JOURNAL MANUAL

Contents

[in package JOURNAL with nicknames JRN]

- **[system]** **"journal"**

```
- _Version:_ 0.1.0
- _Description:_ A library built around explicit execution traces for
    logging, tracing, testing and persistence.
- _Licence:_ MIT, see COPYING.
- _Author:_ Gábor Melis <mega@retes.hu>
- _Homepage:_ <http://github.com/melisgl/journal>
- _Bug tracker:_ <http://github.com/melisgl/journal/issues>
```

```
- _Source control:_ [GIT](https://github.com/melisgl/journal.git)
- *Depends on:* alexandria, bordeaux-threads, local-time, [mgl-pax][6fdb],
↪  osicat(?), sb-posix(?), trivial-features, trivial-garbage
```

1 Links

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

2 Portability

Tested and supported on ABCL, CCL, CMUCL, ECL, and SBCL. AllegroCL Express edition runs out of heap while running the tests. On Lisps that seem to lack support for disabling and enabling of interrupts, such as ABCL, durability is compromised, and any attempt to `sync-journal` (see Synchronization strategies and Safety) will be a runtime error.

Journal depends on BORDEAUX-THREADS. Consequently, it does not load on implementations without real thread such as CLISP.

3 Background

Logging, tracing, testing, and persistence are about what happened during code execution. Recording machine-readable logs and traces can be repurposed for white-box testing. More, when the code is rerun, selected frames may return their recorded values without executing the code, which could serve as a mock object framework for writing tests. This ability to isolate external interactions and to reexecute traces is sufficient to reconstruct the state of a program, achieving simple persistence not unlike a journaling filesystem or event sourcing.

Journal is the library to log, trace, test and persist. It has a single macro at its heart: `journaled`, which does pretty much what was described. It can be thought of as generating two events around its body: one that records the name and an argument list (as in a function call), and another that records the return values. In Lisp-like pseudocode:

```
(defmacro journaled (name args &body body)
  `(progn
     (record-event `(:in ,name :args ,args))
     (let ((,return-values (multiple-value-list (progn ,@body))))
       (record-event `(:out ,name :values ,return-values))
       (values-list ,return-values))))
```

This is basically how recording works. When replaying events from a previous run, the return values of `body` can be checked against the recorded ones, or we may return the recorded values without even running `body`.

In summary, we can produce selective execution traces by wrapping code in `journaled` and use those traces for various purposes. The Journal library is this idea taken to its logical conclusion.

4 Distinguishing features

As a logging facility

- Nested contexts and single messages
- Customizable content and format
- Human- or machine-readable output

```
#68200.234: ("some-context")
#68200.234:   Informative log message
#68200.250: => NIL
```

See Logging for a complete example.

Compared to `cl:trace(0 1)`

- Ability to handle non-local exits
- Customizable content and format
- Optional timestamps, internal real- and run-time

```
(FOO 2.1)
  (1+ 2.1)
  => 3.1
=E "SIMPLE-ERROR" "The assertion (INTEGERP 3.1) failed."
```

See Tracing for a complete example.

As a test framework

- White-box testing based on execution traces
- Isolation of external dependencies
- Record-and-replay testing

```
(define-file-bundle-test (test-user-registration :directory "registration")
  (let ((username (replayed ("ask-username")
                    (format t "Please type your username: ")
                    (read-line))))
    (add-user username)
    (assert (user-exists-p username))))
```

See Testing for a complete example.

As a solution for persistence

- Event Sourcing: replay interactions with the external world
- Unchanged control flow
- Easy to implement history, undo

```
(defun my-resumable-autosaving-game-with-history ()
  (with-bundle (bundle)
    (play-guess-my-number)))
```

See Persistence for a complete example.

5 Basics

The journaled macro does both recording and replaying of events, possibly at the same time. Recording is easy: events generated by journaled are simply written to a journal, which is a sequence of events much like a file. What events are generated is described in journaled. Replay is much more involved, thus it gets its own section. The journals used for recording and replaying are specified by with-journaling or by with-bundle.

The Journals reference is presented later, but for most purposes, creating them (e.g. with make-in-memory-journal, make-file-journal) and maybe querying their contents with list-events will suffice. Some common cases of journal creation are handled by the convenience function to-journal.

Built on top of journals, Bundles juggle repeated replay-and-record cycles focussing on persistence.

- **[generic-function]** **to-journal** *designator*

  Return the journal designated by designator or signal an error. The default implementation

  - returns designator itself if it is of type journal,
  - returns a new in-memory-journal if designator is t,
  - returns a new file-journal if designator is a pathname(0 1).

- **[macro]** **with-journaling** *(&key record replay replay-eoj-error-p) &body body*

  Turn recording and/or replaying of events on or off for the duration of body. Both record and replay should be a journal designator (in the sense of to-journal) or nil.

  If record designates a journal, then events generated by enclosed journaled blocks are written to that journal (with exceptions, see the log-record argument of journaled). If replay designates a journal, then the generated events are matched against events from that journal according to the rules of Replay.

  A journal-error is signalled if record is a journal that has been previously recorded to by another with-journaling (that is, if its journal-state is not :new) or if replay is a journal that is not a complete recording of successful replay (i.e. its journal-state is not :completed). These checks are intended to catch mistakes that would render the new or existing records unusable for replay. When with-journaling finishes, the record journal is marked :completed or :failed in its journal-state.

  replay-eoj-error-p controls whether end-of-journal is signalled when a new event is being matched to the replay journal from which there are no more events to read. If

there was a `journaling-failure` or a `replay-failure` during execution, then `end-of-journal` is not signalled.

If `body` completes successfully, but `replay` has unprocessed events, then `replay-incomplete` is signalled.

`with-journaling` for different `record` journals can be nested and run independently.

- **[glossary-term] block**

  A journaled block, or simply block, is a number of forms wrapped in `journaled`. When a block is executed, a frame is created.

- **[glossary-term] frame**

  A frame is an `in-event`, `out-event` pair, which are created when a block is entered and left, respectively.

- **[function] record-journal**

  Return the `journal` in which events are currently being recorded (see `with-journaling` and `with-bundle`) or `nil`.

- **[function] replay-journal**

  Return the `journal` from which events are currently being replayed (see `with-journaling` and `with-bundle`) or `nil`.

- **[macro] journaled** *(name &key (log-record :record) version args values condition insertable replay-values replay-condition) &body body*

  `journaled` generates events upon entering and leaving the dynamic extent of `body` (also known as the journaled block), which we call the In-events and Out-events. Between generating the two events, `body` is typically executed normally (except for Replaying the outcome).

  Where the generated events are written is determined by the `:record` argument of the enclosing `with-journaling`. If there is no enclosing `with-journaling` and `log-record` is `nil`, then event recording is turned off and `journaled` imposes minimal overhead.

  - `name` can be of any type except `null`, not evaluated. For names, and for anything that gets written to a journal, a non-keyword symbol is a reasonable choice as it can be easily made unique. However, it also exposes the package structure, which might make reading stuff back more difficult. Keywords and strings do not have this problem.
  - `args` can be of any type, but is typically a list.

  Also see `:log-record` in the Logging section. For a description of `version`, `insertable`, `replay-values` and `replay-condition`, see Journaled for replay.

5.1 In-events

Upon entering a block, `journaled` generates an `in-event`, which conceptually opens a new frame. These in-events are created from the `name`, `version` and `args` arguments of `journaled`. For example,

```
(journaled (name :version version :args args) ...)
```

creates an event like this:

```
`(:in ,name :version ,version :args ,args)
```

where `:version` and `:args` may be omitted if they are `nil`. Versions are used for Replay.

5.2 Out-events

Upon leaving a block, `journaled` generates an `out-event`, closing the frame opened by the corresponding `in-event`. These out-events are property lists like this:

```
(:out foo :version 1 :values (42))
```

Their `name` and `version` (`foo` and `1` in the example) are the same as in the in-event: they come from the corresponding arguments of `journaled`. `exit` and `outcome` are filled in differently depending on how the block finished its execution.

- **[type] event-exit**

  One of `:values`, `:condition`, `:error` and `:nlx`. Indicates whether a journaled block

  - returned normally (`:values`, see values outcome),
  - unwound on an expected condition (`:condition`, see condition outcome),
  - unwound on an unexpected condition (`:error`, see error outcome),
  - unwound by performing a non-local exit of some other kind such as a throw (`:nlx`, see nlx outcome).

  The first two are expected outcomes, while the latter two are unexpected outcomes.

- **[glossary-term] values outcome**

  If the `journaled` block returns normally, `event-exit` is `:values`, and the outcome is the list of values returned:

  ```
  (journaled (foo) (values 7 t))
  ;; generates the out-event
  (:out foo :values (7 t))
  ```

  The list of return values of the block is transformed by the `values` argument of `journaled`, whose default is `#'identity`. Also see Working with unreadable values).

- **[glossary-term] condition outcome**

If the block unwound due to a condition, and `journaled`'s `condition` argument (a function whose default is `(constantly nil)`) returns non-`nil` when invoked on it, then `event-exit` is `:condition`, and the outcome is this return value:

```
(journaled (foo :condition (lambda (c) (prin1-to-string c)))
  (error "xxx"))
;; generates the out-event
(:out foo :condition "xxx")
```

Conditions thus recognized are those that can be considered part of normal execution. Just like return values, these expected conditions may be required to match what's in the replay journal. Furthermore, given a suitable `replay-condition` in `journaled`, they may be replayed without running the block.

- **[glossary-term] error outcome**

  If the `journaled` block unwound due to a condition, but `journaled`'s `condition` argument returns `nil` when invoked on it, then `event-exit` is `:error` and the outcome the string representations of the type of the condition and the condition itself.

```
(journaled (foo)
  (error "xxx"))
;; generates this out-event:
;; (:out foo :error ("simple-error" "xxx"))
```

  The conversion to string is performed with `princ` in `with-standard-io-syntax`. This scheme is intended to avoid leaking random implementation details into the journal, which would make `read`ing it back difficult.

  In contrast with condition outcomes, error outcomes are what the code is not prepared to handle or replay in a meaningful way.

- **[glossary-term] nlx outcome**

  If the `journaled` block performed a non-local exit that was not due to a condition, then `event-exit` is `:nlx` and the outcome is `nil`.

```
(catch 'xxx
  (journaled (foo)
    (throw 'xxx nil)))
;; generates the out-event
(:out foo :nlx nil)
```

  Note that condition outcomes and error outcomes are also due to non-local exits but are distinct from nlx outcomes.

  Currently, nlx outcomes are detected rather heuristically as there is no portable way to detect what really caused the unwinding of the stack.

There is a further grouping of outcomes into expected and unexpected.

- **[glossary-term] expected outcome**

An `out-event` is said to have an expected outcome if it had a values outcome or a condition outcome, or equivalently, when its `event-exit` is `:values` or `:condition`.

- **[glossary-term] unexpected outcome**

  An `out-event` is said to have an unexpected outcome if it had an error outcome or an nlx outcome, or equivalently, when its `event-exit` is `:error` or `:nlx`.

5.3 Working with unreadable values

The events recorded often need to be readable. This is always required with `file-journal`s, often with `in-memory-journal`s, but never with `pprint-journal`s. By choosing an appropriate identifier or string representation of the unreadable object to journal, this is not a problem in practice. `journaled` provides the `values` hook for this purpose.

With `external-event`s, whose outcome is replayed (see Replaying the outcome), we also need to be able to reverse the transformation of `values`, and this is what the `replay-values` argument of `journaled` is for.

Let's see a complete example.

```
(defclass user ()
  ((id :initarg :id :reader user-id)))

(defmethod print-object ((user user) stream)
  (print-unreadable-object (user stream :type t)
    (format stream "~S" (slot-value user 'id))))

(defvar *users* (make-hash-table))

(defun find-user (id)
  (gethash id *users*))

(defun add-user (id)
  (setf (gethash id *users*) (make-instance 'user :id id)))

(defvar *user7* (add-user 7))

(defun get-message ()
  (replayed (listen :values (values-> #'user-id)
                    :replay-values (values<- #'find-user))
    (values *user7* "hello")))

(jtrace user-id find-user get-message)

(let ((bundle (make-file-bundle "/tmp/user-example/")))
  (format t "Recording")
  (with-bundle (bundle)
    (get-message))
  (format t "~%Replaying")
  (with-bundle (bundle)
    (get-message)))
.. Recording
```

```
.. (GET-MESSAGE)
..   (USER-ID #<USER 7>)
..   => 7
.. => #<USER 7>, "hello"
.. Replaying
.. (GET-MESSAGE)
..   (FIND-USER 7)
..   => #<USER 7>, T
.. => #<USER 7>, "hello"
==> #<USER 7>
=> "hello"
```

To be able to journal the return values of `get-message`, the `user` object must be transformed to something readable. On the `Recording` run, `(values-> #'user-id)` replaces the user object with its id in the `event-outcome` recorded, but the original user object is returned.

When `Replaying`, the journaled `out-event` is replayed (see Replaying the outcome):

```
(:OUT GET-MESSAGE :VERSION :INFINITY :VALUES (7 "hello"))
```

The user object is looked up according to `:replay-values` and is returned along with `"hello"`.

- **[function]** **values->** *&rest fns*

  A utility to create a function suitable as the `values` argument of `journaled`. The `values` function is called with the list of values returned by the block and returns a transformed set of values that may be recorded in a journal. While arbitrary transformations are allowed, `values->` handles the common case of transforming individual elements of the list independently by calling the functions in FN with the values of the list of the same position.

  ```
  (funcall (values-> #'1+) '(7 :something))
  => (8 :SOMETHING)
  ```

  Note how `#'1+` is applied only to the first element of the values list. The list of functions is shorter than the values list, so `:something` is not transformed. A value can be left explicitly untransformed by specifying `#'identity` or `nil` as the function:

  ```
  (funcall (values-> #'1+ nil #'symbol-name)
           '(7 :something :another))
  => (8 :SOMETHING "ANOTHER")
  ```

- **[function]** **values<-** *&rest fns*

  The inverse of `values->`, this returns a function suitable as the `replay-values` argument of `journaled`. It does pretty much what `values->` does, but the function returned returns the transformed list as multiple values instead of as a list.

  ```
  (funcall (values<- #'1-) '(8 :something))
  => 7
  => :SOMETHING
  ```

5.4 Utilities

- **[function]** **list-events** *&optional (journal (record-journal))*

  Return a list of all the events in the journal designated by `journal`. Calls `sync-journal` first to make sure that all writes are taken into account.

- **[function]** **events-to-frames** *events*

  Convert a flat list of events, such as those returned by `list-events`, to a nested list representing the frames. Each frame is a list of the form `(<in-event> <nested-frames>* <out-event>?)`. Like in `print-events`, `events` may be a `journal`.

```
(events-to-frames '((:in foo :args (1 2))
                    (:in bar :args (7))
                    (:leaf "leaf")
                    (:out bar :values (8))
                    (:out foo :values (2))
                    (:in foo :args (3 4))
                    (:in bar :args (8))))
=> (((:IN FOO :ARGS (1 2))
     ((:IN BAR :ARGS (7))
      (:LEAF "leaf")
      (:OUT BAR :VALUES (8)))
     (:OUT FOO :VALUES (2)))
    ((:IN FOO :ARGS (3 4)) ((:IN BAR :ARGS (8)))))
```

  Note that, as in the above example, incomplete frames (those without an `out-event`) are included in the output.

- **[function]** **expected-type** *type*

  Return a function suitable as the `condition` argument of `journaled`, which returns the type of its single argument as a string if it is of `type`, else `nil`.

5.5 Pretty-printing

- **[function]** **print-events** *events &key stream*

  Print `events` to `stream` as lists, starting a new line for each event and indenting them according to their nesting structure. `events` may be a sequence or a `journal`, in which case `list-events` is called on it first.

```
(print-events '((:in log :args ("first arg" 2))
                (:in versioned :version 1 :args (3))
                (:out versioned :version 1 :values (42 t))
                (:out log :condition "a :CONDITION outcome")
                (:in log-2)
                (:out log-2 :nlx nil)
                (:in external :version :infinity)
                (:out external :version :infinity
                 :error ("ERROR" "an :ERROR outcome"))))
..
```

```
.. (:IN LOG :ARGS ("first arg" 2))
..   (:IN VERSIONED :VERSION 1 :ARGS (3))
..   (:OUT VERSIONED :VERSION 1 :VALUES (42 T))
.. (:OUT LOG :CONDITION "a :CONDITION outcome")
.. (:IN LOG-2)
.. (:OUT LOG-2 :NLX NIL)
.. (:IN EXTERNAL :VERSION :INFINITY)
.. (:OUT EXTERNAL :VERSION :INFINITY :ERROR ("ERROR" "an :ERROR outcome"))
=> ; No value
```

- **[function]** **pprint-events** *events &key stream (prettifier 'prettify-event)*

  Like `print-events`, but produces terser, more human readable output.

```
(pprint-events '((:in log :args ("first arg" 2))
                 (:in versioned :version 1 :args (3))
                 (:leaf "This is a leaf, not a frame.")
                 (:out versioned :version 1 :values (42 t))
                 (:out log :condition "a :CONDITION outcome")
                 (:in log-2)
                 (:out log-2 :nlx nil)
                 (:in external :version :infinity)
                 (:out external :version :infinity
                  :error ("ERROR" "an :ERROR outcome"))))
..
.. (LOG "first arg" 2)
..   (VERSIONED 3) v1
..     This is a leaf, not a frame.
..   => 42, T
.. =C "a :CONDITION outcome"
.. (LOG-2)
.. =X
.. (EXTERNAL) ext
.. =E "ERROR" "an :ERROR outcome"
=> ; No value
```

  The function given as the `prettifier` argument formats individual events. The above output was produced with `prettify-event`. For a description of `prettifier`'s arguments see `prettify-event`.

- **[function]** **prettify-event** *event depth stream*

  Write `event` to `stream` in a somewhat human-friendly format. This is the function `pprint-journal`, `pprint-events`, and Tracing use by default. In addition to the basic example in `pprint-events`, decoration on events is printed before normal, indented output like this:

```
(pprint-events '((:leaf "About to sleep" :time "19:57:00" :function "FOO")))
..
.. 19:57:00 FOO: About to sleep
```

  `depth` is the nesting level of the `event`. Top-level events have depth

0. `prettify-event` prints indents the output after printing the decorations by 2 spaces per depth.

Instead of collecting events and then printing them, events can be pretty-printed to a stream as they generated. This is accomplished with Pretty-printing journals, discussed in detail later, in the following way:

```
(let ((journal (make-pprint-journal)))
  (with-journaling (:record journal)
    (journaled (foo) "Hello")))
..
.. (FOO)
.. => "Hello"
```

Note that Pretty-printing journals are not tied to `with-journaling` and are most often used for Logging and Tracing.

5.6 Error handling

- **[condition]** **journaling-failure** *serious-condition*

  Signalled during the dynamic extent of `with-journaling` when an error threatens to leave the journaling mechanism in an inconsistent state. These include I/O errors encountered reading or writing journals by `with-journaling`, `journaled`, `logged`, `with-replay-filter`, `sync-journal`, and also `storage-condition`s, assertion failures, errors calling `journaled`'s `values` and `condition` function arguments. Crucially, this does not apply to non-local exits from other code, such as `journaled` blocks, whose error handling is largely unaltered (see Out-events and Replay failures).

  In general, any non-local exit from critical parts of the code is turned into a `journaling-failure` to protect the integrity of the `record-journal`. The condition that caused the unwinding is in `journaling-failure-embedded-condition`, or `nil` if it was a pure non-local exit like `throw`. This is a `serious-condition`, not to be handled within `with-journaling`.

  After a `journaling-failure`, the journaling mechanism cannot be trusted anymore. The `replay-journal` might have failed a read and be out-of-sync. The `record-journal` may have missing events (or even half-written events with `file-journal`s without `sync`, see Synchronization strategies), and further writes to it would risk replayability, which is equivalent to database corruption. Thus, upon signalling `journaling-failure`, `journal-state` is set to

  - `:completed` if the journal is in state `:recording` or `:logging` and the transition to `:recording` was reflected in storage,
  - else it is set to `:failed`.

  After a `journaling-failure`, any further attempt within the affected `with-journaling` to use the critical machinery mentioned above (`journaled`, `logged`, etc) resignals the same journal failure condition. As a consequence, the record journal cannot be changed,

and the only way to recover is to leave `with-journaling`. This does not affect processing in other threads, which by design cannot write to the record journal.

Note that in contrast with `journaling-failure` and `replay-failure`, which necessitate leaving `with-journaling` to recover from, the other conditions – `journal-error`, and `streamlet-error` – are subclasses of `error` as their handling need not be so heavy-handed.

- **[reader]** **journaling-failure-embedded-condition** *journaling-failure* *(:embedded-condition)*
- **[condition]** **record-unexpected-outcome**

  Signalled (with `signal`: this is not an `error`) by `journaled` when a `versioned-event` or an `external-event` had an UNEXPECTED-OUTCOME while in `journal-state` `:recording`. Upon signalling this condition, `journal-state` is set to `:logging`, thus no more events can be recorded that will affect replay of the journal being recorded. The event that triggered this condition is recorded in state `:logging`, with its version downgraded. Since Replay (except invoked) is built on the assumption that control flow is deterministic, an unexpected outcome is significant because it makes this assumption to hold unlikely.

  Also see `replay-unexpected-outcome`.

- **[condition]** **data-event-lossage** *journaling-failure*

  Signalled when a data event is about to be recorded in `journal-state` `:mismatched` or `:logging`. Since the data event will not be replayed that constitutes data loss.

- **[condition]** **journal-error** *error*

  Signalled by `with-journaling`, `with-bundle` and by `:log-record`. It is also signalled by the low-level streamlet interface (see Streamlets reference).

- **[condition]** **end-of-journal** *journal-error*

  This might be signalled by the replay mechanism if `with-journaling`'s `replay-eoj-error-p` is true. Unlike `replay-failure`s, this does not affect `journal-state` of `record-journal`. At a lower level, it is signalled by `read-event` upon reading past the end of the `journal` if `eoj-error-p`.

6 Logging

Before we get into the details, here is a self-contained example that demonstrates typical use.

```
(defvar *communication-log* nil)
(defvar *logic-log* nil)
(defvar *logic-log-level* 0)

(defun call-with-connection (port fn)
  (framed (call-with-connection :log-record *communication-log*
                                :args `(,port))
    (funcall fn)))
```

```
(defun fetch-data (key)
  (let ((value 42))
    (logged ((and (<= 1 *logic-log-level*) *logic-log*))
      "The value of ~S is ~S." key value)
    value))

(defun init-logging (&key (logic-log-level 1))
  (let* ((stream (open "/tmp/xxx.log"
                       :direction :output
                       :if-does-not-exist :create
                       :if-exists :append))
         (journal (make-pprint-journal
                   :stream (make-broadcast-stream
                            (make-synonym-stream '*standard-output*)
                            stream))))
    (setq *communication-log* journal)
    (setq *logic-log* journal)
    (setq *logic-log-level* logic-log-level)))

(init-logging)

(call-with-connection 8080 (lambda () (fetch-data :foo)))
..
.. (CALL-WITH-CONNECTION 8080)
..   The value of :FOO is 42.
.. => 42
=> 42

(setq *logic-log-level* 0)
(call-with-connection 8080 (lambda () (fetch-data :foo)))
..
.. (CALL-WITH-CONNECTION 8080)
.. => 42
=> 42

(ignore-errors
  (call-with-connection 8080 (lambda () (error "Something unexpected."))))
..
.. (CALL-WITH-CONNECTION 8080)
.. =E "SIMPLE-ERROR" "Something unexpected."
```

Default to muffling Imagine a utility library called glib.

```
(defvar *glib-log* nil)
(defvar *patience* 1)

(defun sl33p (seconds)
  (logged (*glib-log*) "Sleeping for ~As." seconds)
  (sleep (* *patience* seconds)))
```

Glib follows the recommendation to have a special variable globally bound to `nil` by default.

The value of `*glib-log*` is the journal to which glib log messages will be routed. Since it's `nil`, the log messages are muffled, and to record any log message, we need to change its value.

Routing logs to a journal Let's send the logs to a `pprint-journal`:

```
(setq *glib-log* (make-pprint-journal
                  :log-decorator (make-log-decorator :time t)))
(sl33p 0.01)
..
.. 2020-08-31T12:45:23.827172+02:00: Sleeping for 0.01s.
```

That's a bit too wordy. For this tutorial, let's stick to less verbose output:

```
(setq *glib-log* (make-pprint-journal))
(sl33p 0.01)
..
.. Sleeping for 0.01s.
```

To log to a file:

```
(setq *glib-log* (make-pprint-journal
                  :stream (open "/tmp/glib.log"
                                :direction :output
                                :if-does-not-exist :create
                                :if-exists :append)))
```

Capturing logs in `with-journaling`'s `record-journal` If we were recording a journal for replay and wanted to include glib logs in the journal, we would do something like this:

```
(with-journaling (:record t)
  (let ((*glib-log* :record))
    (sl33p 0.01)
    (journaled (non-glib-stuff :version 1)))
  (list-events))
=> ((:LEAF "Sleeping for 0.01s.")
    (:IN NON-GLIB-STUFF :VERSION 1)
    (:OUT NON-GLIB-STUFF :VERSION 1 :VALUES (NIL)))
```

We could even `(setq *glib-log* :record)` to make it so that glib messages are included by default in the `record-journal`. In this example, the special `*glib-log*` acts like a log category for all the log messages of the glib library (currently one).

Rerouting a category Next, we route `*glib-log*` to wherever `*app-log*` is pointing by binding `*glib-log*` *to the symbol* `*app-log*` (see `:log-record`).

```
(defvar *app-log* nil)

(let ((*glib-log* '*app-log*))
  (setq *app-log* nil)
  (logged (*glib-log*) "This is not written anywhere.")
  (setq *app-log* (make-pprint-journal :pretty nil))
```

```
  (sl33p 0.01))
..
.. (:LEAF "Sleeping for 0.01s.")
```

Note how pretty-printing was turned off, and we see the `leaf-event` generated by `logged` in its raw plist form.

Conditional routing Finally, to make routing decisions conditional we need to change `sl33p`:

```
(defvar *glib-log-level* 1)

(defun sl33p (seconds)
  (logged ((and (<= 2 *glib-log-level*) *glib-log*))
          "Sleeping for ~As." (* *patience* seconds))
  (sleep seconds))

;;; Check that it all works:
(let ((*glib-log-level* 1)
      (*glib-log* (make-pprint-journal)))
  (format t "~%With log-level ~A" *glib-log-level*)
  (sl33p 0.01)
  (setq *glib-log-level* 2)
  (format t "~%With log-level ~A" *glib-log-level*)
  (sl33p 0.01))
..
.. With log-level 1
.. With log-level 2
.. Sleeping for 0.01s.
```

Nested log contexts `logged` is for single messages. `journaled`, or in this example `framed`, can provide nested context:

```
(defun callv (var value symbol &rest args)
  "Call SYMBOL-FUNCTION of SYMBOL with VAR dynamically bound to VALUE."
  (framed ("glib:callv" :log-record *glib-log*
                        :args `(,var ,value ,symbol ,@args))
    (progv (list var) (list value)
      (apply (symbol-function symbol) args))))

(callv '*print-base* 2 'print 10)
..
.. ("glib:callv" *PRINT-BASE* 2 PRINT 10)
.. 1010
.. => 10
=> 10

(let ((*glib-log-level* 2))
  (callv '*patience* 7 'sl33p 0.01))
..
.. ("glib:callv" *PATIENCE* 7 SL33P 0.01)
```

```
..   Sleeping for 0.07s.
.. => NIL
```

6.1 Customizing logs

Customizing the output format is possible if we don't necessarily expect to be able to read the logs back programmatically. There is an example in Tracing, which is built on Pretty-printing journals.

Here, we discuss how to make logs more informative.

- **[glossary-term]** **decoration**

  `journal-log-decorator` adds additional data to `log-event`s as they are written to the journal. This data is called decoration, and it is to capture the context in which the event was triggered. See `make-log-decorator` for a typical example. Decorations, since they can be on `log-event`s only, do not affect Replay. Decorations are most often used with Pretty-printing.

- **[accessor]** **journal-log-decorator** *journal* *(:log-decorator = nil)*

  If non-`nil`, this is a function to add decoration to `log-event`s before they are written to a journal. The only allowed transformation is to *append* a plist to the event, which is a plist itself. The keys can be anything.

- **[function]** **make-log-decorator** *&key time real-time run-time thread depth out-name*

  Return a function suitable as `journal-log-decorator` that may add a string timestamp, the internal real-time or run-time (both in seconds), the name of the thread, to events, which will be handled by `prettify-event`. If `depth`, then `prettify-event` will print the nesting level of the event being printed. If `out-name`, the `prettify-event` will print the name of Out-events.

  All arguments are boolean-valued symbols.

```
(funcall (make-log-decorator :depth t :out-name t :thread t
                             :time t :real-time t :run-time t)
         (make-leaf-event :foo))
=> (:LEAF :FOO :DEPTH T :OUT-NAME T :THREAD "worker"
               :TIME "2023-05-26T12:27:44.172614+01:00"
               :REAL-TIME 2531.3254 :RUN-TIME 28.972797)
```

6.2 `:log-record`

`with-journaling` and `with-bundle` control replaying and recording within their dynamic extent, which is rather a necessity because Replay needs to read the events in the same order as the `journaled` blocks are being executed. However, `log-event`s do not affect replay, so we can allow more flexibility in routing them.

The `log-record` argument of `journaled` and `logged` controls where `log-event`s are written both within `with-journaling` and without. The algorithm to determine the target journal is

this:

1. If `log-record` is `:record`, then the `record-journal` is returned.
2. If `log-record` is `nil`, then it is returned.
3. If `log-record` is a `journal`, then it is returned.
4. If `log-record` is a symbol (other than `nil`), then the `symbol-value` of that symbol is assigned to `log-record`, and we go to step 1.

If the return value is `nil`, then the event will not be written anywhere, else it is written to the journal returned.

This is reminiscent of `synonym-stream`s, also in that it is possible end up in cycles in the resolution. For this reason, the algorithm stop with a `journal-error` after 100 iterations.

Interactions Events may be written to `log-record` even without an enclosing `with-journaling`, and it does not affect the `journal-state`. However, it is a `journal-error` to write to a `:completed` journal (see `journal-state`).

When multiple threads log to the same journal, it is guaranteed that individual events are written atomically, but frames from different threads do not necessarily nest. To keep the log informative, the name of thread may be added to the events as decoration.

Also, see notes on thread Safety.

6.3 Logging with `leaf-events`

- **[macro]** **logged** *(&optional (log-record :record)) format-control &rest format-args*

  `logged` creates a single `leaf-event`, whose name is the string constructed by `format`. For example:

```
(with-journaling (:record t)
  (logged () "Hello, ~A." "world")
  (list-events))
=> ((:LEAF "Hello, world."))
```

  `leaf-events` are `log-events` with no separate in- and out-events. They have an `event-name` and no other properties. Use `logged` for point-in-time textual log messages, and `journaled` with `version` `nil` (i.e. `framed`) to provide context.

  Also, see `:log-record`.

7 Tracing

`jtrace` behaves similarly to `cl:trace(0 1)` but deals with non-local exits gracefully.

```
(defun foo (x)
  (sleep 0.12)
  (1+ x))

(defun bar (x)
  (foo (+ x 2))
  (error "xxx"))

(jtrace foo bar)

(ignore-errors (bar 1))
..
.. 0: (BAR 1)
..   1: (FOO 3)
..   1: FOO => 4
.. 0: BAR =E "SIMPLE-ERROR" "xxx"
```

Basic tracing

Log-like output It can also include the name of the originating thread and timestamps in the output:

```
(let ((*trace-thread* t)
      (*trace-time* t)
      (*trace-depth* nil)
      (*trace-out-name* nil))
  (ignore-errors (bar 1)))
..
.. 2020-09-02T19:58:19.415204+02:00 worker: (BAR 1)
.. 2020-09-02T19:58:19.415547+02:00 worker:   (FOO 3)
.. 2020-09-02T19:58:19.535766+02:00 worker:   => 4
.. 2020-09-02T19:58:19.535908+02:00 worker: =E "SIMPLE-ERROR" "xxx"
```

```
(let ((*trace-real-time* t)
      (*trace-run-time* t)
      (*trace-depth* nil)
      (*trace-out-name* nil))
  (ignore-errors (bar 1)))
..
.. #16735.736 !68.368: (BAR 1)
.. #16735.736 !68.369:   (FOO 3)
.. #16735.857 !68.369:   => 4
.. #16735.857 !68.369: =E "SIMPLE-ERROR" "xxx"
```

Profiler-like output

Customizing the content and the format If these options are insufficient, the content and the format of the trace can be customized:

```
(let ((*trace-journal*
       (make-pprint-journal :pretty '*trace-pretty*
                            :prettifier (lambda (event depth stream)
                                          (format stream "~%Depth: ~A, event: ~S"
                                                  depth event))
                            :stream (make-synonym-stream '*error-output*)
                            :log-decorator (lambda (event)
                                             (append event '(:custom 7))))))
  (ignore-errors (bar 1)))
..
.. Depth: 0, event: (:IN BAR :ARGS (1) :CUSTOM 7)
.. Depth: 1, event: (:IN FOO :ARGS (3) :CUSTOM 7)
.. Depth: 1, event: (:OUT FOO :VALUES (4) :CUSTOM 7)
.. Depth: 0, event: (:OUT BAR :ERROR ("SIMPLE-ERROR" "xxx") :CUSTOM 7)
```

In the above, `*trace-journal*` was bound locally to keep the example from wrecking the global default, but the same effect could be achieved by `setf`ing `pprint-journal-prettifier`, `pprint-journal-stream` and `journal-log-decorator`.

- **[macro]** **jtrace** *&rest names*

  Like `cl:trace`(`0` `1`), `jtrace` takes a list of symbols. When functions denoted by those `names` are invoked, their names, arguments and outcomes are printed in human readable form to `*trace-output*`. These values may not be readable, `jtrace` does not care.

  The format of the output is the same as that of `pprint-events`. Behind the scenes, `jtrace` encapsulates the global functions with `names` in wrapper that behaves as if `foo` in the example above was defined like this:

  ```
  (defun foo (x)
    (framed (foo :args `(,x) :log-record *trace-journal*)
      (1+ x)))
  ```

  If `jtrace` is invoked with no arguments, it returns the list of symbols currently traced.

  On Lisps other than SBCL, where a function encapsulation facility is not available or it is not used by Journal, `jtrace` simply sets `symbol-function`. This solution loses the tracing encapsulation when the function is recompiled. On these platforms, `(jtrace)` also retraces all functions that should be traced but aren't.

  The main advantage of `jtrace` over `cl:trace` is the ability to trace errors, not just normal return values. As it is built on `journaled`, it can also detect – somewhat heuristically – `throw`s and similar.

- **[macro]** **juntrace** *&rest names*

  Like `cl:untrace`(`0` `1`), `juntrace` makes it so that the global functions denoted by the symbols `names` are no longer traced by `jtrace`. When invoked with no arguments, it untraces all traced functions.

- **[variable]** ***trace-pretty*** *t*

If *trace-pretty* is true, then jtrace produces output like pprint-events, else it's like print-events.

- **[variable]** ***trace-depth*** *t*

  Controls whether to decorate the trace with the depth of event. See make-log-decorator.

- **[variable]** ***trace-out-name*** *t*

  Controls whether trace should print the event-name of Out-events, which is redundant with the event-name of the corresponding In-events. See make-log-decorator.

- **[variable]** ***trace-thread*** *nil*

  Controls whether to decorate the trace with the name of the originating thread. See make-log-decorator.

- **[variable]** ***trace-time*** *nil*

  Controls whether to decorate the trace with a timestamp. See make-log-decorator.

- **[variable]** ***trace-real-time*** *nil*

  Controls whether to decorate the trace with the internal real-time. See make-log-decorator.

- **[variable]** ***trace-run-time*** *nil*

  Controls whether to decorate the trace with the internal run-time. See make-log-decorator.

- **[variable]** ***trace-journal*** *#<pprint-journal :new 1>*

  The journal where jtrace writes log-events. By default, it is a pprint-journal that sets up a synonym-stream to *trace-output* and sends its output there. It pays attention to *trace-pretty*, and its log decorator is affected by *trace-time* and *trace-thread*. However, by changing journal-log-decorator and pprint-journal-prettifier, content and output can be customized.

7.1 Slime integration

Slime, by default, binds C-c C-t to toggling cl:trace(0 1). To integrate jtrace into Slime, load src/mgl-jrn.el into Emacs.

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

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

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

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

Alternatively, with use-package:

```
(use-package mgl-jrn :load-path "~/quicklisp/"
  :after slime
  :demand t)
```

Since `jtrace` lacks some features of `cl:trace`, most notably that of tracing non-global functions, it is assigned a separate binding, `C-c C-j`.

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

  Copy `mgl-jrn.el` distributed with this package to `target-dir`.

8 Replay

During replay, code is executed normally with special rules for blocks. There are two modes for dealing with blocks: replaying the code and replaying the outcome. When code is replayed, upon entering and leaving a block, the events generated are matched to events read from the journal being replayed. If the events don't match, `replay-failure` is signalled, which marks the record journal as having failed the replay. This is intended to make sure that the state of the program during the replay matches the state at the time of recording. In the other mode, when the outcome is replayed, a block may not be executed at all, but its recorded outcome is reproduced (i.e. the recorded return values are simply returned).

Replay can be only be initiated with `with-journaling` (or its close kin `with-bundle`). After the per-event processing described below, when `with-journaling` finishes, it might signal `replay-incomplete` if there are unprocessed non-log events left in the replay journal.

Replay is deemed successful or failed depending on whether all events are replayed from the replay journal without a `replay-failure`. A journal that records events from a successful replay can be used in place of the journal that was replayed, and so on. The logic of replacing journals with their successful replays is automated by Bundles. `with-journaling` does not allow replay from journals that were failed replays themselves. The mechanism, in terms of which tracking success and failure of replays is implemented, revolves around `journal-state` and `event-versions`, which we discuss next.

- **[type]** **journal-state**

  `journal`'s state with respect to replay is updated during `with-journaling`. The possible states are:

  - **`:new`**: This journal was just created but never recorded to.
  - **`:replaying`**: Replaying events has started, some events may have been replayed successfully, but there are more non-log events to replay.
  - **`:mismatched`**: There was a `replay-failure`. In this state, `versioned-events` generated are downgraded to `log-events`, `external-events` and invoked trigger `data-event-lossage`.
  - **`:recording`**: All events from the replay journal were successfully replayed, and now new events are being recorded without being matched to the replay journal.

- **`:logging`**: There was a `record-unexpected-outcome`. In this state, `versioned-events` generated are downgraded to `log-events`, `external-events` and invoked trigger `data-event-lossage`.
- **`:failed`**: The journal is to be discarded. It encountered a `journaling-failure` or a `replay-failure` without completing the replay and reaching `:recording`.
- **`:completed`**: All events were successfully replayed and `with-journaling` finished or a `journaling-failure` occurred while `:recording` or `:logging`.

The state transitions are:

```
:NEW                -> :REPLAYING  (on entering WITH-JOURNALING)
:REPLAYING          -> :MISMATCHED (on REPLAY-FAILURE)
:REPLAYING          -> :FAILED     (on REPLAY-INCOMPLETE)
:REPLAYING          -> :FAILED     (on JOURNALING-FAILURE)
:REPLAYING          -> :RECORDING  (on successfully replaying all events)
:MISMATCHED         -> :FAILED     (on leaving WITH-JOURNALING)
:RECORDING          -> :LOGGING    (on RECORD-UNEXPECTED-OUTCOME)
:RECORDING/:LOGGING -> :COMPLETED  (on leaving WITH-JOURNALING)
:RECORDING/:LOGGING -> :COMPLETED  (on JOURNALING-FAILURE)
```

`:new` is the starting state. It is a `journal-error` to attempt to write to journals in `:completed`. Note that once in `:recording`, the only possible terminal state is `:completed`.

8.1 Journaled for replay

The following arguments of `journaled` control behaviour under replay.

- `version`: see `event-version` below.
- `insertable` controls whether `versioned-events` and `external-events` may be replayed with the *insert* replay strategy (see The replay strategy). Does not affect `log-events`, which are always _insert_ed. Note that inserting `external-events` while `:replaying` is often not meaningful (e.g. asking the user for input may lead to a `replay-failure`). See `peek-replay-event` for an example on how to properly insert these kinds of `external-events`.
- `replay-values`, a function or `nil`, may be called with `event-outcome` when replaying and `:version :infinity`. `nil` is equivalent to `values-list`. See `values<-` for an example.
- `replay-condition`, a function or `nil`, may be called with `event-outcome` (the return value of the function provided as `:condition`) when replaying and `:version` is `:infinity`. `nil` is equivalent to the `error` function. Replaying conditions is cumbersome and best avoided.
- **[variable]** ***force-insertable*** *nil*

  The default value of the `insertable` argument of `journaled` for `versioned-events`. Binding this to `t` allows en-masse structural upgrades in combination with `with-replay-filter`. Does not affect `external-events`. See Upgrades and replay.
- **[type]** **event-version**

An event's version is either `nil`, a positive `fixnum`, or `:infinity`, which correspond to `log-event`s, `versioned-event`s, and `external-event`s, respectively, and have an increasingly strict behaviour with regards to Replay. All `event`s have versions. The versions of the in- and out-events belonging to the same frame are the same.

- **[type] log-event**

  Events with `event-version` `nil` called log events. During Replay, they are never matched to events from the replay journal, and log events in the replay do not affect events being recorded either. These properties allow log events to be recorded in arbitrary journals with `journaled`'s `log-record` argument. The convenience macro `framed` is creating frames of log-events, while the `logged` generates a log-event that's a `leaf-event`.

- **[type] versioned-event**

  Events with a positive integer `event-version` are called versioned events. In Replay, they undergo consistency checks unlike `log-event`s, but the rules for them are less strict than for `external-event`s. In particular, higher versions are always considered compatible with lower versions, they become an *upgrade* in terms of the The replay strategy, and versioned events can be inserted into the record without a corresponding replay event with `journaled`'s `insertable`.

  If a `versioned-event` has an unexpected outcome, `record-unexpected-outcome` is signalled.

- **[type] external-event**

  Events with `event-version` `:infinity` are called external events. They are like `versioned-event`s whose version was bumped all the way to infinity, which rules out easy, non-matching upgrades. Also, they are never inserted to the record without a matching replay event (see The replay strategy).

  In return for these restrictions, external events can be replayed without running the corresponding block (see Replaying the outcome). This allows their out-event variety, called data events, to be non-deterministic. Data events play a crucial role in Persistence.

  If an `external-event` has an unexpected outcome, `record-unexpected-outcome` is signalled.

Built on top of `journaled`, the macros below record a pair of In-events and Out-events but differ in how they are replayed and the requirements on their blocks. The following table names the type of `event` produced (`Event`), how In-events are replayed (`In-e.`), whether the block is always run (`Run`), how Out-events are replayed (`Out-e.`), whether the block must be deterministic (`Det`) or side-effect free (`sef`).

```
|          | Event     | In-e.  | Run | Out-e. | Det | SEF |
|----------+-----------+--------+-----+--------+-----+-----|
| FRAMED   | log       | skip   | y   | skip   | n   | n   |
| CHECKED  | versioned | match  | y   | match  | y   | n   |
| REPLAYED | external  | match  | n   | replay | n   | y   |
| INVOKED  | versioned | replay | y   | match  | y   | n   |
```

Note that the replay-replay combination is not implemented because there is nowhere to return values from replay-triggered functions.

- **[macro]** **framed** *(name &key log-record args values condition) &body body*

  A wrapper around `journaled` to produce frames of `log-event`s. That is, `version` is always `nil`, and some irrelevant arguments are omitted. The related `logged` creates a single `leaf-event`.

  With `framed`, `body` is always run and no `replay-failure`s are triggered. `body` is not required to be deterministic, and it may have side-effects.

- **[macro]** **checked** *(name &key (version 1) args values condition insertable) &body body*

  A wrapper around `journaled` to produce frames of `versioned-event`s. `version` defaults to 1. `checked` is for ensuring that supposedly deterministic processing does not veer off the replay.

  With `checked`, `body` – which must be deterministic – is always run and `replay-failure`s are triggered when the events generated do not match the events in the replay journal. `body` may have side-effects.

  For further discussion of determinism, see `replayed`.

- **[macro]** **replayed** *(name &key args values condition insertable replay-values replay-condition) &body body*

  A wrapper around `journaled` to produce frames of `external-event`s. `version` is `:infinity`. `replayed` is for primarily for marking and isolating non-deterministic processing.

  With `replayed`, the `in-event` is checked for consistency with the replay (as with `checked`), but `body` is not run (assuming it has a recorded expected outcome), and the outcome in the `out-event` is reproduced (see Replaying the outcome). For this scheme to work, `replayed` requires its `body` to be side-effect free, but it may be non-deterministic.

- **[glossary-term]** **invoked**

  Invoked refers to functions and blocks defined by `define-invoked` or `flet-invoked`. Invoked frames may be recorded in response to asynchronous events, and at replay the presence of its in-event triggers the execution of the function associated with the name of the event.

  On the one hand, `framed`, `checked`, `replayed` or plain `journaled` have In-events that are always predictable from the code and the preceding events. The control flow – on the level of recorded frames – is deterministic in this sense. On the other hand, Invoked encodes in its `in-event` what function to call next, introducing non-deterministic control flow.

  By letting events choose the code to run, Invoked resembles typical event sourcing frameworks. When Invoked is used exclusively, the journal becomes a sequence of events. In contrast, `journaled` and its wrappers put code first, and the journal will be a projection of the call tree.

- **[macro]** **define-invoked** *function-name args (name &key (version 1) insertable) &body body*

`define-invoked` is intended for recording asynchronous function invocations like event or signal handlers. It defines a function that records `versioned-event`s with `args` set to the actual arguments. At replay, it is invoked whenever the recorded `in-event` becomes the replay event.

`defun` and `checked` rolled into one, `define-invoked` defines a top-level function with `function-name` and `args` (only simple positional arguments are allowed) and wraps `checked` with `name`, the same `args` and `insertable` around `body`. Whenever an `in-event` becomes the replay event, and it has a `define-invoked` defined with the name of the event, `function-name` is invoked with `event-args`.

While `body`'s return values are recorded as usual, the defined function returns no values to make it less likely to affect control flow in a way that's not possible to reproduce when the function is called by the replay mechanism.

```
(defvar *state*)

(define-invoked foo (x) ("foo")
  (setq *state* (1+ x)))

(define-invoked bar (x) ("bar")
  (setq *state* (+ 2 x)))

(if (zerop (random 2))
    (foo 0)
    (bar 1))
```

The above can be alternatively implemented with `replayed` explicitly encapsulating the non-determinism:

```
(let ((x (replayed (choose) (random 2))))
  (if (zerop x)
      (checked (foo :args `(,x))
        (setq *state* (1+ x)))
      (checked (bar :args `(,x))
        (setq *state* (+ 2 x)))))
```

`fmakunbound` and `unintern` undefine invoked functions. Note that this is thread-safe only on SBCL. Do not delete `define-invoked` functions in production.

- **[macro]** **flet-invoked** *definitions &body body*

Like `define-invoked`, but with `flet` instead of `defun`. The event name and the function are associated in the dynamic extent of `body`. `with-journaling` does not change the bindings. The example in `define-invoked` can be rewritten as:

```
(let ((state nil))
  (flet-invoked ((foo (x) ("foo")
                   (setq state (1+ x)))
                 (bar (x) ("bar")
                   (setq state (+ 2 x))))
    (if (zerop (random 2))
```

```
      (foo 0)
      (bar 1))))
```

8.2 Bundles

Consider replaying the same code repeatedly, hoping to make progress in the processing. Maybe based on the availability of external input, the code may error out. After each run, one has to decide whether to keep the journal just recorded or stick with the replay journal. A typical solution to this would look like this:

```
(let ((record nil))
  (loop
    (setq record (make-in-memory-journal))
    (with-journaling (:record record :replay replay)
      ...)
    (when (and
           ;; RECORD is a valid replay of REPLAY ...
           (eq (journal-state record) :completed)
           ;; ... and is also significantly different from it ...
           (journal-diverged-p record))
      ;; so use it for future replays.
      (setq replay record))))
```

This is pretty much what bundles automate. The above becomes:

```
(let ((bundle (make-in-memory-bundle)))
  (loop
    (with-bundle (bundle)
      ...)))
```

With `file-journal`s, the motivating example above would be even more complicated, but `file-bundle`s work the same way as `in-memory-bundle`s.

- **[macro]** **with-bundle** *(bundle) &body body*

  This is like `with-journaling` where the `replay-journal` is the last successfully completed one in `bundle`, and the `record-journal` is a new one created in `bundle`. When `with-bundle` finishes, the record journal is in `journal-state` `:failed` or `:completed`.

  To avoid accumulating useless data, the new record is immediately deleted when `with-bundle` finishes if it has not diverged from the replay journal (see `journal-divergent-p`). Because `:failed` journals are always divergent in this sense, they are deleted instead based on whether there is already a previous failed journal in the bundle and the new record is identical to that journal (see `identical-journals-p`).

  It is a `journal-error` to have concurrent or nested `with-bundle`s on the same bundle.

8.3 The replay strategy

The replay process for both In-events and Out-events starts by determining how the generated event (the *new* event from now on) shall be replayed. Roughly, the decision is based on the `name`

and `version` of the new event and the replay event (the next event to be read from the replay). There are four possible strategies:

- **match**: A new in-event must match the replay event in its `args`. See Matching in-events for details. A new out-event must match the replay event's `exit` and `outcome`, see Matching out-events.
- **upgrade**: The new event is not matched to any replay event, but an event is consumed from the replay journal. This happens if the next new event has the same name as the replay event, but its version is higher.
- **insert**: The new event is not matched to any replay event, and no events are consumed from the replay journal, which may be empty. This is always the case for new `log-event`s and when there are no more events to read from the replay journal (unless `replay-eoj-error-p`). For `versioned-event`s, it is affected by setting `journaled`'s `insertable` to true (see Journaled for replay).

  The out-event's strategy is always *insert* if the strategy for the corresponding in-event was *insert*.
- Also, `end-of-journal`, `replay-name-mismatch` and `replay-version-downgrade` may be signalled. See the algorithm below details.

The strategy is determined by the following algorithm, invoked whenever an event is generated by a journaled block:

1. Log events are not matched to the replay. If the new event is a log event or a `replay-failure` has been signalled before (i.e. the record journal's `journal-state` is `:mismatched`), then **insert** is returned.
2. Else, log events to be read in the replay journal are skipped, and the next unread, non-log event is peeked at (without advancing the replay journal).
   - **end of replay**: If there are no replay events left, then:
     - If `replay-eoj-error-p` is `nil` in `with-journaling` (the default), **insert** is returned.
     - If `replay-eoj-error-p` is true, then **`end-of-journal`** is signalled.
   - **mismatched name**: Else, if the next unread replay event's name is not `equal` to the name of the new event, then:
     - For `versioned-event`s, **`replay-name-mismatch`** is signalled if `insertable` is `nil`, else **insert** is returned.
     - For `external-event`s, **`replay-name-mismatch`** is signalled.
   - **matching name**: Else, if the name of the next unread event in the replay journal is `equal` to the name of new event, then it is chosen as the *replay* event.
     - If the replay event's version is higher than the new event's version, then **`replay-version-downgrade`** is signalled.
     - If the two versions are equal, then **match** is returned.

    - If the new event's version is higher, then **upgrade** is returned.

Where `:infinity` is considered higher than any integer and equal to itself.

In summary:

```
| new event | end-of-replay     | mismatched name   | matching name |
|-----------+-------------------+-------------------+---------------|
| Log       | insert            | insert            | insert        |
| Versioned | insert/eoj-error  | insert/name-error | match-version |
| External  | insert/eoj-error  | insert/name-error | match-version |
```

Version matching (`match-version` above) is based on which event has a higher version:

```
| replay event    | =     | new event |
|-----------------+-------+-----------|
| downgrade-error | match | upgrade   |
```

- **[glossary-term]** **replay event**

  The replay event is the next event to be read from `replay-journal` which is not to be skipped. There may be no replay event if there are no more unread events in the replay journal.

  An event in the replay journal is skipped if it is a `log-event` or there is a `with-replay-filter` with a matching `:skip`. If `:skip` is in effect, the replay event may be indeterminate.

  Events from the replay journal are read when they are `:matched` or `:upgraded` (see The replay strategy), when nested events are echoed while Replaying the outcome, or when there is an invoked defined with the same name as the replay event.

  The replay event is available via `peek-replay-event`.

8.4 Matching in-events

If the replay strategy is *match*, then, for in-events, the matching process continues like this:

- If the `event-args` are not `equal`, then **`replay-args-mismatch`** signalled.
- At this point, two things might happen:
    - For `versioned-event`s, the block will be executed as normal and its outcome will be matched to the replay event (see Matching out-events).
    - For `external-event`s, the corresponding replay `out-event` is looked at. If there is one, meaning that the frame finished with an expected outcome, then its outcome will be replayed (see Replaying the outcome). If the `out-event` is missing, then `external-event`s behave like `versioned-event`s, and the block is executed.

8.4.1 Replaying the outcome

So, if an in-event is triggered that matches the replay, `event-version(0 1)` is `:infinity`, then normal execution is altered in the following manner:

- The journaled block is not executed.
- To keep execution and the replay journal in sync, events of frames nested in the current one are skipped over in the replay journal.
- All events (including `log-event`s) skipped over are echoed to the record journal. This serves to keep a trail of what happened during the original recording. Note that functions corresponding to invoked frames are called when their `in-event` is skipped over.
- The out-event corresponding to the in-event being processed is then read from the replay journal and is recorded again (to allow recording to function properly).

To be able to reproduce the outcome in the replay journal, some assistance may be required from `replay-values` and `replay-condition`:

- If the replay event has a normal return (i.e. `event-exit(0 1) :values`), then the recorded return values (in `event-outcome`) are returned immediately as in `(values-list (event-outcome replay-event))`. If `replay-values` is specified, it is called instead of `values-list`. See Working with unreadable values for an example.
- Similarly, if the replay event has unwound with an expected condition (has `event-exit :condition`), then the recorded condition (in `event-outcome`) is signalled as IN `(error (event-outcome replay-event))`. If `replay-condition` is specified, it is called instead of `error(0 1)`. `replay-condition` must not return normally, and it's a `journal-error` if it does.

`with-replay-filter`'s `no-replay-outcome` can selectively turn off replaying the outcome. See Testing on multiple levels, for an example.

8.5 Matching out-events

If there were no Replay failures during the matching of the `in-event`, and the conditions for Replaying the outcome were not met, then the block is executed. When the outcome of the block is determined, an `out-event` is triggered and is matched to the replay journal. The matching of out-events starts out as in The replay strategy with checks for `event-name` and `event-version`.

If the replay strategy is *insert* or *upgrade*, then the out-event is written to `record-journal`, consuming an event with a matching name from the `replay-journal` in the latter case. If the strategy is *match*, then:

- If the new event has an unexpected outcome, then **`replay-unexpected-outcome`** is signalled. Note that the replay event always has an expected outcome due to the handling of `record-unexpected-outcome`.
- If the new event has an expected outcome, then unless the new and replay event's `event-exit(0 1)`s are `eq` and their `event-outcome`s are `equal`, **`replay-outcome-mismatch`** is signalled.
- Else, the replay event is consumed and the new event is written to the `record-journal`.

Note that The replay strategy for the in-event and the out-event of the same frame may differ if the corresponding out-event is not present in `replay-journal`, which may be the case when

the recording process failed hard without unwinding properly, or when an unexpected outcome triggered the transition to `journal-state :logging`.

8.6 Replay failures

- **[condition]** **replay-failure** *serious-condition*

  A abstract superclass (never itself signalled) for all kinds of mismatches between the events produced and the replay journal. Signalled only in `journal-state :replaying` and only once per `with-journaling`. If a `replay-failure` is signalled for an `event`, then the event will be recorded, but `record-journal` will transition to `journal-state :mismatched`. Like `journaling-failure`, this is a serious condition because it is to be handled outside the enclosing `with-journaling`. If a `replay-failure` were to be handled inside the `with-journaling`, keep in mind that in `:mismatched`, replay always uses the *insert* replay strategy (see The replay strategy).

- **[reader]** **replay-failure-new-event** *replay-failure (:new-event)*

- **[reader]** **replay-failure-replay-event** *replay-failure (:replay-event)*

- **[reader]** **replay-failure-replay-journal** *replay-failure (= '(replay-journal))*

- **[condition]** **replay-name-mismatch** *replay-failure*

  Signalled when the new event's and replay event's `event-name` are not `equal`. The `replay-force-insert`, `replay-force-upgrade` restarts are provided.

- **[condition]** **replay-version-downgrade** *replay-failure*

  Signalled when the new event and the replay event have the same `event-name`, but the new event has a lower version. The `replay-force-upgrade` restart is provided.

- **[condition]** **replay-args-mismatch** *replay-failure*

  Signalled when the new event's and replay event's `event-args` are not `equal`. The `replay-force-upgrade` restart is provided.

- **[condition]** **replay-outcome-mismatch** *replay-failure*

  Signalled when the new event's and replay event's `event-exit(0 1)` and/or `event-outcome` are not `equal`. The `replay-force-upgrade` restart is provided.

- **[condition]** **replay-unexpected-outcome** *replay-failure*

  Signalled when the new event has an unexpected outcome. Note that the replay event always has an expected outcome due to the logic of `record-unexpected-outcome`. No restarts are provided.

- **[condition]** **replay-incomplete** *replay-failure*

  Signalled if there are unprocessed non-log events in `replay-journal` when `with-journaling` finishes and the body of `with-journaling` returned normally, which is to

prevent this condition to cancel an ongoing unwinding. No restarts are provided.

- **[restart]** **replay-force-insert**

  This restart forces The replay strategy to be `:insert`, overriding `replay-name-mismatch`. This is intended for upgrades, and extreme care must be taken not to lose data.

- **[restart]** **replay-force-upgrade**

  This restart forces The replay strategy to be `:upgrade`, overriding `replay-name-mismatch`, `replay-version-downgrade`, `replay-args-mismatch`, `replay-outcome-mismatch`. This is intended for upgrades, and extreme care must be taken not to lose data.

8.7 Upgrades and replay

The replay mechanism is built on the assumption that the tree of frames is the same when the code is replayed as it was when the replay journal was originally recorded. Thus, non-deterministic control flow poses a challenge, but non-determinism can be isolated with `external-event`s. However, when the code changes, we might find the structure of frames in previous recordings hard to accommodate. In this case, we might decide to alter the structure, giving up some of the safety provided by the replay mechanism. There are various tools at our disposal to control this tradeoff between safety and flexibility:

- We can insert individual frames with `journaled`'s `insertable`, upgrade frames by bumping `journaled`'s `version`, and filter frames with `with-replay-filter`. This option allows for the most consistency checks.
- The `replay-force-upgrade` and `replay-force-insert` restarts allow overriding The replay strategy, but their use requires great care to be taken.
- Or we may decide to keep the bare minimum of the replay journal around and discard everything except for `external-event`s. This option is equivalent to

```
(let ((*force-insertable* t))
  (with-replay-filter (:skip '((:name nil)))
    42))
```

- Rerecording the journal without replay might be another option if there are no `external-event`s to worry about.
- Finally, we can rewrite the replay journal using the low-level interface (see Streamlets reference). In this case, extreme care must be taken not to corrupt the journal (and lose data) as there are no consistency checks to save us.

With that, let's see how `with-replay-filter` works.

- **[macro]** **with-replay-streamlet** *(var) &body body*

  Open `replay-journal` for reading with `with-open-journal`, set the `read-position` on it to the event next read by the Replay mechanism (which is never a `log-event`). The low-level Reading from streamlets api is then available to inspect the contents of the replay. It is an error if `replay-journal` is `nil`.

- **[function] peek-replay-event**

  Return the replay event to be read from `replay-journal`. This is roughly equivalent to

```
(when (replay-journal)
  (with-replay-streamlet (streamlet)
    (peek-event streamlet))
```

  except `peek-replay-event` takes into account `with-replay-filter :map`, and it may return `(:indeterminate)` if `with-replay-filter :skip` is in effect and what events are to be skipped cannot be decided until the next in-event generated by the code.

  Imagine a business process for paying an invoice. In the first version of this process, we just pay the invoice:

```
(replayed (pay))
```

  We have left the implementation of PAY blank. In the second version, we need to get an approval first:

```
(when (replayed (get-approval)
        (= (random 2) 0))
  (replayed (pay)))
```

  Replaying a journal produced by the first version of the code with the second version would run into difficulties because inserting `external-event`s is tricky.

  We have to first decide how to handle the lack of approval in the first version. Here, we just assume the processes started by the first version get approval automatically. The implementation is based on a dummy `process` block whose version is bumped when the payment process changes and is inspected at the start of journaling.

  When v1 is replayed with v2, we introduce an `insertable`, versioned `get-approval` block that just returns `t`. When replaying the code again, still with v2, the `get-approval` block will be upgraded to `:infinity`.

```
(let ((bundle (make-in-memory-bundle)))
  ;; First version of the payment process. Just pay.
  (with-bundle (bundle)
    (checked (process :version 1))
    (replayed (pay)))
  ;; Second version of the payment process. Only pay if approved.
  (loop repeat 2 do
    (with-bundle (bundle)
      (let ((replay-process-event (peek-replay-event)))
        (checked (process :version 2))
        (when (if (and replay-process-event
                       (< (event-version replay-process-event) 2))
                  ;; This will be upgraded to :INFINITY the second
                  ;; time around the LOOP.
                  (checked (get-approval :insertable t)
                    t)
                  (replayed (get-approval)
```

```
                    (= (random 2) 0)))
          (replayed (pay)))))))
```

- **[macro]** **with-replay-filter** *(&key map skip no-replay-outcome) &body body*

  `with-replay-filter` performs journal upgrade during replay by allowing events to be transformed as they are read from the replay journal or skipped if they match some patterns. For how to add new blocks in a code upgrade, see `journaled`'s `:insertable` argument. In addition, it also allows some control over Replaying the outcome.

  - `map`: A function called with an event read from the replay journal which returns a transformed event. See Events reference. `map` takes effect before before `skip`.
  - `skip`: In addition to filtering out `log-event`s (which always happens during replay), filter out all events that belong to frames that match any of its `skip` patterns. Filtered out events are never seen by `journaled` as it replays events. `skip` patterns are of the format `(&key name version<)`, where `version<` is a valid `event-version`, and `name` may be `nil`, which acts as a wildcard.

    `skip` is for when `journaled` blocks are removed from the code, which would render replaying previously recorded journals impossible. Note that, for reasons of safety, it is not possible to filter `external-event`s.
  - `no-replay-outcome` is a list of `event-name`s. Replaying the outcome is prevented for frames with `equal` names. See Testing on multiple levels for an example.

  `with-replay-filter` affects only the immediately enclosing `with-journaling`. A `with-replay-filter` nested within another in the same `with-journaling` inherits the `skip` patterns of its parent, to which it adds its own. The `map` function is applied to before the parent's `map`.

  Examples of `skip` patterns:

```
;; Match events with name FOO and version 1, 2, 3 or 4
(:name foo :version< 5)
;; Match events with name BAR and any version
(:name bar :version< :infinity)
;; Same as the previous
(:name bar)
;; Match all names
(:name nil)
;; Same as the previous
()
```

  Skipping can be thought of as removing nodes of the tree of frames, connecting its children to its parent. The following example removes frames `j1` and `j2` from around `j3`, the `j1` frame from within `j3`, and the third `j1` frame.

```
(let ((journal (make-in-memory-journal)))
  ;; Record trees J1 -> J2 -> J3 -> J1, and J1.
  (with-journaling (:record journal)
    (checked (j1)
```

```
    (checked (j2)
      (checked (j3)
        (checked (j1)
          42))))
  (checked (j1)
    7))
;; Filter out all occurrences of VERSIONED-EVENTs named J1 and
;; J2 from the replay, leaving only J3 to match.
(with-journaling (:replay journal :record t :replay-eoj-error-p t)
  (with-replay-filter (:skip '((:name j1) (:name j2)))
    (checked (j3)
      42)))
```

9 Testing

Having discussed the Replay mechanism, next are Testing and Persistence, which rely heavily on replay. Suppose we want to unit test user registration. Unfortunately, the code communicates with a database service and also takes input from the user. A natural solution is to create mock objects for these external systems to unshackle the test from the cumbersome database dependency and to allow it to run without user interaction.

We do this below by wrapping external interaction in `journaled` with `:version :infinity` (see Replaying the outcome).

```
(defparameter *db* (make-hash-table))

(defun set-key (key value)
  (replayed ("set-key" :args `(,key ,value))
    (format t "Updating db~%")
    (setf (gethash key *db*) value)
    nil))

(defun get-key (key)
  (replayed ("get-key" :args `(,key))
    (format t "Query db~%")
    (gethash key *db*)))

(defun ask-username ()
  (replayed ("ask-username")
    (format t "Please type your username: ")
    (read-line)))

(defun maybe-win-the-grand-prize ()
  (checked ("maybe-win-the-grand-prize")
    (when (= 1000000 (hash-table-count *db*))
      (format t "You are the lucky one!"))))

(defun register-user (username)
  (unless (get-key username)
    (set-key username `(:user-object :username ,username))
```

```
    (maybe-win-the-grand-prize)))
```

Now, we write a test that records these interactions in a file when it's run for the first time.

```
(define-file-bundle-test (test-user-registration
                          :directory (asdf:system-relative-pathname
                                      :journal "test/registration/"))
  (let ((username (ask-username)))
    (register-user username)
    (assert (get-key username))
    (register-user username)
    (assert (get-key username))))

;; Original recording: everything is executed
JRN> (test-user-registration)
Please type your username: joe
Query db
Updating db
Query db
Query db
Query db
=> NIL
```

On reruns, none of the external stuff is executed. The return values of the external `journaled` blocks are replayed from the journal:

```
;; Replay: all external interactions are mocked.
JRN> (test-user-registration)
=> NIL
```

Should the code change, we might want to upgrade carefully (see Upgrades and replay) or just rerecord from scratch:

```
JRN> (test-user-registration :rerecord t)
Please type your username: joe
Query db
Updating db
Query db
Query db
Query db
=> NIL
```

Thus satisfied that our test runs, we can commit the journal file in the bundle into version control. Its contents are:

```
(:IN "ask-username" :VERSION :INFINITY)
(:OUT "ask-username" :VERSION :INFINITY :VALUES ("joe" NIL))
(:IN "get-key" :VERSION :INFINITY :ARGS ("joe"))
(:OUT "get-key" :VERSION :INFINITY :VALUES (NIL NIL))
(:IN "set-key" :VERSION :INFINITY :ARGS ("joe" (:USER-OBJECT :USERNAME "joe")))
(:OUT "set-key" :VERSION :INFINITY :VALUES (NIL))
```

```
(:IN "maybe-win-the-grand-prize" :VERSION 1)
(:OUT "maybe-win-the-grand-prize" :VERSION 1 :VALUES (NIL))
(:IN "get-key" :VERSION :INFINITY :ARGS ("joe"))
(:OUT "get-key" :VERSION :INFINITY :VALUES ((:USER-OBJECT :USERNAME "joe") T))
(:IN "get-key" :VERSION :INFINITY :ARGS ("joe"))
(:OUT "get-key" :VERSION :INFINITY :VALUES ((:USER-OBJECT :USERNAME "joe") T))
(:IN "get-key" :VERSION :INFINITY :ARGS ("joe"))
(:OUT "get-key" :VERSION :INFINITY :VALUES ((:USER-OBJECT :USERNAME "joe") T))
```

Note that when this journal is replayed, new `versioned-event`s are required to match the replay. So, after the original recording, we can check by eyeballing that the record represents a correct execution. Then on subsequent replays, even though `maybe-win-the-grand-prize` sits behind `register-user` and is hard to test with `assert`s, the replay mechanism verifies that it is called only for new users.

This record-and-replay style of testing is not the only possibility: direct inspection of a journal with the low-level events api (see Events reference) can facilitate checking non-local invariants.

- **[macro]** **define-file-bundle-test** *(name &key directory (equivalentp t)) &body body*

  Define a function with `name` for record-and-replay testing. The function's `body` is executed in a `with-bundle` to guarantee replayability. The bundle in question is a `file-bundle` created in `directory`. The function has a single keyword argument, `rerecord`. If `rerecord` is true, the bundle is deleted with `delete-file-bundle` to start afresh.

  Furthermore, if `body` returns normally, and it is a replay of a previous run, and `equivalentp`, then it is `assert`ed that the record and replay journals are `equivalent-replay-journals-p`. If this check fails, `record-journal` is discarded when the function returns. In addition to the replay consistency, this checks that no inserts or upgrades were performed (see The replay strategy).

9.1 Testing on multiple levels

Nesting `replayed`s (that is, frames of `external-event`s) is not obviously useful since the outer `replayed` will be replayed by outcome, and the inner one will be just echoed to the record journal. However, if we turn off Replaying the outcome for the outer, the inner will be replayed.

This is useful for testing layered communication. For example, we might have written code that takes input from an external system (`read-line`) and does some complicated processing (`read-from-string`) before returning the input in a form suitable for further processing. Suppose we wrap `replayed` around `read-from-string` for Persistence because putting it around `read-line` would expose low-level protocol details in the journal, making protocol changes difficult.

However, upon realizing that `read-from-string` was not the best tool for the job and switching to `parse-integer`, we want to test by replaying all previously recorded journals. For this, we prevent the outer `replayed` from being replayed by outcome with `with-replay-filter`:

```
(let ((bundle (make-in-memory-bundle)))
  ;; Original with READ-FROM-STRING
  (with-bundle (bundle)
    (replayed ("accept-number")
```

```
      (values (read-from-string (replayed ("input-number")
                                  (read-line))))))
  ;; Switch to PARSE-INTEGER and test by replay.
  (with-bundle (bundle)
    (with-replay-filter (:no-replay-outcome '("accept-number"))
      (replayed ("accept-number")
        ;; 1+ is our bug.
        (values (1+ (parse-integer (replayed ("input-number")
                                     (read-line)))))))))
```

The inner `input-number` block is replayed by outcome, and `parse-integer` is called with the string `read-line` returned in the original invocation. The outcome of the outer `accept-number` block checked as if it was a `versioned-event` and we get a `replay-outcome-mismatch` due to the bug.

10 Persistence

10.1 Persistence tutorial

Let's write a simple game.

```
(defun play-guess-my-number ()
  (let ((my-number (replayed (think-of-a-number)
                     (random 10))))
    (format t "~%I thought of a number.~%")
    (loop for i upfrom 0 do
      (write-line "Guess my number:")
      (let ((guess (replayed (read-guess)
                     (values (parse-integer (read-line))))))
        (format t "You guessed ~D.~%" guess)
        (when (= guess my-number)
          (checked (game-won :args `(,(1+ i))))
          (format t "You guessed it in ~D tries!" (1+ i))
          (return))))))

(defparameter *the-evergreen-game* (make-in-memory-bundle))
```

Original recording Unfortunately, the implementation is lacking in the input validation department. In the transcript below, `parse-integer` fails with `junk in string` when the user enters `not a number`:

```
CL-USER> (handler-case
             (with-bundle (*the-evergreen-game*)
               (play-guess-my-number))
           (error (e)
             (format t "Oops. ~A~%" e)))
I thought of a number.
Guess my number:
7 ; real user input
You guessed 7.
```

```
Guess my number:
not a number ; real user input
Oops. junk in string "not a number"
```

Replay and extension Instead of fixing this bug, we just restart the game from the beginning, Replaying the outcome of external interactions marked with `replayed`:

```
CL-USER> (with-bundle (*the-evergreen-game*)
           (play-guess-my-number))
I thought of a number.
Guess my number:
You guessed 7.
Guess my number: ; New recording starts here
5 ; real user input
You guessed 5.
Guess my number:
4 ; real user input
You guessed 4.
Guess my number:
2 ; real user input
You guessed 2.
You guessed it in 4 tries!
```

It's evergreen We can now replay this game many times without any user interaction:

```
CL-USER> (with-bundle (*the-evergreen-game*)
           (play-guess-my-number))
I thought of a number.
Guess my number:
You guessed 7.
Guess my number:
You guessed 5.
Guess my number:
You guessed 4.
Guess my number:
You guessed 2.
You guessed it in 4 tries!
```

The generated events This simple mechanism allows us to isolate external interactions and write tests in record-and-replay style based on the events produced:

```
CL-USER> (list-events *the-evergreen-game*)
((:IN THINK-OF-A-NUMBER :VERSION :INFINITY)
 (:OUT THINK-OF-A-NUMBER :VERSION :INFINITY :VALUES (2))
 (:IN READ-GUESS :VERSION :INFINITY)
 (:OUT READ-GUESS :VERSION :INFINITY :VALUES (7))
 (:IN READ-GUESS :VERSION :INFINITY :ARGS NIL)
 (:OUT READ-GUESS :VERSION :INFINITY :VALUES (5))
 (:IN READ-GUESS :VERSION :INFINITY :ARGS NIL)
 (:OUT READ-GUESS :VERSION :INFINITY :VALUES (4))
```

```
(:IN READ-GUESS :VERSION :INFINITY :ARGS NIL)
(:OUT READ-GUESS :VERSION :INFINITY :VALUES (2))
(:IN GAME-WON :VERSION 1 :ARGS (4))
(:OUT GAME-WON :VERSION 1 :VALUES (NIL)))
```

In fact, being able to replay this game at all already checks it through the `game-won` event that the number of tries calculation is correct.

In addition, thus being able to reconstruct the internal state of the program gives us persistence by replay. If instead of a `in-memory-bundle`, we used a `file-bundle`, the game would have been saved on disk without having to write any code for saving and loading the game state.

Discussion Persistence by replay, also known as event sourcing, is appropriate when the external interactions are well-defined and stable. Storing events shines in comparison to persisting state when the control flow is too complicated to be interrupted and resumed easily. Resuming execution in deeply nested function calls is fraught with such peril that it is often easier to flatten the program into a state machine, which is as pleasant as manually managing continuations.

In contrast, the Journal library does not favour certain styles of control flow and only requires that non-determinism is packaged up in `replayed`, which allows it to reconstruct the state of the program from the recorded events at any point during its execution and resume from there.

10.2 Synchronization to storage

In the following, we explore how journals can serve as a persistence mechanism and the guarantees they offer. The high-level summary is that journals with `sync` can serve as a durable and consistent storage medium. The other two ACID properties, atomicity and isolation, do not apply because Journal is single-client and does not need transactions.

- **[glossary-term] aborted execution**

  Aborted execution is when the operating system or the application crashes, calls `abort()`, is killed by a `sigkill` signal or there is a power outage. Synchronization guarantees are defined in the face of aborted execution and do not apply to hardware errors, Lisp or OS bugs.

- **[glossary-term] data event**

  Data events are the only events that may be non-deterministic. They record information that could change if the same code were run multiple times. Data events typically correspond to interactions with the user, servers or even the random number generator. Due to their non-determinism, they are the only parts of the journal not reproducible by rerunning the code. In this sense, only the data events are not redundant with the code, and whether other events are persisted does not affect durability. There are two kinds of data events:

  - An `external-event` that is also an `out-event`.
  - The `in-event` of an invoked function, which lies outside the normal, deterministic control flow.

10.2.1 Synchronization strategies

When a journal or bundle is created (see `make-in-memory-journal`, `make-file-journal`, `make-in-memory-bundle`, `make-file-bundle`), the `sync` option determines when – as a `record-journal` – the recorded events and `journal-state` changes are persisted durably. For `file-journal`s, persisting means calling something like `fsync`, while for `in-memory-journal`s, a user defined function is called to persist the data.

- `nil`: Never synchronize. A `file-journal`'s file may be corrupted on aborted execution. In `in-memory-journal`s, `sync-fn` is never called.
- `t`: This is the *no data loss* setting with minimal synchronization. It guarantees *consistency* (i.e. no corruption) and *durability* up to the most recent data event written in `journal-state :recording` or for the entire record journal in states `:failed` and `:completed`. `:failed` or `:completed` is guaranteed when leaving `with-journaling` at the latest.
- Values other than `nil` and `t` are reserved for future extensions. Using them triggers a `journal-error`.

10.2.2 Synchronization with in-memory journals

Unlike `file-journal`s, `in-memory-journal`s do not have any built-in persistent storage backing them, but with `sync-fn`, persistence can be tacked on. If non-`nil`, `sync-fn` must be a function of a single argument, an `in-memory-journal`. `sync-fn` is called according to Synchronization strategies, and upon normal return the journal must be stored durably.

The following example saves the entire journal history when a new data event is recorded. Note how `sync-to-db` is careful to overwrite `*db*` only if it is called with a journal that has not failed the replay (as in Replay failures) and is sufficiently different from the replay journal as determined by `journal-divergent-p`.

```
(defparameter *db* ())

(defun sync-to-db (journal)
  (when (and (member (journal-state journal)
                     '(:recording :logging :completed))
             (journal-divergent-p journal))
    (setq *db* (journal-events journal))
    (format t "Saved ~S~%New events from position ~S~%" *db*
            (journal-previous-sync-position journal))))

(defun make-db-backed-record-journal ()
  (make-in-memory-journal :sync-fn 'sync-to-db))

(defun make-db-backed-replay-journal ()
  (make-in-memory-journal :events *db*))

(with-journaling (:record (make-db-backed-record-journal)
                  :replay (make-db-backed-replay-journal))
  (replayed (a)
    2)
  (ignore-errors
```

```
    (replayed (b)
      (error "Whoops"))))
.. Saved #((:IN A :VERSION :INFINITY)
..         (:OUT A :VERSION :INFINITY :VALUES (2)))
.. New events from position 0
.. Saved #((:IN A :VERSION :INFINITY)
..         (:OUT A :VERSION :INFINITY :VALUES (2))
..         (:IN B :VERSION :INFINITY)
..         (:OUT B :ERROR ("SIMPLE-ERROR" "Whoops")))
.. New events from position 2
..
```

In a real application, external events often involve unreliable or high-latency communication. In the above example, block `b` signals an error, say, to simulate some kind of network condition. Now, a new journal *for replay* is created and initialized with the saved events, and the whole process is restarted.

```
(defun run-with-db ()
  (with-journaling (:record (make-db-backed-record-journal)
                    :replay (make-db-backed-replay-journal))
    (replayed (a)
      (format t "A~%")
      2)
    (replayed (b)
      (format t "B~%")
      3)))

(run-with-db)
.. B
.. Saved #((:IN A :VERSION :INFINITY)
..         (:OUT A :VERSION :INFINITY :VALUES (2))
..         (:IN B :VERSION :INFINITY)
..         (:OUT B :VERSION :INFINITY :VALUES (3)))
.. New events from position 0
..
=> 3
```

Note that on the rerun, block `a` is not executed because external events are replayed simply by reproducing their outcome, in this case returning 2. See Replaying the outcome. Block `b`, on the other hand, was rerun because it had an unexpected outcome the first time around. This time it ran without error, a data event was triggered, and `sync-fn` was invoked.

If we were to invoke the now completed `run-with-db` again, it would simply return 3 without ever invoking `sync-fn`:

```
(run-with-db)
=> 3
```

With `journal-replay-mismatch`, `sync-fn` can be optimized to to reuse the sequence of events in the replay journal up until the point of divergence.

10.2.3 Synchronization with file journals

For `file-journal`s, `sync` determines when the events written to the `record-journal` and its `journal-state` will be persisted durably in the file. Syncing to the file involves two calls to `fsync` and is not cheap.

Syncing events to files is implemented as follows.

- When the journal file is created, its parent directory is immediately fsynced to make sure that the file will not be lost on aborted execution.
- When an event is about to be written the first time after file creation or after a sync, a transaction start marker is written to the file.
- Any number of events may be subsequently written until syncing is deemed necessary (see Synchronization strategies).
- At this point, `fsync` is called to flush all event data and state changes to the file, and the transaction start marker is *overwritten* with a transaction completed marker and another `fsync` is performed.
- When reading back this file (e.g. for replay), an open transaction marker is treated as the end of file.

Note that this implementation assumes that after writing the start transaction marker, a crash cannot leave any kind of garbage bytes around: it must leave zeros. This is not true for all filesytems. For example, ext3/ext4 with `data=writeback` can leave garbage around.

11 Safety

Thread safety Changes to journals come in two varieties: adding an event and changing the `journal-state`. Both are performed by `journaled` only unless the low-level streamlet interface is used (see Streamlets reference). Using `journaled` wrapped in a `with-journaling`, `with-bundle`, or `:log-record` without `with-journaling` is thread-safe.

- Every journal is guaranteed to have at most a single writer active at any time. Writers are mainly `with-journaling` and `with-bundle`, but any journals directly logged to have a log writer stored in the journal object. See Logging.
- `with-journaling` and `with-bundle` have dynamic extent as writers, but log writers of journals have indefinite extent: once a journal is used as a `log-record`, there remains a writer.
- Attempting to create a second writer triggers a `journal-error`.
- Writing to the same journal via `:log-record` from multiple threads concurrently is possible since this doesn't create multiple writers. It is ensured with locking that events are written atomically. Frames can be interleaved, but these are `log-event`s, so this does not affect replay.
- The juggling of replay and record journals performed by `with-bundle` is also thread-safe.

- It is ensured that there is at most one `file-journal` object in the same Lisp image is backed by the same file.
- Similarly, there is at most `file-bundle` object for a directory.

Process safety Currently, there is no protection against multiple OS processes writing the same `file-journal` or `file-bundle`.

Signal safety Journal is *designed* to be async-unwind safe but *not reentrant*. Interrupts are disabled only for the most critical cleanup forms. If a thread is killed without unwinding, that constitutes aborted execution, so guarantees about Synchronization to storage apply, but `journal` objects written by the thread are not safe to access, and the Lisp should probably be restarted.

12 Events reference

Events are normally triggered upon entering and leaving the dynamic extent of a `journaled` block (see In-events and Out-events) and also by `logged`. Apart from being part of the low-level substrate of the Journal library, working with events directly is sometimes useful when writing tests that inspect recorded events. Otherwise, skip this entire section.

All `event`s have `event-name` and `event-version`(`0` `1`), which feature prominently in The replay strategy. After the examples in In-events and Out-events, the following example is a reminder of how events look in the simplest case.

```
(with-journaling (:record t)
  (journaled (foo :version 1 :args '(1 2))
    (+ 1 2))
  (logged () "Oops")
  (list-events))
=> ((:IN FOO :VERSION 1 :ARGS (1 2))
    (:OUT FOO :VERSION 1 :VALUES (3))
    (:LEAF "Oops"))
```

So, a `journaled` block generates an `in-event` and an `out-event`, which are simple property lists. The following reference lists these properties, their semantics and the functions to read them.

- **[type]** **event**

  An event is either an `in-event`, an `out-event` or a `leaf-event`.

- **[function]** **event=** *event-1 event-2*

  Return whether `event-1` and `event-2` represent the same event. In- and out-events belonging to the same frame are *not* the same event. `event-outcome`s are not compared when `event-exit`(`0` `1`) is `:error` to avoid undue dependence on implementation specific string representations. This function is useful in conjunction with `make-in-event` and `make-out-event` to write tests.

- **[function]** **event-name** *event*

The name of an event can be of any type. It is often a symbol or a string. When replaying, names are compared with `equal`. All `event`s have names. The names of the in- and out-events belonging to the same frame are the same.

12.1 Event versions

- **[function]** **event-version** *event*

  Return the version of `event` of type `event-version`.

- **[function]** **log-event-p** *event*

  See if `event` is a `log-event`.

- **[function]** **versioned-event-p** *event*

  See if `event` is a `versioned-event`.

- **[function]** **external-event-p** *event*

  See if `event` is an `external-event`.

12.2 In-events

- **[type]** **in-event**

  `in-event`s are triggered upon entering the dynamic extent of a `journaled` block. `in-event`s have `event-name`, `event-version`, and `event-args`. See In-events for a more introductory treatment.

- **[function]** **in-event-p** *event*

  See if `event` is a `in-event`.

- **[function]** **make-in-event** *&key name version args*

  Create an `in-event` with `name`, `version` (of type `event-version`) and `args` as its `event-name`, `event-version` and `event-args`.

- **[function]** **event-args** *in-event*

  Return the arguments of `in-event`, normally populated using the `args` form in `journaled`.

12.3 Out-events

- **[type]** **out-event**

  `out-event`s are triggered upon leaving the dynamic extent of the `journaled` block. `out-event`s have `event-name`, `event-version`, `event-exit` and `event-outcome`. See Out-events for a more introductory treatment.

- **[function]** **out-event-p** *event*

  See if `event` is an `out-event`.

- **[function]** **make-out-event** *&key name version exit outcome*

  Create an `out-event` with `name`, `version` (of type `event-version`), `exit` (of type `event-exit`), and `outcome` as its `event-name`, `event-version`, `event-exit` and `event-outcome`.

- **[function]** **event-exit** *out-event*

  Return how the journaled block finished. See `event-exit` for the possible types.

- **[function]** **expected-outcome-p** *out-event*

  See if `out-event` has an expected outcome.

- **[function]** **unexpected-outcome-p** *out-event*

  See if `out-event` has an unexpected outcome.

- **[function]** **event-outcome** *out-event*

  Return the outcome of the frame (or loosely speaking of a block) to which `out-event` belongs.

12.4 Leaf-events

- **[type]** **leaf-event**

  Leaf events are triggered by `logged`. Unlike `in-event`s and `out-event`s, which represent a frame, leaf events represent a point in execution thus cannot have children. They are also the poorest of their kind: they only have an `event-name`. Their `version` is always `nil`, which makes them `log-event`s.

- **[function]** **leaf-event-p** *event*

  See if `event` is a `leaf-event`.

- **[function]** **make-leaf-event** *name*

  Create a `leaf-event` with `name`.

13 Journals reference

In Basics, we covered the bare minimum needed to work with journals. Here, we go into the details.

- **[class]** **journal**

  `journal` is an abstract base class for a sequence of events. In case of `file-journal`s, the events are stored in a file, while for `in-memory-journal`s, they are in a Lisp array. When a journal is opened, it is possible to perform I/O on it (see Streamlets reference), which is normally taken care of by `with-journaling`. For this reason, the user's involvement with journals normally only consists of creating and using them in `with-journaling`.

- **[reader]** **journal-state** *journal (:state)*

  Return the state of `journal`, which is of type `journal-state`.

- **[reader]** **journal-sync** *journal (:sync = nil)*

  The `sync` argument specified at instantiation. See Synchronization strategies.

- **[function]** **sync-journal** *&optional (journal (record-journal))*

  Durably persist changes made to `journal` if `journal-sync` is `t`. The changes that are persisted are

  - `write-event`s and `journal-state` changes made in an enclosing `with-journaling`; and
  - `log-record`s from any thread.

  In particular, writes made in a `with-journaling` in another thread are not persisted. `sync-journal` is a noop if `journal-sync` is `nil`. It is safe to call from any thread.

- **[reader]** **journal-replay-mismatch** *journal (= nil)*

  If `journal-divergent-p`, then this is a list of two elements: the `read-position`s in the `record-journal` and `replay-journal` of the first events that were different (ignoring `log-event`s). It is `nil`, otherwise.

- **[function]** **journal-divergent-p** *journal*

  See if `with-journaling` recorded any event so far in this journal that was not `equal` to its replay event or it had no corresponding replay event. This completely ignores `log-event`s in both journals being compared and can be called any time during Replay. It plays a role in `with-bundle` deciding when a journal is important enough to keep and also in Synchronization with in-memory journals.

  The position of the first mismatch is available via `journal-replay-mismatch`.

13.1 Comparing journals

After replay finished (i.e. `with-journaling` completed), we can ask whether there were any changes produced. This is answered in the strictest sense by `identical-journals-p` and somewhat more functionally by `equivalent-replay-journals-p`.

Also see `journal-divergent-p`.

- **[generic-function]** **identical-journals-p** *journal-1 journal-2*

  Compare two journals in a strict sense: whether they have the same `journal-state` and the lists of their events (as in `list-events`) are `equal`.

- **[generic-function]** **equivalent-replay-journals-p** *journal-1 journal-2*

  See if two journals are equivalent when used the for `replay` in `with-journaling`. `equivalent-replay-journals-p` is like `identical-journals-p`, but it ignores `log-`

events and allows events with `event-exit(0 1)` `:error` to differ in their outcomes, which may very well be implementation specific, anyway. Also, it considers two groups of states as different `:new`, `:replaying`, `:mismatched`, `:failed` vs `:recording`, `:logging`, COMPLETED.

The rest of section is about concrete subclasses of `journal`.

13.2 In-memory journals

- **[class]** **in-memory-journal** *journal*

  `in-memory-journal`s are backed by a non-persistent Lisp array of events. Much quicker than `file-journal`s, they are ideal for smallish journals persisted manually (see Synchronization with in-memory journals for an example).

  They are also useful for writing tests based on what events were generated. They differ from `file-journal`s in that events written to `in-memory-journal`s are not serialized (and deserialized on replay) with the following consequences for the objects recorded by `journaled` (i.e. its `name`, `args` arguments, and also the return `values(0 1)` of the block, or the value returned by `condition`):

  - These objects need not be readable.
  - Their identity (`eq`ness) is not lost.
  - They must **must not be mutated** in any way.

- **[function]** **make-in-memory-journal** *&key (events nil eventsp) state (sync nil syncp) sync-fn*

  Create an `in-memory-journal`.

  The returned journal's `journal-state` will be set to `state`. If `state` is `nil`, then it is replaced by a default value, which is `:completed` if the `events` argument is provided, else it is `:new`.

  Thus, `(make-in-memory-journal)` creates a journal suitable for recording, and to make a replay journal, use `:state :completed` with some sequence of `events`:

  ```
  (make-in-memory-journal :events '((:in foo :version 1)) :state :completed)
  ```

  `sync` determines when `sync-fn` will be invoked on the `record-journal`. `sync` defaults to `t` if `sync-fn`, else to `nil`. For a description of possible values, see Synchronization strategies. For more discussion, see Synchronization with in-memory journals.

- **[reader]** **journal-events** *in-memory-journal (:events)*

  A sequence of events in the journal. Not to be mutated by client code.

- **[reader]** **journal-previous-sync-position** *in-memory-journal (= 0)*

  The length of `journal-events` at the time of the most recent invocation of `sync-fn`.

13.3 File journals

- **[class]** **file-journal** *journal*

  A `file-journal` is a journal whose contents and `journal-state` are persisted in a file. This is the `journal` subclass with out-of-the-box persistence, but see File bundles for a more full-featured solution for repeated Replays.

  Since serialization in `file-journals` is built on top of Lisp `read` and `write`, everything that `journaled` records in events (i.e. its `name`, `args` arguments, and also the return `values`(`0` `1`) of the block, or the value returned by `condition`) must be readable.

  File journals are human-readable and editable by hand with some care. When editing, the following needs to be remembered:

  - The first character of the file represents its `journal-state`. It is a `#\Space` (for state `:new`, `:replaying`, `:mismatched` and `:failed`), or a `#\Newline` (for state `:recording`, `:logging` and `:completed`).
  - If the journal has `sync` (see Synchronization strategies), then between two events, there may be `#\Del` (also called `#\Rubout`) or `#\Ack` characters (`char-code` 127 and 6). `#\Del` marks the end of the journal contents that may be read back: it's kind of an uncommitted-transaction marker for the events that follow it. `#\Ack` characters, of which there may be many in the file, mark the sequence of events until the next marker of either kind as valid (or committed). `#\Ack` characters are ignored when reading the journal.

  Thus, when editing a file, don't change the first character and leave the `#\Del` character, if any, where it is. Also see Synchronization with file journals.

- **[function]** **make-file-journal** *pathname &key sync*

  Return a `file-journal` backed by the file with `pathname`. The file is created when the journal is opened for writing. For a description of `sync`, see Synchronization strategies.

  If there is already an existing `file-journal` backed by the same file, then that object is returned. If the existing object has different options (e.g. it has `sync t` while the `sync` argument is `nil` here), then a `journal-error` is signalled.

  If there is already an existing `file-journal` backed by the same file, the `journal-state` is not `:new`, but the file doesn't exist, then the existing object is **invalidated**: attempts to write will fail with `journal-error`. If the existing journal object is being written, then invalidation fails with a `journal-error`. After invalidation, a new `file-journal` object is created.

- **[reader]** **pathname-of** *file-journal (:pathname)*

  The pathname of the file backing the journal.

13.4 Pretty-printing journals

- **[class]** **pprint-journal** *journal*

Events written to a `pprint-journal` have a customizable output format. `pprint-journal`s are intended for producing prettier output for Logging and Tracing, but they do not support reads, so they cannot be used as a `replay-journal` or in `list-events`, for example. On the other hand, events written to `pprint-journal`s need not be readable.

- **[function]** **make-pprint-journal** *&key (stream (make-synonym-stream '*standard-output*)) (pretty t) (prettifier 'prettify-event) log-decorator*

  Creates a `pprint-journal`.

- **[accessor]** **pprint-journal-stream** *pprint-journal (:stream = *standard-output*)*

  The stream where events are dumped. May be set any time to another `stream`.

- **[accessor]** **pprint-journal-pretty** *pprint-journal (:pretty = t)*

  Whether to use `pprint-journal-prettifier` or write events in as the property lists they are. A boolean-valued symbol.

- **[accessor]** **pprint-journal-prettifier** *pprint-journal (:prettifier = 'prettify-event)*

  A function like `prettify-event` that writes an event to a stream. Only used when `pprint-journal-pretty`, this is the output format customization knob. Also see decorations.

14 Bundles reference

In Bundles, we covered the repeated replay problem that `with-bundle` automates. Here, we provide a reference for the bundle classes.

- **[class]** **bundle**

  A `bundle` consists of a sequence of journals which are all reruns of the same code, hopefully making more and more progress towards completion. These journals are Replays of the previous successful one, extending it with new events. Upon replay (see `with-bundle`), the latest journal in the bundle in `journal-state` `:completed` plays the role of the replay journal, and a new journal is added to the bundle for recording. If the replay succeeds, this new journal eventually becomes `:completed` and takes over the role of the replay journal for future replays until another replay succeeds. When the bundle is created and it has no journals yet, the replay journal is an empty, completed one.

  This is an abstract base class. Direct subclasses are `in-memory-bundle` and `file-bundle`.

- **[accessor]** **max-n-failed** *bundle (:max-n-failed = 1)*

  If `max-n-failed` is non-`nil`, and the number of journals of `journal-state` `:failed` in the bundle exceeds its value, then some journals (starting with the oldest) are deleted.

- **[accessor]** **max-n-completed** *bundle (:max-n-completed = 1)*

  If `max-n-completed` is non-`nil`, and the number of journals of `journal-state` `:completed` in the bundle exceeds its value, then some journals (starting with the oldest) are deleted.

14.1 In-memory bundles

- **[class]** **in-memory-bundle** *bundle*

  An `in-memory-bundle` is a `bundle` that is built on `in-memory-journal`s. `in-memory-bundle`s have limited utility as a persistence mechanism and are provided mainly for reasons of symmetry and for testing. See Synchronization with in-memory journals for an example of how to achieve persistence without bundles.

- **[function]** **make-in-memory-bundle** *&key (max-n-failed 1) (max-n-completed 1) sync sync-fn*

  Create a new `in-memory-bundle` with `max-n-failed` and `max-n-completed`. `sync` and `sync-fn` are passed on to `make-in-memory-journal`.

14.2 File bundles

- **[class]** **file-bundle** *bundle*

  A `file-bundle` is a `bundle` that is built on `file-journal`s. It provides easy replay-based persistence.

- **[reader]** **directory-of** *file-bundle (:directory)*

  The directory where the files backing the `file-journal`s in the `file-bundle` are kept.

- **[function]** **make-file-bundle** *directory &key (max-n-failed 1) (max-n-completed 1) sync*

  Return a `file-bundle` object backed by `file-journal`s in `directory`. See `max-n-failed` and `max-n-completed`. For a description of `sync`, see Synchronization strategies.

  If there is already a `file-bundle` with the same directory (according to `truename`), that object is returned if it has the same `max-n-failed`, `max-n-completed` and `sync` options, else `journal-error` is signalled.

- **[function]** **delete-file-bundle** *directory*

  Delete all journal files (`*.jrn`) from `directory`. Delete the directory if empty after the journal files were deleted, else signal an error. Existing `file-bundle` objects are not updated, so `make-file-journal` with FORCE-RELOAD may be required.

15 Streamlets reference

This section is relevant mostly for implementing new kinds of `journal`s in addition to `file-journal`s and `in-memory-journal`s. In normal operation, `streamlet`s are not worked with directly.

15.1 Opening and closing

- **[class]** **streamlet**

A `streamlet` is a handle to perform I/O on a `journal`. The high-level stuff (`with-journaling`, `journaled`, etc) is built on top of streamlets.

- **[reader]** **journal** *streamlet (:journal)*

  The `journal` that was passed to `open-streamlet`. This is the journal `streamlet` operates on.

- **[generic-function]** **open-streamlet** *journal &key direction*

  Return a `streamlet` suitable for performing I/O on `journal`. `direction` (defaults to `:input`) is one of `:input`, `:output`, `:io`, and it has the same purpose as the similarly named argument of `cl:open`.

- **[generic-function]** **close-streamlet** *streamlet*

  Close `streamlet`, which was returned by `open-streamlet`. After closing, `streamlet` may not longer be used for IO.

- **[generic-function]** **make-streamlet-finalizer** *streamlet*

  Return `nil` or a function of no arguments suitable as a finalizer for `streamlet`. That is, a function that closes `streamlet` but holds no reference to it. This is intended for streamlets that are not dynamic-extent, so using `with-open-journal` is not appropriate.

- **[generic-function]** **open-streamlet-p** *streamlet*

  Return true if `streamlet` is open. `streamlet`s are open until they have been explicitly closed with `close-streamlet`.

- **[function]** **input-streamlet-p** *streamlet*

  See if `streamlet` was opened for input (the `direction` argument of `open-streamlet` was `:input` or `:io`).

- **[function]** **output-streamlet-p** *streamlet*

  See if `streamlet` was opened for input (the `direction` argument of `open-streamlet` was `:output` or `:io`).

- **[macro]** **with-open-journal** *(var journal &key (direction :input)) &body body*

  This is like `with-open-file` but for `journal`s. Open the journal designated by `journal` (see `to-journal`) with `open-streamlet`, passing `direction` along, and bind `var` to the resulting `streamlet`. Call `close-streamlet` after `body` finishes. If `journal` is `nil`, then `var` is bound to `nil` and no streamlet is created.

- **[condition]** **streamlet-error** *error*

  Like `cl:stream-error`: failures pertaining to I/O on a closed `streamlet` or of the wrong `direction`. Actual I/O errors are *not* encapsulated in `streamlet-error`.

15.2 Reading from streamlets

- **[generic-function]** **read-event** *streamlet &optional eoj-error-p*

  Read the event at the current read position from `streamlet`, and move the read position to the event after. If there are no more events, signal `end-of-journal` or return `nil` depending on `eoj-error-p`. Signals `streamlet-error` if `streamlet` is not `input-streamlet-p` or not `open-streamlet-p`.

- **[generic-function]** **read-position** *streamlet*

  Return an integer that identifies the position of the next event to be read from `streamlet`. `setf`able, see `set-read-position`.

- **[generic-function]** **set-read-position** *streamlet position*

  Set the read position of `streamlet` to `position`, which must have been acquired from `read-position`.

- **[macro]** **save-excursion** *(streamlet) &body body*

  Save `read-position` of `streamlet`, execute `body`, and make sure to restore the saved read position.

- **[generic-function]** **peek-event** *streamlet*

  Read the next event from `streamlet` without changing the read position, or return `nil` if there is no event to be read.

- **[method]** **peek-event** *(streamlet streamlet)*

  This is a slow default implementation, which relies on `save-excursion` and `read-event`.

15.3 Writing to streamlets

- **[generic-function]** **write-event** *event streamlet*

  Write `event` to `streamlet`. Writing always happens at the end of `streamlet`'s journal regardless of the `read-position`, and the read position is not changed. Signals `streamlet-error` if `streamlet` is not `output-streamlet-p` or not `open-streamlet-p`.

- **[method]** **write-event** *event (journal journal)*

  For convenience, it is possible to write directly to a `journal`, in which case the journal's internal output streamlet is used. This internal streamlet is opened for `:output` and may be used by `:log-record`.

- **[generic-function]** **write-position** *streamlet*

  Return an integer that identifies the position of the next event to be written to `streamlet`.

- **[generic-function]** **request-completed-on-abort** *streamlet*

  Make it so that upon aborted execution, `streamlet`'s `journal` will be in `journal-state` `:completed` when loaded fresh (e.g. when creating a `file-journal` with an existing

file). Any previously written events must be persisted before making this change. Before `request-completed-on-abort` is called, a journal must be reloaded in state `:failed`.

It is permissible to defer carrying out this request until the next `sync-streamlet` call. If the request was carried out, return true. If it was deferred, return `nil`.

- **[generic-function]** **sync-streamlet** *streamlet*

  Durably persist the effects of all preceding `write-event` calls made via `streamlet` to its journal and any deferred `request-completed-on-abort` in this order.

16 Glossary

- **[glossary-term]** **async-unwind**

  If an asynchronous event, say a `sigint` triggered by `C-c`, is delivered to a thread running Lisp or foreign code called from Lisp, a Lisp condition is typically signalled. If the handler for this condition unwinds the stack, then we have an asynchronous unwind. Another example is `bt:interrupt-thread`, which, as it can execute arbitrary code, may unwind the stack in the target thread.

- **[glossary-term]** **boolean-valued symbol**

  Imagine writing two `stream`s with a spaghetti of functions and wanting to have pretty-printed output on one of them. Unfortunately, binding `*print-pretty*` to `t` will affect writes to both streams.

  One solution would be to have streams look up their own print-pretty flag with `(symbol-value (stream-pretty-print stream))` and have the caller specify the dynamic variable they want:

```
(defvar *print-pretty-1* nil)
(setf (stream-print-pretty stream-1) '*print-pretty-1*)
(let ((*print-pretty-1* t))
  (spaghetti stream-1 stream-2))
```

  Note that if the default `stream-print-pretty` is `'*print-pretty*`, then we have the normal Common Lisp behaviour. Setting `stream-print-pretty` to `nil` or `t` also works, because they are self-evaluating.

  The above hypothetical example demonstrates the concept of boolean-valued symbols on `cl:stream`s. In Journal, they are used by `make-log-decorator` and `pprint-journal`s.

- **[glossary-term]** **readable**

  In Common Lisp, readable objects are those that can be printed readably. Anything written to stream-based journals needs to be readable.

17 Indices

Referrer definition type abbreviations:

- *f*: for definitions in the function namespace (macros, compiler macros and also methods)
- *t*: DEFTYPEs, 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

17.1 Function and Macro Index

17.2 Variable and Constant Index

17.3 Type Index

17.4 Misc Index

17.5 Concept Index