

Re: Question about design, defmacro, macrolet, and &environment

Source: <http://coding.derkeiler.com/Archive/Lisp/comp.lang.lisp/2003-11/1604.html>

From: Damien Kick (*dkick1_at_email.mot.com*)

Date: 11/16/03

Date: 15 Nov 2003 18:26:40 -0600

Peter Seibel <peter@javamonkey.com> writes:

> [...] I have no idea what a program that uses *EXPECT* is going to
> look like (1) and what it would do (2).

FWIW, this is that with which I've been playing as an excuse to get some more experience with Common Lisp. As always, I would appreciate any feedback as c.l.l is the only place I've any chance to meet lispniks.

```
;;; First version of my excuse to play with CMU CL, using to
;;; implement Don Libe's Expect. *SPAWN* is similiar to Expect's
;;; "spawn_id". EXPECT-STRING is a stupified version of Expect's
;;; "expect" as it does not allow for regular expressions. Also,
;;; none of this allows for controlling multiple "spawn"s or
;;; "expect"ing from more than one "spawn".
```

```
(defpackage "LONE-TRIP-INVESTMENTS"
  (nicknames "LTI")
  (use "COMMON-LISP" "EXTENSIONS")
  (export "WITH-SPAWN-ID" "WITH-SPAWN-STREAM" "*SPAWN*"
          "*TRACE-EXPECT-STRING-LEMMA*" "SPAWN" "EXPECT-STRING"
          "EXPECT-SOMEONE-SOMEWHERE-PROMPT"))
```

```
(in-package "LONE-TRIP-INVESTMENTS")
```

```
;;; Helper/utility functions; i.e. generally not exported but rather
;;; only used internally.
```

```
(declaim (inline synonym-value))
(defun synonym-value (symbol synonym value &key (test #'eql))
  "If the value of the symbol is a synonym (or abbreviation) for some
other value, return the real value. Otherwise, just use the value of
the symbol."
  ;; Should any of this attempt to be 'setf'able? In other words,
  ;; somehow make use of generalized variables?
```

comp.lang.lisp: Re: Question about design, defmacro, macrolet, and &environment

```
;; (setf (synonym-value #|something|#) #|something-else|#)
;; But I wouldn't know what that should look like.
(if (funcall test symbol synonym) value symbol))
```

;;; The rest of the stuff...

```
(defmacro with-spawn-process ((id exec-name . exec-args) &body code)
  `(let ((id (spawn ,exec-name ',exec-args)))
      (unwind-protect
        (progn ,@code)
        (process-close ,id))))
```

```
(defmacro with-spawn-stream ((bi exec-name . exec-args) &body code)
  (let ((id (gensym "SPAWN-PROCESS-")))
    `(with-spawn-process (,id ,exec-name ,@exec-args)
      (let ((bi (process-pty ,id)))
        ,@code))))
```

```
(defvar *spawn* nil)
(defvar *trace-expect-string-lemma* nil)
```

```
(defun spawn (name &optional args)
  "The CMU CL implementation of Don Libes spawn."
  (run-program name args :wait nil :pty t :input t :output t :error t))
```

```
(defun expect-string (string &optional stream &key (echo t))
  "Expects to find the literal string on the stream."
  (labels
    ((lemma (string in end match echo pos)
      (when *trace-expect-string-lemma*
        (format t "expect-string lemma ~A~%"
          (list string in end match echo pos)))
      (unless (= pos end)
        (let ((actual (read-char in))
              (expect (char string pos)))
          (write-char actual match)
          (when echo
            (write-char actual echo)
            (force-output echo))
          (let ((next (if (char= actual expect) (1+ pos) 0)))
            (lemma string in end match echo next))))))
    (values string
      (let ((in (or stream (process-pty *spawn*)))
            (end (length string))
            (echo (synonym-value echo t *standard-output*)))
        (with-output-to-string (match)
          (lemma string in end match echo 0))))))
```

```
(defun expect-someone-somewhere-prompt (&optional stream)
  "Expects to find Hashim's prompt on the MGTS server."
  (let ((stream (or stream (process-pty *spawn*))))
```

comp.lang.lisp: Re: Question about design, defmacro, macrolet, and &environment

```
(expect-string "tekelec:[" stream)
(expect-string "]" stream)
(expect-string "%" stream)))
```

```
(defun send (string &optional stream)
  (let ((stream (or stream (process-pty *spawn*))))
    (write-string string stream)
    (force-output stream)
    string))
```

```
... *****
;;;
;;; Testing functions, variables, etc.
... *****
;;;
```

```
(defvar *test-user-name* "someone")
(defvar *test-password* "something")
```

```
(declaim (inline make-adjustable-string))
(defun make-adjustable-string ()
  (make-array '(0)
    :element-type 'base-char :fill-pointer 0 :adjustable t))
```

```
(declaim (inline string-cat))
(defun string-cat (&rest args)
  (apply #'concatenate 'string args))
```

```
(declaim (inline make-test-expect-string))
(defun make-test-expect-string ()
  "telnet somewhere
Trying 136.182.32.250...
Connected to somewhere.
Escape character is '^]'.

```

SunOS 5.6

```
login: kick
Password: ")
```

```
(declaim (inline make-expected-match-string))
(defun make-expected-match-string ()
  "telnet somewhere
Trying 136.182.32.250...
Connected to somewhere.
Escape character is '^]'.

```

SunOS 5.6

```
login:")
```

```
(declaim (inline make-expected-expect-string))
(defun make-expected-expect-string ()
```

```
"login:")
```

```
(defun call-expect-string-tester (f)
  (let ((expect "login:"))
    (with-input-from-string (phake-spawn-in (make-test-expect-string))
      (multiple-value-bind (what-was-expected what-was-matched)
        (funcall f expect phake-spawn-in)
        (assert (string= what-was-expected
                        (make-expected-expect-string)))
        (assert (string= what-was-matched
                        (make-expected-match-string)))))))
```

```
(defun test-expect-string ()
  (call-expect-string-tester
   #'(lambda (string stream)
        (expect-string string stream :echo nil)))
  (assert (string= (with-output-to-string (phake-echo-stream)
                    (call-expect-string-tester
                     #'(lambda (string stream)
                           (expect-string string stream
                                           :echo phake-echo-stream))))
                (make-expected-match-string)))
  (call-expect-string-tester
   #'(lambda (string stream)
        (expect-string string stream))))
t)
```

```
(defun test-telnet-somewhere (&optional (user-name *test-user-name*)
                                     (password *test-password*))
  (with-spawn-process (*spawn* "telnet" "somewhere")
    (expect-string "login:")
    (send (string-cat (string user-name) (string #\Newline)))
    (expect-string "assword:")
    (send (string-cat (string password) (string #\Newline)))
    (expect-someone-somewhere-prompt)
    (send (string-cat "ls" (string #\Newline)))
    (expect-someone-somewhere-prompt)
    t))
```

```
;;; What follows is the diff of the first draft and what I changed to
;;; support defaults differently.
```

```
--- lti.lisp.cll.bak Thu Nov 13 17:02:44 2003
```

```
+++ lti.lisp.cll Sat Nov 15 18:14:12 2003
```

```
@@ -1,16 +1,14 @@
```

```
-;;; First version of my excuse to play with CMU CL, using to
-;;; implement Don Libe's Expect. *SPAWN* is similiar to Expect's
-;;; "spawn_id". EXPECT-STRING is a stupified version of Expect's
-;;; "expect" as it does not allow for regular expressions. Also,
-;;; none of this allows for controlling multiple "spawn"s or
-;;; "expect"ing from more than one "spawn".
```

```
+;;; This is a second version that removes *SPAWN* and any code on
+;;; which it depended. Instead, I use WITH-SPAWN-STREAM to create
+;;; local functions that supply the default values of STREAM for
+;;; EXPECT and SPAWN. Somehow, this seems more cl-style (&optional
+;;; flame-war-about-what-is-cl-style) to me than using *SPAWN*.
```

```
(defpackage "LONE-TRIP-INVESTMENTS"
  (:nicknames "LTI")
  (:use "COMMON-LISP" "EXTENSIONS")
  - (:export "WITH-SPAWN-ID" "WITH-SPAWN-STREAM" "*SPAWN*"
    - "*TRACE-EXPECT-STRING-LEMMA*" "SPAWN" "EXPECT-STRING"
    - "EXPECT-SOMEONE-SOMEWHERE-PROMPT"))
  + (:export "WITH-SPAWN-ID" "WITH-SPAWN-STREAM" "*TRACE-EXPECT-STRING-LEMMA*"
    + "SPAWN" "EXPECT-STRING" "EXPECT-SOMEONE-SOMEWHERE-PROMPT"))
```

```
(in-package "LONE-TRIP-INVESTMENTS")
```

```
@@ -40,19 +38,31 @@
  (let ((id (gensym "SPAWN-PROCESS-")))
    `(with-spawn-process (,id ,exec-name ,@exec-args)
      (let ((,bi (process-pty ,id)))
        - ,@code))))
  + (flet ((expect-string (string &optional (stream ,bi) &key (echo t))
    + (expect-string string stream :echo echo))
    + (send (string &optional (stream ,bi)
    + (send string stream)))
    + ,@code))))
```

```
-(defvar *spawn* nil)
  (defvar *trace-expect-string-lemma* nil)
```

```
+(declaim (inline spawn))
  (defun spawn (name &optional args)
    "The CMU CL implementation of Don Libes spawn."
    (run-program name args :wait nil :pty t :input t :output t :error t))
```

```
-(defun expect-string (string &optional stream &key (echo t))
+;; I wonder how long it will take someone on c.l.l to propose a
+;; version of EXPECT-STRING that uses LOOP. As I've been introduced
+;; to CL by Graham's _ANSI Common Lisp_, I have been introduced to an
+;; anti-LOOP cl-view and appreciate having been introduced to a
+;; pro-LOOP view by c.l.l. However, I'm still not comfortable with
+;; LOOP and wish I knew of a tutorial for it.
```

```
+(defun expect-string (string stream &key (echo t))
  "Expects to find the literal string on the stream."
  (labels
```

```
    ((lemma (string in end match echo pos)
  + ;; If/when CMU CL allows one to trace local functions, I will
  + ;; be able to remove the following WHEN.
    (when *trace-expect-string-lemma*
      (format t "expect-string lemma ~A~%"
```

```

        (list string in end match echo pos)))
@@ -66,24 +76,21 @@
        (let ((next (if (char= actual expect) (1+ pos) 0)))
          (lemma string in end match echo next))))))
      (values string
- (let ((in (or stream (process-pty *spawn*)))
- (end (length string))
+ (let ((end (length string))
          (echo (synonym-value echo t *standard-output*)))
            (with-output-to-string (match)
- (lemma string in end match echo 0))))))
+ (lemma string stream end match echo 0))))))

-(defun expect-someone-somewhere-prompt (&optional stream)
+(defun expect-someone-somewhere-prompt (stream)
  "Expects to find Hashim's prompt on the MGTS server."
- (let ((stream (or stream (process-pty *spawn*))))
- (expect-string "tekelec:[" stream)
- (expect-string "]" stream)
- (expect-string "%" stream))
-
-(defun send (string &optional stream)
- (let ((stream (or stream (process-pty *spawn*))))
- (write-string string stream)
- (force-output stream)
- string))
+ (expect-string "tekelec:[" stream)
+ (expect-string "]" stream)
+ (expect-string "%" stream))
+
+(defun send (string stream)
+ (write-string string stream)
+ (force-output stream)
+ string)

;;; *****
;;; Testing functions, variables, etc.
@@ -157,12 +164,19 @@

```

```

(defun test-telnet-somewhere (&optional (user-name *test-user-name*)
                             (password *test-password*))
- (with-spawn-process (*spawn* "telnet" "somewhere")
+ (with-spawn-stream (stream "telnet" "somewhere")
  (expect-string "login:")
  (send (string-cat (string user-name) (string #\Newline)))
  (expect-string "assword:")
  (send (string-cat (string password) (string #\Newline)))
- (expect-someone-somewhere-prompt)
+ ;; It would be nice to be able to somehow get WITH-SPAWN-STREAM to
+ ;; generate FLET versions of functions, like
+ ;; EXPECT-SOMEONE-SOMEWHERE-PROMPT, that take STREAM as an

```

comp.lang.lisp: Re: Question about design, defmacro, macrolet, and &environment

```
+ ;; optional parameter, as is done with EXPECT and SEND. I wonder
+ ;; how much extra work would be required to accomplish this for
+ ;; any function, FOO. Or would it be better to do this with
+ ;; closures?
+ (expect-someone-somewhere-prompt stream)
  (send (string-cat "ls" (string #\Newline)))
- (expect-someone-somewhere-prompt)
+ (expect-someone-somewhere-prompt stream)
  t))
```

```
;;; Applying these changes, I get a second version that removes
;;; *SPAWN* and any code on which it depended. Instead, I use
;;; WITH-SPAWN-STREAM to create local functions that supply the
;;; default values of STREAM for EXPECT and SPAWN. Somehow, this
;;; seems more cl-style (&optional flame-war-about-what-is-cl-style)
;;; to me than using *SPAWN*.
```

```
(defpackage "LONE-TRIP-INVESTMENTS"
  (nicknames "LTI")
  (use "COMMON-LISP" "EXTENSIONS")
  (export "WITH-SPAWN-ID" "WITH-SPAWN-STREAM" "*TRACE-EXPECT-STRING-LEMMA*"
         "SPAWN" "EXPECT-STRING" "EXPECT-SOMEONE-SOMEWHERE-PROMPT"))
```

```
(in-package "LONE-TRIP-INVESTMENTS")
```

```
;;; Helper/utility functions; i.e. generally not exported but rather
;;; only used internally.
```

```
(declaim (inline synonym-value))
(defun synonym-value (symbol synonym value &key (test #'eql))
  "If the value of the symbol is a synonym (or abbreviation) for some
other value, return the real value. Otherwise, just use the value of
the symbol."
  ;; Should any of this attempt to be 'setf'able? In other words,
  ;; somehow make use of generalized variables?
  ;; (setf (synonym-value #|something|#) #|something-else|#)
  ;; But I wouldn't know what that should look like.
  (if (funcall test symbol synonym) value symbol))
```

```
;;; The rest of the stuff...
```

```
(defmacro with-spawn-process ((id exec-name . exec-args) &body code)
  `(let ((,id (spawn ,exec-name ',exec-args)))
    (unwind-protect
      (progn ,@code)
      (process-close ,id))))
```

```
(defmacro with-spawn-stream ((bi exec-name . exec-args) &body code)
  (let ((id (gensym "SPAWN-PROCESS-")))
    `(with-spawn-process (,id ,exec-name ,@exec-args)
      (let ((,bi (process-pty ,id)))
```

comp.lang.lisp: Re: Question about design, defmacro, macrolet, and &environment

```
(flet ((expect-string (string &optional (stream ,bi) &key (echo t))
      (expect-string string stream :echo echo))
      (send (string &optional (stream ,bi))
            (send string stream))))
,@code))))
```

```
(defvar *trace-expect-string-lemma* nil)
```

```
(declaim (inline spawn))
```

```
(defun spawn (name &optional args)
```

```
  "The CMU CL implementation of Don Libes spawn."
```

```
  (run-program name args :wait nil :pty t :input t :output t :error t))
```

```
;; I wonder how long it will take someone on c.l.l to propose a
;; version of EXPECT-STRING that uses LOOP. As I've been introduced
;; to CL by Graham's _ANSI Common Lisp_, I have been introduced to an
;; anti-LOOP cl-view and appreciate having been introduced to a
;; pro-LOOP view by c.l.l. However, I'm still not comfortable with
;; LOOP and wish I knew of a tutorial for it.
```

```
(defun expect-string (string stream &key (echo t))
```

```
  "Expects to find the literal string on the stream."
```

```
  (labels
```

```
    ((lemma (string in end match echo pos)
```

```
      ;; If/when CMU CL allows one to trace local functions, I will
```

```
      ;; be able to remove the following WHEN.
```

```
      (when *trace-expect-string-lemma*
```

```
        (format t "expect-string lemma ~A~%"
```

```
              (list string in end match echo pos))))
```

```
    (unless (= pos end)
```

```
      (let ((actual (read-char in))
```

```
            (expect (char string pos))))
```

```
        (write-char actual match)
```

```
        (when echo
```

```
          (write-char actual echo)
```

```
          (force-output echo))
```

```
        (let ((next (if (char= actual expect) (1+ pos) 0)))
```

```
          (lemma string in end match echo next))))))
```

```
  (values string
```

```
        (let ((end (length string))
```

```
              (echo (synonym-value echo t *standard-output*)))
```

```
              (with-output-to-string (match)
```

```
                (lemma string stream end match echo 0))))))
```

```
(defun expect-someone-somewhere-prompt (stream)
```

```
  "Expects to find Hashim's prompt on the MGTS server."
```

```
  (expect-string "tekelec:[" stream)
```

```
  (expect-string "]" stream)
```

```
  (expect-string "% " stream))
```

```
(defun send (string stream)
```

```
  (write-string string stream))
```

```
(force-output stream)
string)
```

```
... *****
;;;
;;; Testing functions, variables, etc.
... *****
;;;
```

```
(defvar *test-user-name* "someone")
(defvar *test-password* "something")
```

```
(declare (inline make-adjustable-string))
(defun make-adjustable-string ()
  (make-array '(0)
              :element-type 'base-char :fill-pointer 0 :adjustable t))
```

```
(declare (inline string-cat))
(defun string-cat (&rest args)
  (apply #'concatenate 'string args))
```

```
(declare (inline make-test-expect-string))
(defun make-test-expect-string ()
  "telnet somewhere
Trying 136.182.32.250...
Connected to somewhere.
Escape character is '^'.
```

SunOS 5.6

```
login: kick
Password: ")
```

```
(declare (inline make-expected-match-string))
(defun make-expected-match-string ()
  "telnet somewhere
Trying 136.182.32.250...
Connected to somewhere.
Escape character is '^'.
```

SunOS 5.6

```
login:")
```

```
(declare (inline make-expected-expect-string))
(defun make-expected-expect-string ()
  "login:")
```

```
(defun call-expect-string-tester (f)
  (let ((expect "login:"))
    (with-input-from-string (phake-spawn-in (make-test-expect-string))
      (multiple-value-bind (what-was-expected what-was-matched)
        (funcall f expect phake-spawn-in))
```

comp.lang.lisp: Re: Question about design, defmacro, macrolet, and &environment

```
(assert (string= what-was-expected
              (make-expected-expect-string)))
(assert (string= what-was-matched
              (make-expected-match-string))))))

(defun test-expect-string ()
  (call-expect-string-tester
   #'(lambda (string stream)
        (expect-string string stream :echo nil)))
  (assert (string= (with-output-to-string (phake-echo-stream)
                    (call-expect-string-tester
                     #'(lambda (string stream)
                           (expect-string string stream
                                           :echo phake-echo-stream))))
                  (make-expected-match-string)))
  (call-expect-string-tester
   #'(lambda (string stream)
        (expect-string string stream)))
  t)

(defun test-telnet-somewhere (&optional (user-name *test-user-name*)
                              (password *test-password*))
  (with-spawn-stream (stream "telnet" "somewhere")
    (expect-string "login:")
    (send (string-cat (string user-name) (string #\Newline)))
    (expect-string "assword:")
    (send (string-cat (string password) (string #\Newline)))
    ;; It would be nice to be able to somehow get WITH-SPAWN-STREAM to
    ;; generate FLET versions of functions, like
    ;; EXPECT-SOMEONE-SOMEWHERE-PROMPT, that take STREAM as an
    ;; optional parameter, as is done with EXPECT and SEND. I wonder
    ;; how much extra work would be required to accomplish this for
    ;; any function, FOO. Or would it be better to do this with
    ;; closures?
    (expect-someone-somewhere-prompt stream)
    (send (string-cat "ls" (string #\Newline)))
    (expect-someone-somewhere-prompt stream)
    t))
```