Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

*.* 2: Return to Lisp Toplevel.

*.* Debug> :CONTINUE 1

*.* Use value: 3.7

*.* Error: The value of X, 3.7, is neither an integer nor a symbol.

*.* To continue, type :CONTINUE followed by an option number:

*.* 1: Specify a value to use instead.

*.* 2: Return to Lisp Toplevel.

*.* Debug> :CONTINUE 1

*.* Use value: 12

*!* 48

x *!* 12

**Aected By:**

**ctypecase** and **etypecase**, since they might signal an error, are potentially a↵ected by existing *handlers* and **\*debug-io\***.

**Exceptional Situations:**

**ctypecase** and **etypecase** signal an error of *type* **type-error** if no *normal-clause* matches.

The *compiler* may choose to issue a warning of *type* **style-warning** if a *clause* will never be selected because it is completely shadowed by earlier clauses.

**See Also:**

**case**, **cond**, **setf**, Section 5.1 (Generalized Reference)

**Notes:**

(typecase *test-key*

*{*(*type {form}*\*)*}*\*)

*⌘*

(let ((#1=#:g0001 *test-key*))

(cond *{*((typep #1# ’*type*) *{form}*\*)*}*\*))

The specific error message used by **etypecase** and **ctypecase** can vary between implementations. In situations where control of the specific wording of the error message is important, it is better to use **typecase** with an *otherwise-clause* that explicitly signals an error with an appropriate message.

**multiple-value-bind** *Macro*

**Syntax:**

**multiple-value-bind** (*{var}*\*) *values-form {declaration}*\* *{form}*\*

*! {result}*\*

Data and Control Flow **5–79**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**Arguments and Values:**

*var*—a *symbol* naming a variable; not evaluated.

*values-form*—a *form*; evaluated.

*declaration*—a **declare** *expression*; not evaluated.

*forms*—an *implicit progn*.

*results*—the *values* returned by the *forms*.

**Description:**

Creates new variable *bindings* for the *vars* and executes a series of *forms* that use these *bindings*. The variable *bindings* created are lexical unless **special** declarations are specified.

*Values-form* is evaluated, and each of the *vars* is bound to the respective value returned by that *form*. If there are more *vars* than values returned, extra values of **nil** are given to the remaining *vars*. If there are more values than *vars*, the excess values are discarded. The *vars* are bound to the values over the execution of the *forms*, which make up an implicit **progn**. The consequences are unspecified if a type *declaration* is specified for a *var*, but the value to which that *var* is bound is not consistent with the type *declaration*.

The *scopes* of the name binding and *declarations* do not include the *values-form*.

**Examples:**

(multiple-value-bind (f r)

(floor 130 11)

(list f r)) *!* (11 9)

**See Also:**

**let**, **multiple-value-call**

**Notes:**

(multiple-value-bind (*{var}*\*) *values-form {form}*\*)

*⌘* (multiple-value-call #’(lambda (&optional *{var}*\* &rest #1=#:ignore)

(declare (ignore #1#))

*{form}*\*)

*values-form*)

**5–80** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**multiple-value-call** *Special Operator*

**Syntax:**

**multiple-value-call** *function-form form*\* *! {result}*\*

**Arguments and Values:**

*function-form*—a *form*; evaluated to produce *function*.

*function*—a *function designator* resulting from the evaluation of *function-form*.

*form*—a *form*.

*results*—the *values* returned by the *function*.

**Description:**

Applies *function* to a *list* of the *objects* collected from groups of *multiple values*2.

**multiple-value-call** first evaluates the *function-form* to obtain *function*, and then evaluates each *form*. All the values of each *form* are gathered together (not just one value from each) and given as arguments to the *function*.

**Examples:**

(multiple-value-call #’list 1 ’/ (values 2 3) ’/ (values) ’/ (floor 2.5))

*!* (1 / 2 3 / / 2 0.5)

(+ (floor 5 3) (floor 19 4)) *⌘* (+ 1 4)

*!* 5

(multiple-value-call #’+ (floor 5 3) (floor 19 4)) *⌘* (+ 1 2 4 3)

*!* 10

**See Also:**

**multiple-value-list**, **multiple-value-bind**

**multiple-value-list** *Macro*

**Syntax:**

**multiple-value-list** *form ! list*

**Arguments and Values:**

*form*—a *form*; evaluated as described below.

*list*—a *list* of the *values* returned by *form*.

Data and Control Flow **5–81**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**Description:**

**multiple-value-list** evaluates *form* and creates a *list* of the *multiple values*2 it returns. **Examples:**

(multiple-value-list (floor -3 4)) *!* (-1 1)

**See Also:**

**values-list**, **multiple-value-call**

**Notes:**

**multiple-value-list** and **values-list** are inverses of each other.

(multiple-value-list form) *⌘* (multiple-value-call #’list form)

**multiple-value-prog1** *Special Operator*

**Syntax:**

**multiple-value-prog1** *first-form {form}*\* *! first-form-results*

**Arguments and Values:**

*first-form*—a *form*; evaluated as described below.

*form*—a *form*; evaluated as described below.

*first-form-results*—the *values* resulting from the *evaluation* of *first-form*.

**Description:**

**multiple-value-prog1** evaluates *first-form* and saves all the values produced by that *form*. It then evaluates each *form* from left to right, discarding their values.

**Examples:**

(setq temp ’(1 2 3)) *!* (1 2 3)

(multiple-value-prog1

(values-list temp)

(setq temp nil)

(values-list temp)) *!* 1, 2, 3

**See Also:**

**prog1**

**5–82** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**multiple-value-setq**

**multiple-value-setq** *Macro*

**Syntax:**

**multiple-value-setq** *vars form ! result*

**Arguments and Values:**

*vars*—a *list* of *symbols* that are either *variable names* or *names* of *symbol macros*.

*form*—a *form*.

*result*—The *primary value* returned by the *form*.

**Description:**

**multiple-value-setq** assigns values to *vars*.

The *form* is evaluated, and each *var* is *assigned* to the corresponding *value* returned by that *form*. If there are more *vars* than *values* returned, **nil** is *assigned* to the extra *vars*. If there are more *values* than *vars*, the extra *values* are discarded.

If any *var* is the *name* of a *symbol macro*, then it is *assigned* as if by **setf**. Specifically, (multiple-value-setq (*symbol*1 ... *symboln*) *value-producing-form*)

is defined to always behave in the same way as

(values (setf (values *symbol*1 ... *symboln*) *value-producing-form*))

in order that the rules for order of evaluation and side-e↵ects be consistent with those used by **setf**. See Section 5.1.2.3 (VALUES Forms as Places).

**Examples:**

(multiple-value-setq (quotient remainder) (truncate 3.2 2)) *!* 1

quotient *!* 1

remainder *!* 1.2

(multiple-value-setq (a b c) (values 1 2)) *!* 1

a *!* 1

b *!* 2

c *!* NIL

(multiple-value-setq (a b) (values 4 5 6)) *!* 4

a *!* 4

b *!* 5

**See Also:**

**setq**, **symbol-macrolet**

Data and Control Flow **5–83**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**values** *Accessor*

**Syntax:**

**values** &rest *object ! {object}*\*

**(setf (values** &rest *place***)** *new-values***)**

**Arguments and Values:**

*object*—an *object*.

*place*—a *place*.

*new-value*—an *object*.

**Description:**

**values** returns the *objects* as *multiple values*2.

**setf** of **values** is used to store the *multiple values*2 *new-values* into the *places*. See Section 5.1.2.3 (VALUES Forms as Places).

**Examples:**

(values) *! hno valuesi*

(values 1) *!* 1

(values 1 2) *!* 1, 2

(values 1 2 3) *!* 1, 2, 3

(values (values 1 2 3) 4 5) *!* 1, 4, 5

(defun polar (x y)

(values (sqrt (+ (\* x x) (\* y y))) (atan y x))) *!* POLAR

(multiple-value-bind (r theta) (polar 3.0 4.0)

(vector r theta))

*!* #(5.0 0.927295)

Sometimes it is desirable to indicate explicitly that a function returns exactly one value. For example, the function

(defun foo (x y)

(floor (+ x y) y)) *!* FOO

returns two values because **floor** returns two values. It may be that the second value makes no sense, or that for eciency reasons it is desired not to compute the second value. **values** is the standard idiom for indicating that only one value is to be returned:

**5–84** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

(defun foo (x y)

(values (floor (+ x y) y))) *!* FOO

This works because **values** returns exactly one value for each of *args*; as for any function call, if any of *args* produces more than one value, all but the first are discarded.

**See Also:**

**values-list**, **multiple-value-bind**, **multiple-values-limit**, Section 3.1 (Evaluation)

**Notes:**

Since **values** is a *function*, not a *macro* or *special form*, it receives as *arguments* only the *primary values* of its *argument forms*.

**values-list** *Function*

**Syntax:**

**values-list** *list ! {element}*\*

**Arguments and Values:**

*list*—a *list*.

*elements*—the *elements* of the *list*.

**Description:**

Returns the *elements* of the *list* as *multiple values*2.

**Examples:**

(values-list nil) *! hno valuesi*

(values-list ’(1)) *!* 1

(values-list ’(1 2)) *!* 1, 2

(values-list ’(1 2 3)) *!* 1, 2, 3

**Exceptional Situations:**

Should signal **type-error** if its argument is not a *proper list*.

**See Also:**

**multiple-value-bind**, **multiple-value-list**, **multiple-values-limit**, **values**

**Notes:**

(values-list *list*) *⌘* (apply #’values *list*)

(equal *x* (multiple-value-list (values-list *x*))) returns *true* for all *lists x*.

Data and Control Flow **5–85**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**multiple-values-limit** *Constant Variable*

**Constant Value:**

An *integer* not smaller than 20, the exact magnitude of which is *implementation-dependent*.

**Description:**

The upper exclusive bound on the number of *values* that may be returned from a *function*, bound or assigned by **multiple-value-bind** or **multiple-value-setq**, or passed as a first argument to **nth-value**. (If these individual limits might di↵er, the minimum value is used.)

**See Also:**

**lambda-parameters-limit**, **call-arguments-limit**

**Notes:**

Implementors are encouraged to make this limit as large as possible.

**nth-value** *Macro*

**Syntax:**

**nth-value** *n form ! object*

**Arguments and Values:**

*n*—a non-negative *integer* ; evaluated.

*form*—a *form*; evaluated as described below.

*object*—an *object*.

**Description:**

Evaluates *n* and then *form*, returning as its only value the *n*th value *yielded* by *form*, or **nil** if *n* is greater than or equal to the number of *values* returned by *form*. (The first returned value is numbered 0.)

**Examples:**

(nth-value 0 (values ’a ’b)) *!* A

(nth-value 1 (values ’a ’b)) *!* B

(nth-value 2 (values ’a ’b)) *!* NIL

(let\* ((x 83927472397238947423879243432432432)

(y 32423489732)

**5–86** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

(a (nth-value 1 (floor x y)))

(b (mod x y)))

(values a b (= a b)))

*!* 3332987528, 3332987528, *true*

**See Also:**

**multiple-value-list**, **nth**

**Notes:**

Operationally, the following relationship is true, although **nth-value** might be more ecient in some *implementations* because, for example, some *consing* might be avoided.

(nth-value *n form*) *⌘* (nth *n* (multiple-value-list *form*))

**prog, prog***⇤ Macro*

**Syntax:**

**prog** (*{var |* (*var* [*init-form*])*}*\*) *{declaration}*\* *{tag | statement}*\*

*! {result}*\*

**prog\*** (*{var |* (*var* [*init-form*])*}*\*) *{declaration}*\* *{tag | statement}*\*

*! {result}*\*

**Arguments and Values:**

*var*—variable name.

*init-form*—a *form*.

*declaration*—a **declare** *expression*; not evaluated.

*tag*—a *go tag*; not evaluated.

*statement*—a *compound form*; evaluated as described below.

*results*—**nil** if a *normal return* occurs, or else, if an *explicit return* occurs, the *values* that were transferred.

**Description:**

Three distinct operations are performed by **prog** and **prog\***: they bind local variables, they permit use of the **return** statement, and they permit use of the **go** statement. A typical **prog** looks like this:

(prog (var1 var2 (var3 init-form-3) var4 (var5 init-form-5))

*{declaration}*\*

Data and Control Flow **5–87**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**prog, prog***⇤*

statement1

tag1

statement2

statement3

statement4

tag2

statement5

...

)

For **prog**, *init-forms* are evaluated first, in the order in which they are supplied. The *vars* are then bound to the corresponding values in parallel. If no *init-form* is supplied for a given *var*, that *var* is bound to **nil**.

The body of **prog** is executed as if it were a **tagbody** *form*; the **go** statement can be used to transfer control to a *tag*. *Tags* label *statements*.

**prog** implicitly establishes a **block** named **nil** around the entire **prog** *form*, so that **return** can be used at any time to exit from the **prog** *form*.

The di↵erence between **prog\*** and **prog** is that in **prog\*** the *binding* and initialization of the *vars* is done *sequentially*, so that the *init-form* for each one can use the values of previous ones.

**Examples:**

(prog\* ((y z) (x (car y)))

(return x))

returns the *car* of the value of z.

(setq a 1) *!* 1

(prog ((a 2) (b a)) (return (if (= a b) ’= ’/=))) *!* /=

(prog\* ((a 2) (b a)) (return (if (= a b) ’= ’/=))) *!* =

(prog () ’no-return-value) *!* NIL

(defun king-of-confusion (w)

"Take a cons of two lists and make a list of conses.

Think of this function as being like a zipper."

(prog (x y z) ;Initialize x, y, z to NIL

(setq y (car w) z (cdr w))

loop

(cond ((null y) (return x))

((null z) (go err)))

rejoin

(setq x (cons (cons (car y) (car z)) x))

(setq y (cdr y) z (cdr z))

(go loop)

err

**5–88** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

(cerror "Will self-pair extraneous items"

"Mismatch - gleep! ~S" y)

(setq z y)

(go rejoin))) *!* KING-OF-CONFUSION

This can be accomplished more perspicuously as follows:

(defun prince-of-clarity (w)

"Take a cons of two lists and make a list of conses.

Think of this function as being like a zipper."

(do ((y (car w) (cdr y))

(z (cdr w) (cdr z))

(x ’() (cons (cons (car y) (car z)) x)))

((null y) x)

(when (null z)

(cerror "Will self-pair extraneous items"

"Mismatch - gleep! ~S" y)

(setq z y)))) *!* PRINCE-OF-CLARITY

**See Also:**

**block**, **let**, **tagbody**, **go**, **return**, Section 3.1 (Evaluation)

**Notes:**

**prog** can be explained in terms of **block**, **let**, and **tagbody** as follows:

(prog *variable-list declaration* . *body*)

*⌘* (block nil (let *variable-list declaration* (tagbody . *body*)))

**prog1, prog2** *Macro*

**Syntax:**

**prog1** *first-form {form}*\* *! result-1*

**prog2** *first-form second-form {form}*\* *! result-2*

**Arguments and Values:**

*first-form*—a *form*; evaluated as described below.

*second-form*—a *form*; evaluated as described below.

*forms*—an *implicit progn*; evaluated as described below.

*result-1*—the *primary value* resulting from the *evaluation* of *first-form*.

Data and Control Flow **5–89**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**prog1, prog2**

*result-2*—the *primary value* resulting from the *evaluation* of *second-form*.

**Description:**

**prog1** *evaluates first-form* and then *forms*, *yielding* as its only *value* the *primary value yielded* by *first-form*.

**prog2** *evaluates first-form*, then *second-form*, and then *forms*, *yielding* as its only *value* the *primary value yielded* by *first-form*.

**Examples:**

(setq temp 1) *!* 1

(prog1 temp (print temp) (incf temp) (print temp))

*.* 1

*.* 2

*!* 1

(prog1 temp (setq temp nil)) *!* 2

temp *!* NIL

(prog1 (values 1 2 3) 4) *!* 1

(setq temp (list ’a ’b ’c))

(prog1 (car temp) (setf (car temp) ’alpha)) *!* A

temp *!* (ALPHA B C)

(flet ((swap-symbol-values (x y)

(setf (symbol-value x)

(prog1 (symbol-value y)

(setf (symbol-value y) (symbol-value x))))))

(let ((\*foo\* 1) (\*bar\* 2))

(declare (special \*foo\* \*bar\*))

(swap-symbol-values ’\*foo\* ’\*bar\*)

(values \*foo\* \*bar\*)))

*!* 2, 1

(setq temp 1) *!* 1

(prog2 (incf temp) (incf temp) (incf temp)) *!* 3

temp *!* 4

(prog2 1 (values 2 3 4) 5) *!* 2

**See Also:**

**multiple-value-prog1**, **progn**

**Notes:**

**prog1** and **prog2** are typically used to *evaluate* one or more *forms* with side e↵ects and return a *value* that must be computed before some or all of the side e↵ects happen.

(prog1 *{form}*\*) *⌘* (values (multiple-value-prog1 *{form}*\*))

(prog2 *form1 {form}*\*) *⌘* (let () *form1* (prog1 *{form}*\*))

**5–90** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**progn** *Special Operator*

**Syntax:**

**progn** *{form}*\* *! {result}*\*

**Arguments and Values:**

*forms*—an *implicit progn*.

*results*—the *values* of the *forms*.

**Description:**

**progn** evaluates *forms*, in the order in which they are given.

The values of each *form* but the last are discarded.

If **progn** appears as a *top level form*, then all *forms* within that **progn** are considered by the compiler to be *top level forms*.

**Examples:**

(progn) *!* NIL

(progn 1 2 3) *!* 3

(progn (values 1 2 3)) *!* 1, 2, 3

(setq a 1) *!* 1

(if a

(progn (setq a nil) ’here)

(progn (setq a t) ’there)) *!* HERE

a *!* NIL

**See Also:**

**prog1**, **prog2**, Section 3.1 (Evaluation)

**Notes:**

Many places in Common Lisp involve syntax that uses *implicit progns*. That is, part of their syntax allows many *forms* to be written that are to be evaluated sequentially, discarding the results of all *forms* but the last and returning the results of the last *form*. Such places include, but are not limited to, the following: the body of a *lambda expression*; the bodies of various control and conditional *forms* (*e.g.*, **case**, **catch**, **progn**, and **when**).

Data and Control Flow **5–91**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**define-modify-macro**

**define-modify-macro** *Macro*

**Syntax:**

**define-modify-macro** *name lambda-list function* [*documentation*] *! name*

**Arguments and Values:**

*name*—a *symbol*.

*lambda-list*—a *define-modify-macro lambda list*

*function*—a *symbol*.

*documentation*—a *string*; not evaluated.

**Description:**

**define-modify-macro** defines a *macro* named *name* to *read* and *write* a *place*.

The arguments to the new *macro* are a *place*, followed by the arguments that are supplied in *lambda-list*. *Macros* defined with **define-modify-macro** correctly pass the *environment parameter* to **get-setf-expansion**.

When the *macro* is invoked, *function* is applied to the old contents of the *place* and the *lambda-list* arguments to obtain the new value, and the *place* is updated to contain the result.

Except for the issue of avoiding multiple evaluation (see below), the expansion of a **define-modify-macro** is equivalent to the following:

(defmacro *name* (reference . *lambda-list*)

*documentation*

‘(setf ,reference

(*function* ,reference ,*arg1* ,*arg2* ...)))

where *arg1*, *arg2*, ..., are the parameters appearing in *lambda-list*; appropriate provision is made for a *rest parameter* .

The *subforms* of the macro calls defined by **define-modify-macro** are evaluated as specified in Section 5.1.1.1 (Evaluation of Subforms to Places).

*Documentation* is attached as a *documentation string* to *name* (as kind **function**) and to the *macro function*.

If a **define-modify-macro** *form* appears as a *top level form*, the *compiler* must store the *macro* definition at compile time, so that occurrences of the macro later on in the file can be expanded correctly.

**5–92** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**Examples:**

(define-modify-macro appendf (&rest args)

append "Append onto list") *!* APPENDF

(setq x ’(a b c) y x) *!* (A B C)

(appendf x ’(d e f) ’(1 2 3)) *!* (A B C D E F 1 2 3)

x *!* (A B C D E F 1 2 3)

y *!* (A B C)

(define-modify-macro new-incf (&optional (delta 1)) +)

(define-modify-macro unionf (other-set &rest keywords) union)

**Side Eects:**

A macro definition is assigned to *name*.

**See Also:**

**defsetf**, **define-setf-expander**, **documentation**, Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

**defsetf** *Macro*

**Syntax:**

The “short form”:

**defsetf** *access-fn update-fn* [*documentation*]

*! access-fn*

The “long form”:

**defsetf** *access-fn lambda-list* (*{store-variable}*\*) [[ *{declaration}*\* *| documentation* ]] *{form}*\* *! access-fn*

**Arguments and Values:**

*access-fn*—a *symbol* which names a *function* or a *macro*.

*update-fn*—a *symbol* naming a *function* or *macro*.

*lambda-list*—a *defsetf lambda list*.

*store-variable*—a *symbol* (a *variable name*).

*declaration*—a **declare** *expression*; not evaluated.

*documentation*—a *string*; not evaluated.

*form*—a *form*.

Data and Control Flow **5–93**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**defsetf**

**Description:**

**defsetf** defines how to **setf** a *place* of the form (*access-fn* ...) for relatively simple cases. (See **define-setf-expander** for more general access to this facility.) It must be the case that the *function* or *macro* named by *access-fn* evaluates all of its arguments.

**defsetf** may take one of two forms, called the “short form” and the “long form,” which are distinguished by the *type* of the second *argument*.

When the short form is used, *update-fn* must name a *function* (or *macro*) that takes one more argument than *access-fn* takes. When **setf** is given a *place* that is a call on *access-fn*, it expands into a call on *update-fn* that is given all the arguments to *access-fn* and also, as its last argument, the new value (which must be returned by *update-fn* as its value).

The long form **defsetf** resembles **defmacro**. The *lambda-list* describes the arguments of *access-fn*. The *store-variables* describe the value or values to be stored into the *place*. The *body* must compute the expansion of a **setf** of a call on *access-fn*. The expansion function is defined in the same *lexical environment* in which the **defsetf** *form* appears.

During the evaluation of the *forms*, the variables in the *lambda-list* and the *store-variables* are bound to names of temporary variables, generated as if by **gensym** or **gentemp**, that will be bound by the expansion of **setf** to the values of those *subforms*. This binding permits the *forms* to be written without regard for order-of-evaluation issues. **defsetf** arranges for the temporary variables to be optimized out of the final result in cases where that is possible.

The body code in **defsetf** is implicitly enclosed in a *block* whose name is *access-fn*

**defsetf** ensures that *subforms* of the *place* are evaluated exactly once.

*Documentation* is attached to *access-fn* as a *documentation string* of kind **setf**.

If a **defsetf** *form* appears as a *top level form*, the *compiler* must make the *setf expander* available so that it may be used to expand calls to **setf** later on in the *file*. Users must ensure that the *forms*, if any, can be evaluated at compile time if the *access-fn* is used in a *place* later in the same *file*. The *compiler* must make these *setf expanders* available to compile-time calls to **get-setf-expansion** when its *environment* argument is a value received as the *environment parameter* of a *macro*.

**Examples:**

The e↵ect of

(defsetf symbol-value set)

is built into the Common Lisp system. This causes the form (setf (symbol-value foo) fu) to expand into (set foo fu).

Note that

(defsetf car rplaca)

would be incorrect because **rplaca** does not return its last argument.

**5–94** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**defsetf**

(defun middleguy (x) (nth (truncate (1- (list-length x)) 2) x)) *!* MIDDLEGUY

(defun set-middleguy (x v)

(unless (null x)

(rplaca (nthcdr (truncate (1- (list-length x)) 2) x) v))

v) *!* SET-MIDDLEGUY

(defsetf middleguy set-middleguy) *!* MIDDLEGUY

(setq a (list ’a ’b ’c ’d)

b (list ’x)

c (list 1 2 3 (list 4 5 6) 7 8 9)) *!* (1 2 3 (4 5 6) 7 8 9)

(setf (middleguy a) 3) *!* 3

(setf (middleguy b) 7) *!* 7

(setf (middleguy (middleguy c)) ’middleguy-symbol) *!* MIDDLEGUY-SYMBOL

a *!* (A 3 C D)

b *!* (7)

c *!* (1 2 3 (4 MIDDLEGUY-SYMBOL 6) 7 8 9)

An example of the use of the long form of **defsetf**:

(defsetf subseq (sequence start &optional end) (new-sequence)

‘(progn (replace ,sequence ,new-sequence

:start1 ,start :end1 ,end)

,new-sequence)) *!* SUBSEQ

(defvar \*xy\* (make-array ’(10 10)))

(defun xy (&key ((x x) 0) ((y y) 0)) (aref \*xy\* x y)) *!* XY

(defun set-xy (new-value &key ((x x) 0) ((y y) 0))

(setf (aref \*xy\* x y) new-value)) *!* SET-XY

(defsetf xy (&key ((x x) 0) ((y y) 0)) (store)

‘(set-xy ,store ’x ,x ’y ,y)) *!* XY

(get-setf-expansion ’(xy a b))

*!* (#:t0 #:t1),

(a b),

(#:store),

((lambda (&key ((x #:x)) ((y #:y)))

(set-xy #:store ’x #:x ’y #:y))

#:t0 #:t1),

(xy #:t0 #:t1)

(xy ’x 1) *!* NIL

(setf (xy ’x 1) 1) *!* 1

(xy ’x 1) *!* 1

(let ((a ’x) (b ’y))

(setf (xy a 1 b 2) 3)

(setf (xy b 5 a 9) 14))

*!* 14

(xy ’y 0 ’x 1) *!* 1

(xy ’x 1 ’y 2) *!* 3

Data and Control Flow **5–95**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**See Also:**

**documentation**, **setf**, **define-setf-expander**, **get-setf-expansion**, Section 5.1 (Generalized Reference), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

**Notes:**

*forms* must include provision for returning the correct value (the value or values of *store-variable*). This is handled by *forms* rather than by **defsetf** because in many cases this value can be returned at no extra cost, by calling a function that simultaneously stores into the *place* and returns the correct value.

A **setf** of a call on *access-fn* also evaluates all of *access-fn*’s arguments; it cannot treat any of them specially. This means that **defsetf** cannot be used to describe how to store into a *generalized reference* to a byte, such as (ldb field reference). **define-setf-expander** is used to handle situations that do not fit the restrictions imposed by **defsetf** and gives the user additional control.

**define-setf-expander** *Macro*

**Syntax:**

**define-setf-expander** *access-fn lambda-list*

[[ *{declaration}*\* *| documentation* ]] *{form}*\*

*! access-fn*

**Arguments and Values:**

*access-fn*—a *symbol* that *names* a *function* or *macro*.

*lambda-list* – *macro lambda list*.

*declaration*—a **declare** *expression*; not evaluated.

*documentation*—a *string*; not evaluated.

*forms*—an *implicit progn*.

**Description:**

**define-setf-expander** specifies the means by which **setf** updates a *place* that is referenced by *access-fn*.

When **setf** is given a *place* that is specified in terms of *access-fn* and a new value for the *place*, it is expanded into a form that performs the appropriate update.

The *lambda-list* supports destructuring. See Section 3.4.4 (Macro Lambda Lists).

*Documentation* is attached to *access-fn* as a *documentation string* of kind **setf**.

**5–96** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**define-setf-expander**

*Forms* constitute the body of the *setf expander* definition and must compute the *setf expansion* for a call on **setf** that references the *place* by means of the given *access-fn*. The *setf expander* function is defined in the same *lexical environment* in which the **define-setf-expander** *form* appears. While *forms* are being executed, the variables in *lambda-list* are bound to parts of the *place form*. The

body *forms* (but not the *lambda-list*) in a **define-setf-expander** *form* are implicitly enclosed in a *block* whose name is *access-fn*.

The evaluation of *forms* must result in the five values described in Section 5.1.1.2 (Setf Expansions).

If a **define-setf-expander** *form* appears as a *top level form*, the *compiler* must make the *setf expander* available so that it may be used to expand calls to **setf** later on in the *file*. *Programmers* must ensure that the *forms* can be evaluated at compile time if the *access-fn* is used in a *place* later in the same *file*. The *compiler* must make these *setf expanders* available to compile-time calls to **get-setf-expansion** when its *environment* argument is a value received as the *environment parameter* of a *macro*.

**Examples:**

(defun lastguy (x) (car (last x))) *!* LASTGUY

(define-setf-expander lastguy (x &environment env)

"Set the last element in a list to the given value."

(multiple-value-bind (dummies vals newval setter getter)

(get-setf-expansion x env)

(let ((store (gensym)))

(values dummies

vals

‘(,store)

‘(progn (rplaca (last ,getter) ,store) ,store)

‘(lastguy ,getter))))) *!* LASTGUY

(setq a (list ’a ’b ’c ’d)

b (list ’x)

c (list 1 2 3 (list 4 5 6))) *!* (1 2 3 (4 5 6))

(setf (lastguy a) 3) *!* 3

(setf (lastguy b) 7) *!* 7

(setf (lastguy (lastguy c)) ’lastguy-symbol) *!* LASTGUY-SYMBOL

a *!* (A B C 3)

b *!* (7)

c *!* (1 2 3 (4 5 LASTGUY-SYMBOL))

;;; Setf expander for the form (LDB bytespec int).

;;; Recall that the int form must itself be suitable for SETF.

(define-setf-expander ldb (bytespec int &environment env)

(multiple-value-bind (temps vals stores

store-form access-form)

(get-setf-expansion int env);Get setf expansion for int.

Data and Control Flow **5–97**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

(let ((btemp (gensym)) ;Temp var for byte specifier.

(store (gensym)) ;Temp var for byte to store.

(stemp (first stores))) ;Temp var for int to store.

(if (cdr stores) (error "Can’t expand this."))

;;; Return the setf expansion for LDB as five values.

(values (cons btemp temps) ;Temporary variables.

(cons bytespec vals) ;Value forms.

(list store) ;Store variables.

‘(let ((,stemp (dpb ,store ,btemp ,access-form)))

,store-form

,store) ;Storing form.

‘(ldb ,btemp ,access-form) ;Accessing form.

))))

**See Also:**

**setf**, **defsetf**, **documentation**, **get-setf-expansion**, Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

**Notes:**

**define-setf-expander** di↵ers from the long form of **defsetf** in that while the body is being executed the *variables* in *lambda-list* are bound to parts of the *place form*, not to temporary variables that will be bound to the values of such parts. In addition, **define-setf-expander** does not have **defsetf**’s

restriction that *access-fn* must be a *function* or a function-like *macro*; an arbitrary **defmacro** destructuring pattern is permitted in *lambda-list*.

**get-setf-expansion** *Function*

**Syntax:**

**get-setf-expansion** *place* &optional *environment*

*! vars, vals, store-vars, writer-form, reader-form*

**Arguments and Values:**

*place*—a *place*.

*environment*—an *environment object*.

*vars, vals, store-vars, writer-form, reader-form*—a *setf expansion*.

**Description:**

Determines five values constituting the *setf expansion* for *place* in *environment*; see Section 5.1.1.2 (Setf Expansions).

**5–98** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

If *environment* is not supplied or **nil**, the environment is the *null lexical environment*. **Examples:**

(get-setf-expansion ’x)

*!* NIL, NIL, (#:G0001), (SETQ X #:G0001), X

;;; This macro is like POP

(defmacro xpop (place &environment env)

(multiple-value-bind (dummies vals new setter getter)

(get-setf-expansion place env)

‘(let\* (,@(mapcar #’list dummies vals) (,(car new) ,getter))

(if (cdr new) (error "Can’t expand this."))

(prog1 (car ,(car new))

(setq ,(car new) (cdr ,(car new)))

,setter))))

(defsetf frob (x) (value)

‘(setf (car ,x) ,value)) *!* FROB

;;; The following is an error; an error might be signaled at macro expansion time (flet ((frob (x) (cdr x))) ;Invalid

(xpop (frob z)))

**See Also:**

**defsetf**, **define-setf-expander**, **setf**

**Notes:**

Any *compound form* is a valid *place*, since any *compound form* whose *operator f* has no *setf expander* are expanded into a call to (setf *f* ).

**setf, psetf** *Macro*

**Syntax:**

**setf** *{#pair}*\* *! {result}*\*

**psetf** *{#pair}*\* *!* **nil**

*pair::*=*place newvalue*

Data and Control Flow **5–99**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**setf, psetf**

**Arguments and Values:**

*place*—a *place*.

*newvalue*—a *form*.

*results*—the *multiple values*2 returned by the storing form for the last *place*, or **nil** if there are no *pairs*.

**Description:**

**setf** changes the *value* of *place* to be *newvalue*.

(setf place newvalue) expands into an update form that stores the result of evaluating *newvalue* into the location referred to by *place*. Some *place* forms involve uses of accessors that take optional arguments. Whether those optional arguments are permitted by **setf**, or what their use is, is up to the **setf** expander function and is not under the control of **setf**. The documentation for any *function* that accepts **&optional**, **&rest**, or &key arguments and that claims to be usable with **setf** must specify how those arguments are treated.

If more than one *pair* is supplied, the *pairs* are processed sequentially; that is,

(setf place-1 newvalue-1

place-2 newvalue-2

...

place-N newvalue-N)

is precisely equivalent to

(progn (setf place-1 newvalue-1)

(setf place-2 newvalue-2)

...

(setf place-N newvalue-N))

For **psetf**, if more than one *pair* is supplied then the assignments of new values to places are done in parallel. More precisely, all *subforms* (in both the *place* and *newvalue forms*) that are to be evaluated are evaluated from left to right; after all evaluations have been performed, all of the assignments are performed in an unpredictable order.

For detailed treatment of the expansion of **setf** and **psetf**, see Section 5.1.2 (Kinds of Places). **Examples:**

(setq x (cons ’a ’b) y (list 1 2 3)) *!* (1 2 3)

(setf (car x) ’x (cadr y) (car x) (cdr x) y) *!* (1 X 3)

x *!* (X 1 X 3)

y *!* (1 X 3)

(setq x (cons ’a ’b) y (list 1 2 3)) *!* (1 2 3)

(psetf (car x) ’x (cadr y) (car x) (cdr x) y) *!* NIL

x *!* (X 1 A 3)

**5–100** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

y *!* (1 A 3)

**Aected By:**

**define-setf-expander**, **defsetf**, **\*macroexpand-hook\***

**See Also:**

**define-setf-expander**, **defsetf**, **macroexpand-1**, **rotatef**, **shiftf**, Section 5.1 (Generalized Reference)

**shiftf** *Macro*

**Syntax:**

**shiftf** *{place}*+ *newvalue ! old-value-1*

**Arguments and Values:**

*place*—a *place*.

*newvalue*—a *form*; evaluated.

*old-value-1*—an *object* (the old *value* of the first *place*).

**Description:**

**shiftf** modifies the values of each *place* by storing *newvalue* into the last *place*, and shifting the values of the second through the last *place* into the remaining *places*.

If *newvalue* produces more values than there are store variables, the extra values are ignored. If *newvalue* produces fewer values than there are store variables, the missing values are set to **nil**.

In the form (shiftf *place1 place2* ... *placen newvalue*), the values in *place1* through *placen* are *read* and saved, and *newvalue* is evaluated, for a total of n+1 values in all. Values 2 through n+1 are then stored into *place1* through *placen*, respectively. It is as if all the *places* form a shift register; the *newvalue* is shifted in from the right, all values shift over to the left one place, and the value shifted out of *place1* is returned.

For information about the *evaluation* of *subforms* of *places*, see Section 5.1.1.1 (Evaluation of Subforms to Places).

**Examples:**

(setq x (list 1 2 3) y ’trash) *!* TRASH

(shiftf y x (cdr x) ’(hi there)) *!* TRASH

x *!* (2 3)

y *!* (1 HI THERE)

Data and Control Flow **5–101**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

(setq x (list ’a ’b ’c)) *!* (A B C)

(shiftf (cadr x) ’z) *!* B

x *!* (A Z C)

(shiftf (cadr x) (cddr x) ’q) *!* Z

x *!* (A (C) . Q)

(setq n 0) *!* 0

(setq x (list ’a ’b ’c ’d)) *!* (A B C D)

(shiftf (nth (setq n (+ n 1)) x) ’z) *!* B

x *!* (A Z C D)

**Aected By:**

**define-setf-expander**, **defsetf**, **\*macroexpand-hook\***

**See Also:**

**setf**, **rotatef**, Section 5.1 (Generalized Reference)

**Notes:**

The e↵ect of (shiftf *place1 place2* ... *placen newvalue*) is roughly equivalent to

(let ((var1 *place1*)

(var2 *place2*)

...

(varn *placen*)

(var0 *newvalue*))

(setf *place1* var2)

(setf *place2* var3)

...

(setf *placen* var0)

var1)

except that the latter would evaluate any *subforms* of each place twice, whereas **shiftf** evaluates them once. For example,

(setq n 0) *!* 0

(setq x (list ’a ’b ’c ’d)) *!* (A B C D)

(prog1 (nth (setq n (+ n 1)) x)

(setf (nth (setq n (+ n 1)) x) ’z)) *!* B

x *!* (A B Z D)

**5–102** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**rotatef**

**rotatef** *Macro*

**Syntax:**

**rotatef** *{place}*\* *!* **nil**

**Arguments and Values:**

*place*—a *place*.

**Description:**

**rotatef** modifies the values of each *place* by rotating values from one *place* into another.

If a *place* produces more values than there are store variables, the extra values are ignored. If a *place* produces fewer values than there are store variables, the missing values are set to **nil**.

In the form (rotatef *place1 place2* ... *placen*), the values in *place1* through *placen* are *read* and *written*. Values 2 through *n* and value 1 are then stored into *place1* through *placen*. It is as if all the places form an end-around shift register that is rotated one place to the left, with the value of *place1* being shifted around the end to *placen*.

For information about the *evaluation* of *subforms* of *places*, see Section 5.1.1.1 (Evaluation of Subforms to Places).

**Examples:**

(let ((n 0)

(x (list ’a ’b ’c ’d ’e ’f ’g)))

(rotatef (nth (incf n) x)

(nth (incf n) x)

(nth (incf n) x))

x) *!* (A C D B E F G)

**See Also:**

**define-setf-expander**, **defsetf**, **setf**, **shiftf**, **\*macroexpand-hook\***, Section 5.1 (Generalized Reference)

**Notes:**

The e↵ect of (rotatef *place1 place2* ... *placen*) is roughly equivalent to

(psetf *place1 place2*

*place2 place3*

...

*placen place1*)

except that the latter would evaluate any *subforms* of each place twice, whereas **rotatef** evaluates them once.

Data and Control Flow **5–103**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**control-error** *Condition Type*

**Class Precedence List:**

**control-error**, **error**, **serious-condition**, **condition**, **t**

**Description:**

The *type* **control-error** consists of error conditions that result from invalid dynamic transfers of control in a program. The errors that result from giving **throw** a tag that is not active or from giving **go** or **return-from** a tag that is no longer dynamically available are of *type* **control-error**.

**program-error** *Condition Type*

**Class Precedence List:**

**program-error**, **error**, **serious-condition**, **condition**, **t**

**Description:**

The *type* **program-error** consists of error conditions related to incorrect program syntax. The errors that result from naming a *go tag* or a *block tag* that is not lexically apparent are of *type* **program-error**.

**undefined-function** *Condition Type*

**Class Precedence List:**

**undefined-function**, **cell-error**, **error**, **serious-condition**, **condition**, **t**

**Description:**

The *type* **undefined-function** consists of *error conditions* that represent attempts to *read* the definition of an *undefined function*.

The name of the cell (see **cell-error**) is the *function name* which was *funbound*.

**See Also:**

**cell-error-name**

**5–104** Programming Language—Common Lisp