unintern`ed, `fmakunbound`ed, or redefined with `defun`, then it no longer names a global test.

- **[macro]** **with-test** *(&optional var-or-name &key (name nil)) &body body*

  Execute `body` in a `trial` to group together `check`s and other tests in its dynamic scope. `body` is executed in its lexical environment even on a rerun (see Rerunning Trials).

  If `var-or-name` is a non-`nil` symbol, it is bound to the `trial` object. `name` may be of any type, it is purely for presentation purposes. If `name` is not specified, then it defaults to `var-or-name`.

To facilitate returning values, a `block` is wrapped around `body`. The name of the block is `var-or-name` if it is a symbol, else it's `nil`.

Both `var-or-name` and `name` can be specified, but in this case `var-or-name` must be a symbol:

```
(with-test (some-feature :name "obscure feature")
  (prin1 some-feature)
  (is t)
  (return-from some-feature (values 1 2)))
.. #<TRIAL (WITH-TEST ("obscure feature")) RUNNING>
.. "obscure feature"
..   · (IS T)
.. · "obscure feature" ·1
..
==> #<TRIAL (WITH-TEST ("obscure feature")) EXPECTED-SUCCESS 0.200s ·1>
=> 1
=> 2
```

If only `var-or-name` is specified:

```
(with-test (some-feature)
  (prin1 some-feature)
  (is t)
  (return-from some-feature (values 1 2)))
.. #<TRIAL (WITH-TEST (SOME-FEATURE)) RUNNING>
.. SOME-FEATURE
..   · (IS T)
.. · SOME-FEATURE ·1
..
==> #<TRIAL (WITH-TEST (SOME-FEATURE)) EXPECTED-SUCCESS 0.000s ·1>
=> 1
=> 2
```

If neither is specified:

```
(with-test ()
  (prin1 (current-trial))
  (is t)
  (return (values 1 2)))
.. #<TRIAL (WITH-TEST (NIL)) RUNNING>
.. NIL
..   · (IS T)
.. · NIL ·1
..
==> #<TRIAL (WITH-TEST (NIL)) EXPECTED-SUCCESS 0.000s ·1>
=> 1
=> 2
```

Finally, using that `name` defaults to `var-or-name` and that it is valid to specify non-symbols for `var-or-name`, one can also write:

```
(with-test ("Some feature")
  (prin1 (current-trial))
  (is t)
  (return (values 1 2)))
.. #<TRIAL (WITH-TEST ("Some feature")) RUNNING>
.. "Some feature"
..   · (IS T)
.. · "Some feature" ·1
..
==> #<TRIAL (WITH-TEST ("Some feature")) EXPECTED-SUCCESS 0.200s ·1>
=> 1
=> 2
```

In summary and in contrast to `deftest`, `with-test`

- defines and runs a test at the same time,
- the test function cannot have arguments,
- may not bind their trial object to any variable,
- may have a `block` named `nil`,
- has a `name` purely for presentation purposes.

`with-test` can be thought of as analogous to `(funcall (lambda () body))`. The presence of the `lambda`(`0` `1`) is important because it is stored in the `trial` object to support Rerunning Trials.

- **[function]** **list-package-tests** *&optional (package *package*)*

  List all symbols in `package` that name global tests in the sense of `test-bound-p`.

- **[macro]** **with-tests-run** *(tests-run) &body body*

  Bind the symbol `tests-run` to an empty `eq` hash table and execute `body`. The hash table reflects call counts to global tests. Keys are symbols naming global tests, and the values are the number of times the keys have been called.

- **[macro]** **warn-on-tests-not-run** *(&optional (package *package*)) &body body*

  A convenience utility that records the global tests run by `body` with `with-tests-run` and, when `body` finishes, signals a warning for each global tests in `package` not run.

  This is how Try runs its own tests:

```
(defun test ()
  ;; Bind *PACKAGE* so that names of tests printed have package names,
  ;; and M-. works on them in Slime.
  (let ((*package* (find-package :common-lisp)))
    (warn-on-tests-not-run ((find-package :try))
      (print (try 'test-all
                  :print 'unexpected
                  :describe 'unexpected)))))
```

7.1 Calling Test Functions

Tests always run under `try`, but `try` may be explicit or implicit depending whether the outermost test was called via `try` or directly as a Lisp function.

Nested invocations of tests, be them explicit or implicit, do not establish nested `try`s. `event` handling is performed only at the outermost level.

- **[glossary-term] Try var**

  There are lots of special variables that affect `try`. To avoid the plight of Common Lisp `stream`s and guarantee consistent output and behaviour even if these variables are changed during a single `try` run, the values of variables are captured when `try` is first invoked. That is, when the outermost test function is entered. These variables are called Try vars.

7.1.1 Explicit `try`

We speak of an explicit `try` when the outermost test function is called directly by `try`.

Global test functions can be `try`ed explicitly by giving their name:

```
(deftest my-test ()
  (is t))

(try 'my-test)
.. MY-TEST
..   · (IS T)
.. · MY-TEST ·1
..
==> #<TRIAL (MY-TEST) EXPECTED-SUCCESS 0.000s ·1>
```

However, `with-test` has no global name, so to delay its execution until `try` calls it, it needs to be wrapped in a `lambda`(`0` `1`).

```
(try (lambda ()
       (with-test (my-test)
         (is t))))
.. (TRY #<FUNCTION (LAMBDA ()) {531FE50B}>)
..   MY-TEST
..     · (IS T)
..   · MY-TEST ·1
.. · (TRY #<FUNCTION (LAMBDA ()) {531FE50B}>) ·1
..
==> #<TRIAL (TRY #<FUNCTION (LAMBDA ()) {531FE50B}>) EXPECTED-SUCCESS 0.000s ·1>
```

In the example above, the `testable` argument is not the name of a test, so we see that `try` wraps an extra `trial` around the lambda to ensure that the tree of `trial`s has a single root. This `trial` object also remembers the lambda function for rerunning. This situation also arises when there are multiple basic Testables.

Explicit and implicit `try`s are very similar. The differences are that explicit `try`

- can run Testables (it takes care of wrapping them in a function),

- has a function argument for each of the `*debug*`, `*collect*`, etc variables.

Those arguments default to `*try-debug*`, `*try-collect*`, etc, which parallel and in turn default to `*debug*`, `*collect*`, etc when set to `:unspecified`, their default value. The exception is `*try-debug*`, which defaults to `nil`. These defaults encourage the use of an explicit `try` call in the non-interactive case and calling the test functions directly in the interactive one, but this is not enforced in any way.

- **[function]** **try** *testable &key (debug *try-debug*) (count *try-count*) (collect *try-collect*) (rerun *try-rerun*) (print *try-print*) (describe *try-describe*) (stream *try-stream*) (printer *try-printer*)*

  `try` runs `testable` and handles the `event`s to count, collect, debug, print the results of checks and trials, and to decide what tests to `skip` and what to rerun.

  `debug`, `count`, `collect`, `rerun`, `print`, and `describe` must all be valid specifiers for types that are either `nil` (the empty type) or have a non-empty intersection with the type `event` (e.g. `t`, `outcome`, `unexpected`, `verdict`).

  `try` sets up a `handler-bind` handler for `event`s and runs `testable` (see Testables). When an `event` is signalled, the handler matches its type to the value of the `debug` argument (in the sense of `(typep event debug)`). If it matches, then the debugger is invoked with the event. In the debugger, the user has a number of restarts available to change (see Event Restarts, Outcome Restarts, Check Restarts, Trial Restarts, and `set-try-debug`).

  If the debugger is not invoked, `try` invokes the very first restart available, which is always `record-event`.

  Recording the event is performed as follows.

  - Outcome counts are updated (see Counting Events).
  - The event is passed to the collector (see Collecting Events).
  - The event is passed to the printer (see Printing Events).
  - Finally, when rerunning a trial (i.e. when `testable` is a trial), on a `trial-start` event, the trial may be skipped (see Rerunning Trials).

  `try` returns the values returned by the outermost trial. This is just the `trial` object in the absence of an explicit `return` or `return-from` (see examples in Tests).

  If `try` is called within the dynamic extent of another `try` run, then it simply calls `testable`, ignores the other arguments and leaves event handling to the enclosing `try`.

- **[function]** **set-try-debug** *debug*

  Invoke the `set-try-debug` restart to override the `debug` argument of the currently running `try`. `debug` must thus be a suitable type. When the `set-try-debug` restart is invoked interactively, `debug` is read as a non-evaluated form from `*query-io*`.

- **[variable]** ***try-debug*** *nil*

  Try var. The default value for `try`'s `:debug` argument. If `:unspecified`, then the value of `*debug*` is used instead.

- **[variable]** ***try-count*** *:unspecified*

  Try var. The default value for `try`'s `:count` argument. If `:unspecified`, then the value of `*count*` is used instead.

- **[variable]** ***try-collect*** *:unspecified*

  Try var. The default value for `try`'s `:collect` argument. If `:unspecified`, then the value of `*collect*` is used instead.

- **[variable]** ***try-rerun*** *:unspecified*

  Try var. The default value for `try`'s `:rerun` argument. If `:unspecified`, then the value of `*rerun*` is used instead.

- **[variable]** ***try-print*** *:unspecified*

  Try var. The default value for `try`'s `:print` argument. If `:unspecified`, then the value of `*print*` is used instead.

- **[variable]** ***try-describe*** *:unspecified*

  Try var. The default value for `try`'s `:describe` argument. If `:unspecified`, then the value of `*describe*` is used instead.

- **[variable]** ***try-stream*** *:unspecified*

  Try var. The default value for `try`'s `:stream` argument. If `:unspecified`, then the value of `*stream*` is used instead.

- **[variable]** ***try-printer*** *:unspecified*

  Try var. The default value for `try`'s `:printer` argument. If `:unspecified`, then the value of `*printer*` is used instead.

- **[variable]** ***n-recent-trials*** *3*

  See `*recent-trials*`.

- **[function]** **recent-trial** *&optional (n 0)*

  Returns the `n`th most recent trial or `nil` if there are not enough trials recorded. Every `trial` returned by `try` gets pushed onto a list of trials, but only `*n-recent-trials*` are kept.

- **[variable]** **!** *nil*

  The most recent trial. Equivalent to `(recent-trial 0)`.

- **[variable]** **!!** *nil*

  Equivalent to `(recent-trial 1)`.

- **[variable]** **!!!** *nil*

  Equivalent to `(recent-trial 2)`.

Testables Valid first arguments to `try` are called testables. A basic testable is a function designator, which can be

- the name of a global function (as in `defun` or `deftest`), or
- a function object (including `trial`s, which are funcallable).

Composite testables are turned into a list of function designators in a recursive manner.

- When the testable is a list, this is trivial.
- When the testable is a `package`, `list-package-tests` is called on it.

With a list of function designators, `try` does the following:

- If there is only one and it is `test-bound-p`, then the test function is called directly.
- Else, `try` behaves as if its `testable` argument were an anonymous function that calls the function designators one by one. See Explicit `try` for an example.

7.1.2 Implicit `try`

We speak of an implicit `try` when the outermost test is entered via a Lisp function call. In this case, as the test function is entered, it invokes itself behind the scenes (implicitly) via `try`:

```
(try ... :debug *debug* :collect *collect* :rerun *rerun*
     :print *print* :describe *describe*
     :stream *stream* :printer *printer*)
```

As it's invoked again, it sees that it is now running under `try` and proceeds to execute normally.

An implicit `try` can only happen with the following two constructs.

- Global test function

```
(deftest my-test ()
  (is t))

(my-test)
.. MY-TEST
..   · (IS T)
.. · MY-TEST ·1
..
==> #<TRIAL (MY-TEST) EXPECTED-SUCCESS 0.004s ·1>
```

- `with-test`

```
(with-test (my-test)
  (is t))
.. MY-TEST
..   · (IS T)
.. · MY-TEST ·1
..
==> #<TRIAL (WITH-TEST (MY-TEST)) EXPECTED-SUCCESS 0.000s ·1>
```

- **[variable]** ***debug*** *(and unexpected (not nlx) (not verdict))*

  Try var. The default value makes `try` invoke the debugger on `unhandled-error`, `result-abort*`, `unexpected-result-failure`, and `unexpected-result-success`. `nlx` is excluded because it is caught as the test function is being exited, but by that time the dynamic environment of the actual cause is likely gone. `verdict` is excluded because it is a consequence of its child outcomes.

- **[variable]** ***count*** *leaf*

  Try var. Although the default value of `*categories*` lumps `result`s and `verdict`s together, with the default of `leaf`, `verdict`s are not counted. See Counting Events.

- **[variable]** ***collect*** *(or trial-event unexpected)*

  Try var. By default all `trials` and `unexpected` are collected. This is sufficient for being able to Rerunning Trials anything in context.

- **[variable]** ***rerun*** *unexpected*

  Try var. The default matches that of `*collect*`. See Rerunning Trials.

- **[variable]** ***print*** *(or leaf dismissal)*

  Try var. Events of this type are printed.

```
(concrete-events-of-type '(or leaf dismissal))
=> (EXPECTED-RESULT-SUCCESS UNEXPECTED-RESULT-SUCCESS
    EXPECTED-RESULT-FAILURE UNEXPECTED-RESULT-FAILURE RESULT-SKIP
    RESULT-ABORT* VERDICT-SKIP VERDICT-ABORT* UNHANDLED-ERROR NLX)
```

- **[variable]** ***describe*** *(or unexpected failure)*

  Try var. By default, the context (e.g. Captures, and the `ctx` argument of is and other checks) of `unexpected` events is described. See Printing Events.

- **[variable]** ***stream*** *(make-synonym-stream '*debug-io*)*

  Try var.

- **[variable]** ***printer*** *tree-printer*

  Try var.

Implementation of Implicit `try` What's happening in the implementation is that a test function, when it is called, checks whether it is running under the `try` function. If it isn't, then it invokes `try` with its `trial`. `try` realizes the trial cannot be rerun yet (see Rerunning Trials) because it is `runningp`, sets up its event handlers for debugging, collecting, printing, and invokes the trial as if it were rerun but without skipping anything based on the `rerun` argument. Thus the following are infinite recursions:

```
(with-test (recurse)
  (try recurse))
```

```
(with-test (recurse)
  (funcall recurse))
```

7.2 Printing Events

`try` instantiates a printer of the type given by its `printer` argument. All `event`s recorded by `try` are sent to this printer. The printer then prints events that match the type given by the `print` argument of `try`. Events that also match the `describe` argument of `try` are printed with context information (see `is`) and backtraces (see `unhandled-error`).

Although the printing is primarily customized with global special variables, changing the value of those variables after the printer object is instantiated by `try` has no effect. This is to ensure consistent output with nested `try` calls of differing printer setups.

- **[class] tree-printer**

  `tree-printer` prints events in an indented tree-like structure, with each internal node corresponding to a `trial`. This is the default printer (according to `*printer*` and `*try-printer*`) and currently the only one.

  The following example prints all Concrete Events.

```
(let ((*debug* nil)
      (*print* '(not trial-start))
      (*describe* nil))
  (with-test (verdict-abort*)
    (with-test (expected-verdict-success))
    (with-expected-outcome ('failure)
      (with-test (unexpected-verdict-success)))
    (handler-bind (((and verdict success) #'force-expected-failure))
      (with-test (expected-verdict-failure)))
    (handler-bind (((and verdict success) #'force-unexpected-failure))
      (with-test (unexpected-verdict-failure)))
    (with-test (verdict-skip)
      (skip-trial))
    (is t :msg "EXPECTED-RESULT-SUCCESS")
    (with-failure-expected ('failure)
      (is t :msg "UNEXPECTED-RESULT-SUCCESS")
      (is nil :msg "EXPECTED-RESULT-FAILURE"))
    (is nil :msg "UNEXPECTED-RESULT-FAILURE")
    (with-skip ()
      (is nil :msg "RESULT-SKIP"))
    (handler-bind (((and result success) #'abort-check))
      (is t :msg "RESULT-ABORT*"))
    (catch 'foo
      (with-test (nlx-test)
        (throw 'foo nil)))
    (error "UNHANDLED-ERROR")))
.. VERDICT-ABORT*                        ; TRIAL-START
..   · EXPECTED-VERDICT-SUCCESS
..   ⊡ UNEXPECTED-VERDICT-SUCCESS
..   × EXPECTED-VERDICT-FAILURE
```

```
..   ☒ UNEXPECTED-VERDICT-FAILURE
..   - VERDICT-SKIP
..   · EXPECTED-RESULT-SUCCESS
..   ⊡ UNEXPECTED-RESULT-SUCCESS
..   × EXPECTED-RESULT-FAILURE
..   ☒ UNEXPECTED-RESULT-FAILURE
..   - RESULT-SKIP
..   ⊟ RESULT-ABORT*
..   NLX-TEST                          ; TRIAL-START
..     ⊟ non-local exit                 ; NLX
..   ⊟ NLX-TEST ⊟1                     ; VERDICT-ABORT*
..   ⊟ "UNHANDLED-ERROR" (SIMPLE-ERROR)
.. ⊟ VERDICT-ABORT* ⊟3 ☒1 ⊡1 -1 ×1 ·1
..
==> #<TRIAL (WITH-TEST (VERDICT-ABORT*)) ABORT* 0.004s ⊟3 ☒1 ⊡1 -1 ×1 ·1>
```

The ⊟3 ☒1 ⊡1 -1 ×1 ·1 part is the counts for *categories* printed with their markers.

- **[variable]** ***print-parent*** *t*

Try var. When an event is signalled and its parent trial's type matches *print-parent*, the trial is printed as if its trial-start matched the print argument of try.

```
(let ((*print* 'leaf)
      (*print-parent* t))
  (with-test (t0)
    (is t)
    (is t)))
.. T0
..   · (IS T)
..   · (IS T)
.. · T0 ·2
..
==> #<TRIAL (WITH-TEST (T0)) EXPECTED-SUCCESS 0.000s ·2>
```

```
(let ((*print* 'leaf)
      (*print-parent* nil))
  (with-test (t0)
    (is t)
    (is t)))
.. · (IS T)
.. · (IS T)
..
==> #<TRIAL (WITH-TEST (T0)) EXPECTED-SUCCESS 0.000s ·2>
```

print-parent nil combined with printing verdicts results in a flat output:

```
(let ((*print* '(or leaf verdict))
      (*print-parent* nil))
  (with-test (outer)
    (with-test (inner)
      (is t :msg "inner-t"))
    (is t :msg "outer-t")))
```

```
.. · inner-t
.. · INNER ·1
.. · outer-t
.. · OUTER ·2
..
==> #<TRIAL (WITH-TEST (OUTER)) EXPECTED-SUCCESS 0.000s ·2>
```

- **[variable]** ***print-indentation*** *2*

  Try var. The number of spaces each printed `trial` increases the indentation of its children.

- **[variable]** ***print-duration*** *nil*

  Try var. If true, the number of seconds spent during execution is printed.

```
(let ((*print-duration* t)
      (*debug* nil)
      (*describe* nil))
  (with-test (timed)
    (is (progn (sleep 0.1) t))
    (is (progn (sleep 0.2) t))
    (error "xxx")))
..         TIMED
..  0.100    · (IS (PROGN (SLEEP 0.1) T))
..  0.200    · (IS (PROGN (SLEEP 0.2) T))
..           ⊟ "xxx" (SIMPLE-ERROR)
..  0.300 ⊟ TIMED ⊟1 ·2
..
==> #<TRIAL (WITH-TEST (TIMED)) ABORT* 0.300s ⊟1 ·2>
```

Timing is available for all `outcome`s (i.e. for Checks and `trial`s). Checks generally measure the time spent during evaluation of the form they are wrapping. Trials measure the time between `trial-start` and the `verdict`.

Timing information is not available for `trial-start` and `error*` events.

- **[variable]** ***print-compactly*** *nil*

  Try var. `event`s whose type matches `*print-compactly*` are printed less verbosely. `leaf` events are printed only with their marker, and `verdict`s of trials without printed child trials are printed with `=> <marker>` (see `*categories*`).

```
(let ((*print-compactly* t)
      (*debug* nil)
      (*describe* nil))
  (with-test (outer)
    (loop repeat 10 do (is t))
    (with-test (inner)
      (is t)
      (is nil)
      (error "xxx"))
    (loop repeat 10 do (is t))))
.. OUTER ··········
```

```
..   INNER ·☒⊟ => ⊟
..   ··········
.. ☒ OUTER ⊟1 ☒1 ·21
..
==> #<TRIAL (WITH-TEST (OUTER)) UNEXPECTED-FAILURE 0.000s ⊟1 ☒1 ·21>
```

print-compactly has no effect on events being describe**d**.

- **[variable]** ***print-backtrace*** *t*

  Try var. Whether to print backtraces gathered when *gather-backtrace*.

- **[variable]** ***defer-describe*** *nil*

  Try var. When an event is to be *describe*d and its type matches *defer-describe*, then instead of printing the often longish context information in the tree of events, it is deferred until after try has finished. The following example only prints leaf events (due to *print* and *print-parent*) and in compact form (see *print-compactly*), deferring description of events matching *describe* until the end.

```
(let ((*print* 'leaf)
      (*print-parent* nil)
      (*print-compactly* t)
      (*defer-describe* t)
      (*debug* nil))
  (with-test (outer)
    (loop repeat 10 do (is t))
    (with-test (inner)
      (is (= (1+ 5) 7)))))
.. ··········☒
..
.. ;; UNEXPECTED-RESULT-FAILURE (☒) in OUTER INNER:
.. (IS (= #1=(1+ 5) 7))
.. where
..   #1# = 6
..
==> #<TRIAL (WITH-TEST (OUTER)) UNEXPECTED-FAILURE 0.000s ☒1 ·10>
```

7.3 Counting Events

trials have a counter for each category in *categories*. When an event is recorded by try and its type matches *count*, the counters of all categories matching the event type are incremented in the current-trial. When a trial finishes and a verdict is recorded, the trial's event counters are added to that of its parent's (if any). The counts are printed with verdicts (see Printing Events).

If both *count* and *categories* are unchanged from their default values, then only leaf events are counted, and we get separate counters for abort*, unexpected-failure, unexpected-success, skip, expected-failure, and expected-success.

```
(let ((*debug* nil))
  (with-test (outer)
    (with-test (inner)
      (is t))
    (is t)
    (is nil)))
.. OUTER
..   INNER
..     · (IS T)
..   · INNER ·1
..   · (IS T)
..   ☒ (IS NIL)
.. ☒ OUTER ☒1 ·2
..
==> #<TRIAL (WITH-TEST (OUTER)) UNEXPECTED-FAILURE 0.000s ☒1 ·2>
```

As the above example shows, `expected-verdict-success` and `expected-result-success` are both marked with `"·"`, but only `expected-result-success` is counted due to `*count*` being `leaf`.

7.4 Collecting Events

When an `event` is recorded and the type of the `event` matches the `collect` type argument of `try`, then a corresponding object is pushed onto `children` of the `current-trial` for subsequent Rerunning Trials or Reprocessing Trials.

In particular, if the matching event is a `leaf`, then the event itself is collected. If the matching event is a `trial-event`, then `verdict` of its `trial` is collected. Furthermore, trials which collected anything are always collected by their parent.

By default, both implicit and explicit calls to `try` collect the `unexpected` (see `*collect*` and `*try-collect*`), and consequently all the enclosing trials.

- **[reader]** **children** *trial* *(:children = nil)*

  A list of immediate child `verdict`s, `result`s, and `error*`s collected in reverse chronological order (see Collecting Events). The `verdict` of this `trial` is not among `children`, but the `verdict`s of child trials' are.

7.5 Rerunning Trials

When a `trial` is `funcall`ed or passed to `try`, the *test that created the trial* is invoked, and it may be run again in its entirety or in part. As the test runs, it may invoke other tests. Any test (including the top-level one) is skipped if it does not correspond to a collected trial or its `trial-start` event and `verdict` do not match the `rerun` argument of `try`. When that happens, the corresponding function call immediately returns the `trial` object. In trials that are rerun, Checks are executed normally.

- A new trial is skipped (as if with `skip-trial`) if `rerun` is not `t` and

    - there is no trial representing the same function call among the collected but not yet rerun trials in the trial being rerun, or
    - the first such trial does not match the `rerun` type argument of `try` in that neither its `trial-start`, `verdict` events match the type `rerun`, nor do any of its collected `result`s and trials.

- If `rerun` is `t`, then the test is run in its entirety, including even the non-collected trials. Use `rerun event` to run only the collected trials.
- The *test that created the trial* is determined as follows.
    - If the trial was created by calling a `deftest` function, then the test currently associated with that symbol naming the function is called with the arguments of the original function call. If the symbol is no longer `fboundp` (because it was `fmakunbound`) or it no longer names a `deftest` (it was redefined with `defun`), then an error is signalled.
    - If the trial was created by entering a `with-test` form, then its body is executed again in the original lexical but the current dynamic environment. Implementationally speaking, `with-test` defines a local function of no arguments (likely a closure) that wraps its body, stores the closure in the trial object and calls it on a rerun in a `with-test` with the same `var-or-name` and same `name`.
    - If the trial was created by `try` itself to ensure that all events are signalled in a trial (see Explicit `try`), then on a rerun the same `testable` is run again.

  All three possibilities involve entering `deftest` or `with-test`, or invoking `try`: the same cases that we have with Implicit `try`. Thus, even if a trial is rerun with `funcall`, execution is guaranteed to happen under `try`.

- **[variable]** **`*rerun-context*`** *nil*

  Try var. A `trial` or `nil`. If it's a `trial`, then `try` will rerun this trial skipping everything that does not lead to an invocation of a basic testable in its `testable` argument (see Testables). If no route to any basic testable function can be found among the collected events of the context, then a warning is signalled and the context is ignored.

  Consider the following code evaluated in the package `try`:

```
(deftest test-try ()
  (let ((*package* (find-package :cl-user)))
    (test-whatever)
    (test-printing)))

(deftest test-whatever ()
  (is t))

(deftest test-printing ()
  (is (equal (prin1-to-string 'x) "TRY::X")))

(test-try)
.. TEST-TRY
..   TEST-WHATEVER
..     · (IS T)
```

```
..   · TEST-WHATEVER ·1
..   TEST-PRINTING
..     · (IS (EQUAL (PRIN1-TO-STRING 'X) "TRY::X"))
..   · TEST-PRINTING ·1
.. · TEST-TRY ·2
..
==> #<TRIAL (TEST-TRY) EXPECTED-SUCCESS 0.500s ·2>

;; This could also be an implicit try such as (TEST-PRINTING),
;; but this way we avoid entering the debugger.
(try 'test-printing)
.. TEST-PRINTING
..   ☒ (IS (EQUAL #1=(PRIN1-TO-STRING 'X) "TRY::X"))
..     where
..       #1# = "X"
.. ☒ TEST-PRINTING ☒1
..
==> #<TRIAL (TEST-PRINTING) UNEXPECTED-FAILURE 0.200s ☒1>
```

`test-printing` fails because when called directly, `*package*` is not the expected `cl-user`. However, when `*rerun-context*` is set, `test-printing` will be executed in the correct dynamic environment.

```
(setq *rerun-context*
      ;; Avoid the debugger, as a matter of style.
      (try 'test-try))

(test-printing)
.. TEST-TRY
..   - TEST-WHATEVER
..   TEST-PRINTING
..     · (IS (EQUAL (PRIN1-TO-STRING 'X) "TRY::X"))
..   · TEST-PRINTING ·1
.. · TEST-TRY ·1
..
```

Note how `test-whatever` was `skip`ped because it leads to no calls to `test-printing`.

See Emacs Integration for a convenient way of taking advantage of this feature.

7.6 Reprocessing Trials

- **[function]** **replay-events** *trial &key (collect *try-collect*) (print *try-print*) (describe *try-describe*) (stream *try-stream*) (printer *try-printer*)*

  `replay-events` reprocesses the events collected in `trial` without actually running the tests that produced them. It simply signals the events collected in `trial` again to allow further processing. It takes the same arguments as `try` except `debug`, `count` and `rerun`. The values of `*categories*` and `*count*` that were in effect for `trial` are used, and their current values are ignored to be able to keep consistent counts (see Counting Events).

  Suppose we ran a large test using the default `:print` and into a rare non-deterministic

bug.

```
(deftest some-test ()
  (with-test (inner)
    (is t)
    (is (= 10 7))   ; fake non-deterministic bug
    (is t))
  (error "my-msg"))

(try 'some-test)
```

Now, the output is too large with `expected` events and the backtrace cluttering it. We could try running the test again with different settings, or even just rerunning it, but that might make the bug go away. Instead of searching for the interesting bits in the text output, we can replay the events and print only the `unexpected` events:

```
(let ((*print-backtrace* nil))
  (replay-events ! :print 'unexpected))
.. SOME-TEST
..   INNER
..     ☒ (IS (= 10 7))
..   ☒ INNER ☒1 ·2
..   ⊟ "my-msg" (SIMPLE-ERROR)
.. ⊟ SOME-TEST ⊟1 ☒1 ·2
..
==> #<TRIAL (SOME-TEST) ABORT* 0.500s ⊟1 ☒1 ·2>
```

Now, we decide that nesting of test is unimportant here, change to new printer settings and replay:

```
(let ((*print-backtrace* nil)
      (*print-parent* nil)
      (*print-compactly* t)
      (*defer-describe* t))
  (replay-events !))
.. ☒⊟
.. ⊟ SOME-TEST ⊟1 ☒1 ·2
..
.. ;; UNEXPECTED-RESULT-FAILURE (☒) in SOME-TEST INNER:
.. (IS (= 10 7))
..
.. ;; UNHANDLED-ERROR (⊟) in SOME-TEST:
.. "my-msg" (SIMPLE-ERROR)
..
==> #<TRIAL (SOME-TEST) ABORT* 0.500s ⊟1 ☒1 ·2>
```

8 Implementation Notes

Try is supported on ABCL, AllegroCL, CLISP, CCL, CMUCL, ECL and SBCL.

- Pretty printing is non-existent on CLISP and broken on ABCL. The output may look garbled

on them.

- Gray streams are broken on ABCL so the output may look even worse https://abcl.org/trac/ticket/373.
- ABCL, CMUCL and ECL have a bug related to losing `eql`ness of source literals https://gitlab.com/embeddable-common-lisp/ecl/-/issues/665. The result is somewhat cosmetic; it may cause multiple captures being made for the same thing.

9 Glossary

- **[glossary-term] funcallable instance**

  This is a term from the MOP. A funcallable instance is an instance of a class that's a subclass of `mop:funcallable-standard-class`. It is like a normal instance, but it can also be `funcall`ed.

- **[glossary-term] cancelled non-local exit**

  This is a term from the Common Lisp ANSI standard. If during the unwinding of the stack initiated by a non-local exit another nlx is initiated in, and exits from an `unwind-protect` cleanup form, then this second nlx is said to have cancelled the first, and the first nlx will not continue.

```
(catch 'foo
  (catch 'bar
    (unwind-protect
         (throw 'foo 'foo)
      (throw 'bar 'bar))))
=> BAR
```

10 Indices

Referrer definition type abbreviations:

- *f*: for definitions in the function namespace (macros, compiler macros and also methods)
- *t*: `DEFTYPE`s, classes, conditions, structs
- *d*: documentation sections and glossary terms
- *l*: definitions of definition types
- *s*: ASDF systems
- *p*: packages
- *n*: named readtables
- *v*: special variables and constants
- *r*: restarts
- *?*: other

10.1 Function and Macro Index

10.2 Variable and Constant Index

10.3 Type Index

10.4 Misc Index

10.5 Concept Index