Common Lisp Special Forms

author: Jon L White
files: [Eris]<Lisp>...Library> -- CMLSPECIALFORMS
documentation: CMLSPECIALFORMS.tedit/press
created: 13-Jan-84 by JonL
modified: 5-Oct-84, 14-Nov-84, and 18-Dec-84 by JonL

Macros are provided for the following Common Lisp constructs (so-called "special forms"): LET, LET*, PROG*, PSETQ, PROGV, LIST*, DEFUN, CATCH, THROW,
*CATCH, *THROW, and UNWINDPROTECT. Additionally, for efficiency in the interpreter, there are functional definitions for CATCH, THROW, and LIST*. In so far as possible, the syntax of these constructs follows that described in the reference manual "Common Lisp: The Language", published in 1984 by Digital Press.

LET is an alternative to LAMBDA, which puts the bindings "up front" like PROG does; e.g.,

```
   (LET ((X (MUMBLE))
         (Y (GRUMBLE)))
     (DO.FIRST.THING)
     (DO.LAST.THING))
```

expands into

```
   ((LAMBDA (X Y)
            (DO.FIRST.THING)
            (DO.LAST.THING))
      (MUMBLE)
      (GRUMBLE))
```

LET* is similar, but the bindings happen sequentially rather than in parallel. E.g.

```
   (LET* ((X (MUMBLE))
          (Y (GRUMBLE)))
     (DO.FIRST.THING)
     (DO.LAST.THING))
```

expands into

```
   ((LAMBDA (X)
            (LAMBDA (Y)
                    (DO.FIRST.THING)
                    (DO.LAST.THING))
              (GRUMBLE))
      (MUMBLE))
```

PROG* is the similar extension to PROG, namely the bindings happen "sequentially" rather than in parallel. Likewise, PSETQ is a "parallel SETQ", using the Common Lisp syntax which permits numerous assignments to appear in one "call" to PSETQ:

```
    (PSETQ A (MUMBLE X)
           B A)
```

will assign to A the value of (MUMBLE X) and assign to B the value which A has before the call to PSETQ.

PROGV provides a means for lambda-binding a list of variables:

```
   (PROGV <var-list> <val-list>
      ... <body>...)
```

is like PROGN except that the first two items in the "arglist" are evaluated to obtain a list of variable names and a list of values; then the variables are bound (as SPECVARS) to the corresponding values, and the forms in the body executed, with the value of the last one being the return value of the PROGV. This functions exists primarily as an aid in writing Lisp-like interpreters in Lisp.

LIST* is similar to LIST except that the last argument is the final CDR of the resultant list rather than the last element:
    (LIST 'A 'B 'C) -> (A B C)
    (LIST* 'A 'B 'C) -> (A B . C)

DEFUN is, at first glance, an alternative syntax for DEFINEQ:
  (DEFUN FOO (X Y Z) ...)   <==> (DEFINEQ (FOO (X Y Z) ...))
  (DEFUN BAR N ...)   <==> (DEFINEQ (BAR (LAMBDA N ...)))
Common Lisp doesn't quite permit the notions of NLAMBDAs, but for compatibility with the various Lisp dialects from which it sprang (e.g., MacLisp), the following extensions to DEFUN have been implemented
  (DEFUN BAZ FEXPR (L) ...) <==> (DEFINEQ (BAZ (NLAMBDA L ...))

Also the forms
  (DEFUN BLEH MACRO (L) ...)
  (DEFUN (BLEH MACRO) (L) ...)
are somewhat equivalent to makeing a MACRO definition for BLEH; the differences are that (1) it is a "computed" macro, and all the code body is defined under a new internal name, and (2) the argument passed to the code body via the lambda variable [in this example, L] will have the name of the macro cons'd onto the front of what would be passed to Interlisp's computed macro. The reason for this variation is that MacLisp's macros receive as argument the pointer to the cons cell that the macro expander is working on, rather than just the cdr of that cell. One additional non-standard format is implemented also:
   (DEFUN (MUMBLE GRUMBLE) (<lambda-list> ...) <codebody> ...)
will put the functional definition under a new internal name, and put a pointer to that definition as the GRUMBLE property of the litatom MUMBLE. [When compiling such a form, MacLisp (and others) also compile the "new, internal" name; but Interlisp may not be able to express this in a filepkg COMS without more development.]

CATCH provides a return point for a non-lexically initiated exit from its scope. For example
  (CATCH 'SOMETAG <form-to-eval>)
sets up a dynamic scoping for the "tag" SOMETAG, and if at any time during the execution of <form-to-eval> there is a call
  (THROW 'SOMETAG <val>)
then the CATCH will be exited with <val> as its value; but if no such THROW is executed, then the normal return value of <form-to-eval> will be the value of the CATCH. Both CATCH and THROW "evaluate their arguments", but CATCH does so in a way that the first argument, the tag, is available during the evaluation of the second argument. *CATCH and *THROW are provided as macros for compatiblity with MacLisp and Franz.

UNWINDPROTECT is very similar to RESETLST/RESETSAVE --
   (UNWINDPROTECT <form-to-eval> ... <cleanup code> ...)
will evaluate <form-to-eval>, and upon exit will execute all the remaining forms in the list -- the so-called "cleanup code". "Exiting" also means an aborting due to RESET (or HARDRESET) or any error. Unfortunately, the Interlisp RETFROM and RETTO do not currently execute the resetsave forms under a resetlst when retfrom'ing a frame higher in the stack than the one with the resetlst in it; this has the effect that a THROW to a frame higher than an UNWINDPROTECT will currently not actually do its cleanup forms.

CAVEAT: Common Lisp is lexically scoped; but Interlisp doesn't provide a mechanism capable of fully implemeting the lexical scoping inherent in PROGV, CATCH, and UNWINDPROTECT. So one must be prepared for the sort of limitations on such coding that is encountered with ERSETQ in Interlisp.