persistent-variables

Warren Wilkinson

February 3, 2013

Contents

1 Overview

Persistent variables can be serialized to and from streams.

They come with niceties, like restarts for failed serialization, late binding (loaded values are cached until the variable is defined), and a test suite.

```
(defpvar *my-account-password*) ;; Don't give it a value in the source file!

;; At the Repl:
(setf *my-account-password* "the-password")
(with-open-file (s "/path/to/somewhere/else")
  (p-save s))

;; Later, at startup:
(with-open-file (s "/path/to/somewhere/else")
  (p-load s))
```

1.1 Features

- Group variables into 'sets' that are saved and loaded independently.
- Late binding of variables: load you the saved values before loading your code.
- Ignores deleted/missing symbols and unbound symbols.
- Errors for serialization and deserialization problems, with nice restarts.
- Low line of code count (around 100 lines of code)

2 Installation

2.1 Quick Lisp

Install Quick Lisp and then run:

```
(ql:quickload 'persistent-variables)
```

If you have problems, see the support section, and you may want to run the tests.

2.2 Gentoo

As root,

```
emerge persistent-variables
```

Once the emerge is finished, the package can be loaded using ASDF:

```
(asdf:operate 'asdf:load-op :persistent-variables)
```

If you have problems, see the support section, otherwise you may want to run the tests.

2.3 Ubunto

```
sudo apt-get install persistent-variables
```

Once the installation is finished, the package is loadable using ASDF:

```
(asdf:operate 'asdf:load-op :persistent-variables)
```

If you have problems, see the support section, otherwise you may want to run the tests.

2.4 Manual Installation

In summary: Untar the .tar package and then symlink the .asd files into a place where ASDF can find them.

1. Untar the files where you want them to be. On windows download the .zip and unzip it instead, it's the same files.
2. ASDF could be looking anywhere – it depends on your setup. Run this in your lisp repl to get a clue as to where ASDF is seeking libraries[1]:
   ```
   (mapcan #'funcall asdf:*default-source-registries*)
   ```
3. Symlink the .asd files to the source directory. If you use windows, these instructions on symlink alternatives apply to you.

[1] you might need to (require 'asdf) before running this example

Once the files are in place, the package can be loaded with ASDF by:

```
(asdf:operate 'asdf:load-op :persistent-variables)
```

If you have problems, see the support section. If you don't have problems you may want to run the tests anyway, because you can.

2.5 Running the Tests

Once the system is loaded, it can be tested with asdf.

```
(asdf:operate 'asdf:test-op :persistent-variables)
```

This should display something like the following. There should be **zero failures**, if you have failures see the support section of this document.

```
RUNNING PERSISTENT-VARIABLE TESTS...
PERSISTENT-VARIABLE TEST RESULTS:
     Tests: 8
   Success: 8
  Failures: 0
```

2.6 Getting Support

You can find support on this libraries website and/or github repository. Or you can email Warren Wilkinson.

3 Implementation

3.1 persist - tracking variables to persist.

Persisted variables are tracked as a lists. There can be multiple lists, so the lists themselves are stored in a hash table.

```
(defvar *persisted* (make-hash-table))
(defvar *default-set* :default)

(defun persist (name &optional (set *default-set*))
  "Add a variable to persistence serialization."
  (push name (gethash set *persisted* nil)))

(defun unpersist (name &optional (set *default-set*))
  "Remove a variable from persistence serialization."
  (setf (gethash set *persisted*)
        (remove name (gethash set *persisted*))))
```

3.2 pv-save - saving variables to a stream

Variables are written as (“package” “name” “readable value”). This avoids using symbols or packages, which may not exist when we come to load the data again.

To save, p-save iterates all the variables in the set. If the variable has a package (e.g. it’s not in a package that was deleted) and has a value, it gets serialized.

```
(flet ((serialize (var)
         (list (package-name (symbol-package var))
               (symbol-name var)
               (let ((*package* (find-package :cl-user))
                     (*print-readably* t))
                 (prin1-to-string (symbol-value var))))))
  (defun pv-save (stream &optional (set *default-set*))
    "Save all defpvar values to stream."
    (dolist (var (remove-duplicates (gethash set *persisted*)))
      (when (and (symbol-package var) (boundp var))
        (prin1 (serialize var) stream)
        (terpri stream)))))
```

3.3 pv-load - loading variables from a stream

Loading variables is more complicated: things can go wrong.

- What if the variable no longer exists?[2]
- What if the data is not readable? See next.

[2] For missing variables, we cache the value in case the variable shows up in the future. For unreadable data, we present skip-variable and use-value as restarts.

```
(define-condition unloadable-variable (error)
  ((name :initarg :name :reader name)
   (text :initarg :text :reader text)
   (expression :initarg :expression :reader expression)))

(defmethod print-object ((c unloadable-variable) stream)
  (format stream "Unloadable variable ~s: ~a in ~s"
          (let ((*package* (find-package :cl-user)))
            (prin1-to-string (name c)))
          (text c)
          (expression c)))

(defun pv-read (symbol value)
  "Attempt to read a saved value."
  (restart-case
      (handler-case (values
                     (let ((*package* (find-package :cl-user)))
                       (read-from-string value))
                     t)
        (error (e) (let* ((msg (princ-to-string e))
                          (msg (subseq msg 0 (position #\Newline msg))))
                     (error 'unloadable-variable
                            :name symbol :text msg :expression value))))
    (skip-variable ()
      :report "Skip loading this variable."
      (values nil nil))
    (use-value (value)
      :report "Specify a value to use."
      :interactive (lambda ()
                     (format t "~&Value for ~s: " symbol)
                     (list (eval (read))))
      (values value t))))
```

p-load processes the stream, reading in saved variables.

For each variable, p-lead attempts to 1) find the variable and 2) set it. Variables that can't be found (and thus can't be set) are stored as a list placed into the hash table loaded.

```
(defvar *loaded* (make-hash-table)
  "Store loaded values that are missing their corresponding variables.")

(flet ((pv-set (package symbol value)
         "Attempt to set package:symbol to value. Return t if done right."
         (let* ((p (find-package package))
                (s (and p (find-symbol symbol p))))
           (multiple-value-bind (val found-p)
               (and s (pv-read s value))
             (when found-p (set s val) t)))))
  (defun pv-load (stream &optional (set *default-set*))
    "Load variable bindings from stream and set persistent-variables.."
    (loop for (package symbol value) = (read stream nil '(:eof :eof :eof))
       until (eq package :eof)
       for did-set? = (pv-set package symbol value)
       unless did-set?
       collect (list package symbol value) into bindings
       finally (setf (gethash (symbol-name set) *loaded*) bindings))))
```

3.4 *defpvar – easily define and register persistent variables*

The macro defpvar defines a variable, registers it for persistence, and loads any cached value it may already have.

It's important to notice that defpvar also **forgets** the cached value. This is so that defpvars first take on any cached loaded value, and only upon re-evalution take the value present in the source code. In summary:

1. On the first evaluation, defpvar's take the **loaded** value, regardless of what is specified.

2. On subsequent evaluations, defpvar's take their **written** default.

This applies only to late binding – when you've already loaded the saved variables before loading your code. If you load code first, none of this matters: the variables are set to the loaded data.

```
(defmacro defpvar (name
                   &optional
                     (val ''unbind)
                     (doc nil doc-supplied-p)
                     (set '*default-set*))
  "Define persistent variable, it'll take it's cached value if available."
  (let ((pset (gensym))  (value (gensym))  (found-p (gensym)))
    `(let ((,pset ,set))
       (defvar ,name
         (multiple-value-bind (,value ,found-p)
             (cached-string *package* ',name ,pset)
           (if ,found-p (pv-read ',name ,value) ,val))
         ,@(if doc-supplied-p (list doc)))

       (when (eq (symbol-value ',name) 'unbind) (makunbound ',name))
       (persist ',name ,pset)
       (cached-string-forget ',name ,pset)
       ',name)))
```

The extra functions defpvar refers to, *cached-value-forget* and *cached-value*, are defined as:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (flet ((load-eq (a b)
           (and (string-equal (first a) (first b))
                (string-equal (second a) (second b)))))
    (defun cached-string-forget (symbol
                                 &optional (set *default-set*))
      (setf (gethash (symbol-name set) *loaded*)
            (remove (list (package-name (symbol-package symbol))
                          (symbol-name symbol))
                    (gethash (symbol-name set) *loaded*)
                    :test #'load-eq)))

    (defun cached-string (package name
                          &optional (set *default-set*))
      (let ((bind (find (list (package-name (find-package package))
                              (symbol-name name))
                        (gethash (symbol-name set) *loaded*)
                        :test #'load-eq)))
        (values (third bind) (not (null bind)))))))
```

4 *Tests*

4.1 *Test Framework*

The test framework deals with the running of tests and printing of results. The tests are discussed in their own subheadings.

```
(defstruct results
  (tests 0)
  (failures nil))
(defun results-failure-count (results)
  (length (results-failures results)))
(defun results-successes (results)
  (- (results-tests results)
     (results-failure-count results)))

(defun runtest (fun results)
  (let* ((success t)
         (output (with-output-to-string (*standard-output*)
                   (unwind-protect
                        (setf success (handler-case (funcall fun)
                                        (error (e) (princ e) nil)))))))
    (make-results
     :tests (1+ (results-tests results))
     :failures (if success
                   (results-failures results)
                   (acons fun output (results-failures results))))))

(defun present-failures (results)
  (format t "~%PERSISTENT-VARIABLES FAILURES:~%")
  (loop for (fn . problems) in (results-failures results)
        do (format t "~%~a~a~%" fn problems)))
(defun present-results (results)
  (format t "~%PERSISTENT-VARIABLES TEST RESULTS:")
  (format t "~%     Tests: ~a~%   Success: ~a~%  Failures: ~a"
          (results-tests results)
          (results-successes results)
          (results-failure-count results))
  (when (results-failures results)
    (present-failures results)))

(defun run-tests ()
  (format t "~%RUNNING PERSISTENT-VARIABLES TESTS...")
  (present-results
   (reduce #'(lambda (results function) (runtest function results))
           *tests* :initial-value (make-results))))
```

Tests are just functions, pushed onto a list.

```
(defvar *tests* nil)
(defvar *success*)
(defmacro deftest (name () &rest body)
  `(progn (defun ,name () ,@body) (pushnew ',name *tests*)))

(defmacro expect (code)
  `(or ,code
      (progn
        (setf *success* nil)
        (format t ,(format nil "~%   unexpected false in:~%    ~s" code)))))
```

4.2 Test Context

Tests work in their own persistence package. They test compile-time, load-time and execute-time semantics, so they use and compile a temporary file to do that.

```
(defvar *temp-file* #p"/tmp/persister-test.lisp")
(defvar *temp-fasl* #p"/tmp/persister-test.fasl")

(defmacro defptest (name () &rest body)
  (let ((compile (assoc :compile body)))
    `(deftest ,name ()
       (remhash 'test persistent-variables::*persisted*)
       (remhash 'test-1 persistent-variables::*persisted*)
       (remhash 'test-2 persistent-variables::*persisted*)
       (when (find-package :persistent-variables.test.workspace)
         (delete-package :persistent-variables.test.workspace))
       (let ((*default-set* 'test)
             (persistent-variables::*loaded* (make-hash-table :test #'equalp))
             (*success* t))
         (declare (special *default-set* *success*
                           persistent-variables::*loaded*))
         (unwind-protect
              (progn
                ;; If there is compile time stuff, compile it.
                (with-open-file (s *temp-file* :direction :output :if-exists
                                   :supersede :if-does-not-exist :create)
                  (write-sequence
                   ,(prin1-to-string
                     `(defpackage :persistent-variables.test.workspace
                        (:use :cl :persistent-variables
                              :persistent-variables.test))) s)
                  (write-sequence
                   ,(prin1-to-string
                     `(in-package :persistent-variables.test.workspace)) s)
                  ,@(mapcar #'(lambda (code) `(write-sequence
                                          ,(prin1-to-string code) s))
                            (cdr compile)))
                (compile-file *temp-file* :output-file *temp-fasl*)

                ;; Now load it and eval/run the execute statements
                ,@(mapcan
                   #'(lambda (execute)
                       `((eval
                          '(progn
                            (load *temp-fasl*)
                            (let ((*standard-output* *standard-output*)
                                  (*package*
                                   (find-package
                                    :persistent-variables.test.workspace)))
                              (eval (read-from-string
                                     ,(prin1-to-string
                                       `(progn ,@(cdr execute)))))))))
                   (remove :execute body :key #'car :test-not #'eq))
                *success*)
           (delete-package :persistent-variables.test.workspace))))))
```

4.3 *Test p-vars are definable*

This test ensures pvars that are defined at compile time, show up, with correct values, when the file is loaded.

Then it ensures pvars that are defined during execution time have their correct values.

```
(defptest p-vars-are-definable ()
  (:compile
   (defpvar *compile-time-unbound*)
   (defpvar *compile-time-bound* :bound)
   (defpvar *compile-time-documented* :documented "documentation"))
  (:execute
   ;; Still around after loading?
   (expect (handler-case *compile-time-unbound*
              (unbound-variable () t)))
   (expect (handler-case (eq *compile-time-bound* :bound)
              (error () nil)))
   (expect (handler-case (eq *compile-time-documented* :documented)
              (error () nil)))
   (expect (handler-case (null (boundp '*compile-time-unbound*))
             (error () nil)))
   (expect (handler-case (not (null (boundp '*compile-time-bound*)))
             (error () nil)))
   (expect (handler-case (not (null (boundp '*compile-time-documented*)))
             (error () nil)))

   ;; How about some new variables?
   (defpvar *eval-time-unbound*)
   (defpvar *eval-time-bound* :ev-bound)
   (defpvar *eval-time-documented* :ev-documented "documentation")

   (expect (handler-case *eval-time-unbound*
             (unbound-variable () t)))
   (expect (handler-case (eq *eval-time-bound* :ev-bound)
             (error () nil)))
   (expect (handler-case (eq *eval-time-documented* :ev-documented)
             (error () nil)))
   (expect (handler-case (null (boundp '*eval-time-unbound*))
             (error () nil)))
   (expect (handler-case (not (null (boundp '*eval-time-bound*)))
             (error () nil)))
   (expect (handler-case (not (null (boundp '*eval-time-documented*)))
             (error () nil)))))
```

4.4 *Test p-vars can be saved and loaded*

Test pv-load and pv-save work in a simple case.

In this test we use 2 sets of variables. We save each set, setf them all set to nonsense and then load the values.

We then test to ensure they've successfully loaded their old values.

Also note that unbound variables do not save/restore their unbound state. So they keep their new value!

```
(defptest test-p-vars-can-be-saved-and-loaded ()
  (:compile
   (let ((*default-set* 'test-1))
     (declare (special *default-set*))
     (defpvar *compile-time-1-unbound*)
     (defpvar *compile-time-1-bound* :bound-1)
     (defpvar *compile-time-1-documented* :documented-1 "documentation"))
   (defpvar *compile-time-1-packaged* :packaged-1 "documentation" 'test-1)

   (let ((*default-set* 'test-2))
     (declare (special *default-set*))
     (defpvar *compile-time-2-unbound*)
     (defpvar *compile-time-2-bound* :bound-2)
     (defpvar *compile-time-2-documented* :documented-2 "documentation"))
   (defpvar *compile-time-2-packaged* :packaged-2 "documentation" 'test-2))
  (:execute

   (let ((*default-set* 'test-1))
     (declare (special *default-set*))
     (defpvar *eval-time-1-unbound*)
     (defpvar *eval-time-1-bound*      :ev-bound-1)
     (defpvar *eval-time-1-documented* :ev-documented-1 "documentation"))
   (defpvar *eval-time-1-packaged*   :ev-packaged-1 "documentation" 'test-1)

   (let ((*default-set* 'test-2))
     (declare (special *default-set*))
     (defpvar *eval-time-2-unbound*)
     (defpvar *eval-time-2-bound*      :ev-bound-2)
     (defpvar *eval-time-2-documented* :ev-documented-2 "documentation"))
   (defpvar *eval-time-2-packaged*   :ev-packaged-2 "documentation" 'test-2)

   (let ((saved-1 (with-output-to-string (saved-1)
                   (pv-save saved-1 'test-1)))
         (saved-2 (with-output-to-string (saved-2)
                   (pv-save saved-2 'test-2))))
     (setf *compile-time-1-unbound* :a-new-value
           *compile-time-1-bound* :a-new-value
           *compile-time-1-documented* :a-new-value
           *compile-time-1-packaged* :a-new-value)
     (setf *eval-time-1-unbound* :a-new-value
           *eval-time-1-bound* :a-new-value
           *eval-time-1-documented* :a-new-value
           *eval-time-1-packaged* :a-new-value)
     (setf *compile-time-2-unbound* :a-new-value
           *compile-time-2-bound* :a-new-value
           *compile-time-2-documented* :a-new-value
           *compile-time-2-packaged* :a-new-value)
     (setf *eval-time-2-unbound* :a-new-value
           *eval-time-2-bound* :a-new-value
           *eval-time-2-documented* :a-new-value
           *eval-time-2-packaged* :a-new-value)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(expect (and (eq *compile-time-1-unbound* :a-new-value)
             (eq *compile-time-1-bound* :a-new-value)
             (eq *compile-time-1-documented* :a-new-value)
             (eq *compile-time-1-packaged* :a-new-value)))
(expect (and (eq *eval-time-1-unbound* :a-new-value)
             (eq *eval-time-1-bound* :a-new-value)
             (eq *eval-time-1-documented* :a-new-value)
             (eq *eval-time-1-packaged* :a-new-value)))
(with-input-from-string (s saved-1) (pv-load s 'test-1))
(expect (and (eq *compile-time-1-unbound* :a-new-value)
             (eq *compile-time-1-bound* :bound-1)
             (eq *compile-time-1-documented* :documented-1)
             (eq *compile-time-1-packaged* :packaged-1)))
(expect (and (eq *eval-time-1-unbound* :a-new-value)
             (eq *eval-time-1-bound* :ev-bound-1)
             (eq *eval-time-1-documented* :ev-documented-1)
             (eq *eval-time-1-packaged* :ev-packaged-1)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(expect (and (eq *compile-time-2-unbound* :a-new-value)
             (eq *compile-time-2-bound* :a-new-value)
             (eq *compile-time-2-documented* :a-new-value)
             (eq *compile-time-2-packaged* :a-new-value)))
(expect (and (eq *eval-time-2-unbound* :a-new-value)
             (eq *eval-time-2-bound* :a-new-value)
             (eq *eval-time-2-documented* :a-new-value)
             (eq *eval-time-2-packaged* :a-new-value)))
(with-input-from-string (s saved-2) (pv-load s 'test-2))
(expect (and (eq *compile-time-2-unbound* :a-new-value)
             (eq *compile-time-2-bound* :bound-2)
             (eq *compile-time-2-documented* :documented-2)
             (eq *compile-time-2-packaged* :packaged-2)))
(expect (and (eq *eval-time-2-unbound* :a-new-value)
             (eq *eval-time-2-bound* :ev-bound-2)
             (eq *eval-time-2-documented* :ev-documented-2)
             (eq *eval-time-2-packaged* :ev-packaged-2))))))
```

4.5 Test p-vars can load things from different packages

This test ensures that if our variable, my-package:my-variable, saves other-package:other-symbol, it doesn't come back as my-package:other-symbol.

We test that by setting our variable to (intern "A-SYMBOL" :cl-user), and ensuring that we get that value back.

```
(defptest test-p-vars-can-load-from-different-packages ()
  (:compile
   (defpvar *compile-time-1-bound* (intern "A-SYMBOL" :cl-user)))
  (:execute
   (let ((saved-1 (with-output-to-string (saved-1) (pv-save saved-1))))
     (setf *compile-time-1-bound* :a-new-value)
     (expect (eq *compile-time-1-bound* :a-new-value))
     (with-input-from-string (s saved-1) (pv-load s))
     (expect (eq *compile-time-1-bound* (intern "A-SYMBOL" :cl-user))))))
```

4.6 Test p-vars can be late bound

This test ensures that late binding works.

In this test we save 2 sets of variable and we update them all and save them.

Then we delete the package – and load the data.

When we execute again (and the package is reloaded), we test to ensure that all the variables have their loaded values.

Finally, we makunbound some of the symbols and run a copy of their definition code again (simulating a second evaluation of them, perhaps via slime). This second time around, they should **not** use the loaded values, but instead the in-code values.

```
(defptest test-p-vars-can-be-late-bound ()
  (:compile
   (let ((*default-set* 'test-1))
     (declare (special *default-set*))
     (defpvar *compile-time-1-unbound*)
     (defpvar *compile-time-1-bound* :bound-1)
     (defpvar *compile-time-1-documented* :documented-1 "documentation"))
   (defpvar *compile-time-1-packaged* :packaged-1 "documentation" 'test-1)

   (let ((*default-set* 'test-2))
     (declare (special *default-set*))
     (defpvar *compile-time-2-unbound*)
     (defpvar *compile-time-2-bound* :bound-2)
     (defpvar *compile-time-2-documented* :documented-2 "documentation"))
   (defpvar *compile-time-2-packaged* :packaged-2 "documentation" 'test-2))

  (:execute
   (format t "~%ROUND ONE")
   (let ((*default-set* 'test-1))
     (declare (special *default-set*))
     (defpvar *eval-time-1-unbound*)
     (defpvar *eval-time-1-bound*      :ev-bound-1)
     (defpvar *eval-time-1-documented* :ev-documented-1 "documentation"))
   (defpvar *eval-time-1-packaged*   :ev-packaged-1 "documentation" 'test-1)
```

```
 (let ((*default-set* 'test-2))
   (declare (special *default-set*))
   (defpvar *eval-time-2-unbound*)
   (defpvar *eval-time-2-bound*      :ev-bound-2)
   (defpvar *eval-time-2-documented* :ev-documented-2 "documentation"))
 (defpvar *eval-time-2-packaged*   :ev-packaged-2 "documentation" 'test-2)

 (setf *compile-time-1-unbound*    :updated-compile-time-1-unbound
       *compile-time-1-bound*      :updated-compile-time-1-bound
       *compile-time-1-documented* :updated-compile-time-1-documented
       *compile-time-1-packaged*   :updated-compile-time-1-packaged)
 (setf *eval-time-1-unbound*       :updated-eval-time-1-unbound
       *eval-time-1-bound*         :updated-eval-time-1-bound
       *eval-time-1-documented*    :updated-eval-time-1-documented
       *eval-time-1-packaged*      :updated-eval-time-1-packaged)
 (setf *compile-time-2-unbound*    :updated-compile-time-2-unbound
       *compile-time-2-bound*      :updated-compile-time-2-bound
       *compile-time-2-documented* :updated-compile-time-2-documented
       *compile-time-2-packaged*   :updated-compile-time-2-packaged)
 (setf *eval-time-2-unbound*       :updated-eval-time-2-unbound
       *eval-time-2-bound*         :updated-eval-time-2-bound
       *eval-time-2-documented*    :updated-eval-time-2-documented
       *eval-time-2-packaged*      :updated-eval-time-2-packaged)

 (let ((saved-1 (with-output-to-string (saved-1)
                  (pv-save saved-1 'test-1)))
       (saved-2 (with-output-to-string (saved-2)
                  (pv-save saved-2 'test-2))))

   ;; Delete the test package...
   (let* ((*package* (find-package :persistent-variables.test)))
     (delete-package :persistent-variables.test.workspace)

     ;; Then load the saved data
     (with-input-from-string (s saved-1) (pv-load s 'test-1))
     (with-input-from-string (s saved-2) (pv-load s 'test-2)))))

(:execute
   (format t "~%ROUND TWO: ~a" *package*)

   ;; Redeclare these guys, as if late-loaded at the REPL.
   (let ((*default-set* 'test-1))
     (declare (special *default-set*))
     (defpvar *eval-time-1-unbound*)
     (defpvar *eval-time-1-bound*      :ev-bound-1)
     (defpvar *eval-time-1-documented*
         :ev-documented-1 "documentation"))
   (defpvar *eval-time-1-packaged*
       :ev-packaged-1 "documentation" 'test-1)

   (let ((*default-set* 'test-2))
     (declare (special *default-set*))
     (defpvar *eval-time-2-unbound*)
     (defpvar *eval-time-2-bound*      :ev-bound-2)
     (defpvar *eval-time-2-documented*
         :ev-documented-2 "documentation"))
   (defpvar *eval-time-2-packaged*
       :ev-packaged-2 "documentation" 'test-2)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(format t "~%ROUND TWO first round of tests...")

(expect (and (eq *compile-time-1-unbound*
                 :updated-compile-time-1-unbound)
             (eq *compile-time-1-bound*
                 :updated-compile-time-1-bound)
             (eq *compile-time-1-documented*
                 :updated-compile-time-1-documented)
             (eq *compile-time-1-packaged*
                 :updated-compile-time-1-packaged)))

(expect (and (eq *eval-time-1-unbound*
                 :updated-eval-time-1-unbound)
             (eq *eval-time-1-bound*
                 :updated-eval-time-1-bound)
             (eq *eval-time-1-documented*
                 :updated-eval-time-1-documented)
             (eq *eval-time-1-packaged*
                 :updated-eval-time-1-packaged)))

(expect (and (eq *compile-time-2-unbound*
                 :updated-compile-time-2-unbound)
             (eq *compile-time-2-bound*
                 :updated-compile-time-2-bound)
             (eq *compile-time-2-documented*
                 :updated-compile-time-2-documented)
             (eq *compile-time-2-packaged*
                 :updated-compile-time-2-packaged)))

(expect (and (eq *eval-time-2-unbound*
                 :updated-eval-time-2-unbound)
             (eq *eval-time-2-bound*
                 :updated-eval-time-2-bound)
             (eq *eval-time-2-documented*
                 :updated-eval-time-2-documented)
             (eq *eval-time-2-packaged*
                 :updated-eval-time-2-packaged)))

;; Redeclare these guys (as if a second time,
;; explicitly with slime -- or as if deleting package and
;; reloading without reloading the data...
(makunbound '*eval-time-1-unbound*)
(makunbound '*eval-time-1-bound*)
(makunbound '*eval-time-1-documented*)
(makunbound '*eval-time-1-packaged*)
(let ((*default-set* 'test-1))
  (declare (special *default-set*))
  (defpvar *eval-time-1-unbound*)
  (defpvar *eval-time-1-bound*      :ev-bound-1)
  (defpvar *eval-time-1-documented* :ev-documented-1 "documentation"))
(defpvar *eval-time-1-packaged*   :ev-packaged-1 "documentation" 'test-1)
(makunbound '*eval-time-2-unbound*)
(makunbound '*eval-time-2-bound*)
(makunbound '*eval-time-2-documented*)
(makunbound '*eval-time-2-packaged*)
(let ((*default-set* 'test-2))
  (declare (special *default-set*))
  (defpvar *eval-time-2-unbound*)
  (defpvar *eval-time-2-bound*      :ev-bound-2)
  (defpvar *eval-time-2-documented* :ev-documented-2 "documentation"))
(defpvar *eval-time-2-packaged*   :ev-packaged-2 "documentation" 'test-2)

(format t "~%ROUND TWO second round of tests.")
(expect (and (not (boundp '*eval-time-1-unbound*))
             (eq *eval-time-1-bound*         :ev-bound-1)
             (eq *eval-time-1-documented*    :ev-documented-1)
             (eq *eval-time-1-packaged*      :ev-packaged-1)))

(expect (and (not (boundp '*eval-time-2-unbound*))
             (eq *eval-time-2-bound*         :ev-bound-2)
             (eq *eval-time-2-documented*    :ev-documented-2)
             (eq *eval-time-2-packaged*      :ev-packaged-2)))))
```

4.7 *Test p-vars ignores unbound variables*

This test ensures that saving unbound variables does nothing. The variable is just not saved.

```
(defptest test-p-vars-ignores-unbound-variables ()
  (:execute
   (defpvar *unbound-variable*)

   ;; Unbound variables don't case saving errors.
   (let ((saved (with-output-to-string (s) (pv-save s))))

     (setf *unbound-variable* :a-value)

     ;; But neither do they cause bound variables to unbind
     ;; upon loading.
     (with-input-from-string (s saved) (pv-load s))
     (expect (eq *unbound-variable* :a-value)))))
```

4.8 *Test p-vars errors on unprintable variables*

This test ensures that we get a print-not-readable error if we attempt to save a value that cannot be read back.

```
(defptest test-p-vars-errors-on-unprintable-variables ()
  (:execute
   (defpvar *unprintable-variable* #'identity)
   (expect (eq 'error (handler-case (with-output-to-string (s) (pv-save s))
                        (print-not-readable () 'error))))))
```

4.9 **TODO** Test p-vars ignores deleted variables

```
(defptest test-p-vars-ignores-deleted-variables ()
  (:execute
   (when (find-package :persistent-variables.test.temporary)
     (delete-package (find-package :persistent-variables.test.temporary)))
   (let* ((var (let ((*package*
                       (make-package
                        :persistent-variables.test.temporary
                        :use (list (find-package :persistent-variables)))))
                  (prog1 (eval
                          (read-from-string
                           "(defpvar *non-existant-variable* \"a-value\")"))
                    (delete-package *package*))))
          (saved (with-output-to-string (s) (pv-save s))))
     (expect (string-equal (symbol-value var) "a-value"))
     (setf (symbol-value var) "a-different-value")
     (expect (string-equal (symbol-value var) "a-different-value"))
     (with-input-from-string (s saved) (pv-load s))

     ;; However, the deleted variables are no longer saved.
     ;; as evidence by it's failure to lead.
     (expect (string-equal (symbol-value var) "a-different-value"))

   (let* ((var (let ((*package*
                       (make-package
                        :persistent-variables.test.temporary
                        :use (list (find-package :persistent-variables)))))
                  (eval
                   (read-from-string
                    "(defpvar *non-existant-variable* \"a-third-value\")")))))
     (expect (string-equal (symbol-value var) "a-third-value"))
     (with-input-from-string (s saved) (pv-load s))
     ;; Still no change, because the variable was never saved.
     (expect (string-equal (symbol-value var) "a-third-value"))
     (delete-package *package*)))))
```

4.10 Test p-vars provides loading restarts

Save a non readable object (a function) and then load it. It should give us two restarts (on every lisp implementation).

But the is SBCL because it depends on the non-standard sb-ext:print-unreadably restart.

```
#+sbcl
(defptest test-p-vars-provides-loading-restarts ()
  (:compile
   (defpvar *bad-var-1* #'identity)
   (defpvar *bad-var-2* #'identity)
   (defvar *storage*
     (handler-bind ((print-not-readable
                     #'(lambda (e) (declare (ignore e))
                          (invoke-restart
                           'sb-ext:print-unreadably))))
       (with-output-to-string (s) (pv-save s)))))
  (:execute
   (makunbound '*bad-var-1*)
   (makunbound '*bad-var-2*)
   (handler-bind ((unloadable-variable
                   #'(lambda (e)
                       (if (eq (name e) '*bad-var-1*)
                           (invoke-restart
                            'use-value :a-new-value)
                           (invoke-restart
                            'skip-variable)))))
     (with-input-from-string (s *storage*) (pv-load s))
     (expect (eq *bad-var-1* :a-new-value))
     (expect (not (boundp '*bad-var-2*))))))
```

5 *License*