10. FUNCTION DEFINITION, MANIPULATION, AND EVALUATION

The Interlisp programming system is designed to help the user define and debug functions. Developing an applications program in Interlisp involves defining a number of functions in terms of the system primitives and other user-defined functions. Once defined, the user's functions may be referenced exactly like Interlisp primitive functions, so the programming process can be viewed as extending the Interlisp language to include the required functionality.

Functions are defined with a list expressions known as an "expr definition." An expr definition specifies if the function has a fixed or variable number of arguments, whether these arguments are evaluated or not, the function argument names, and a series of forms which define the behavior of the function. For example:

(LAMBDA (X Y) (PRINT X) (PRINT Y))

A function defined with this expr definition would have two evaluated arguments, **X** and **Y**, and it would execute **(PRINT X)** and **(PRINT Y)** when evaluated. Other types of expr definitions are described below.

A function is defined by putting an expr definition in the function definition cell of a litatom. There are a number of functions for accessing and setting function definition cells, but one usually defines a function with **DEFINEQ** (page 10.9). For example:

← (DEFINEQ (FOO (LAMBDA (X Y) (PRINT X) (PRINT Y))))
(FOO)

The expression above will define the function **FOO** to have the expr definition **(LAMBDA (X Y) (PRINT X) (PRINT Y))**. After being defined, this function may be evaluated just like any system function:

← (FOO 3 (IPLUS 3 4))
3
7
7
←

All function definition cells do not contain expr definitions. The compiler (page 18.1) translates expr definitions into compiled code objects, which execute much faster. Interlisp provides a

number of "function type functions" which determine how a given function is defined, the number and names of function arguments, etc. See page 10.7.

Usually, functions are evaluated automatically when they appear within another function or when typed into Interlisp. However, sometimes it is useful to envoke the Interlisp interpreter explicitly to apply a given "functional argument" to some data. There are a number of functions which will apply a given function repeatedly. For example, **MAPCAR** will apply a function (or an expr definition) to all of the elements of a list, and return the values returned by the function:

← (MAPCAR '(1 2 3 4 5) '(LAMBDA (X) (ITIMES X X))
(1 4 9 16 25)

When using functional arguments, there are a number of problems which can arise, related with accessing free variables from within a function argument. Many times these problems can be solved using the function **FUNCTION** to create a **FUNARG** object (see page 10.18).

The macro facility provides another way of specifying the behavior of a function (see page 10.21). Macros are very useful when developing code which should run very quickly, which should be compiled differently than it is interpreted, or which should run differently in different implementations of Interlisp.

10.1 Function Types

Interlisp functions are defined using list expressions called "expr definitions." An expr definition is a list of the form (*LAMBDA-WORD ARG-LIST* $FORM_1$ ... $FORM_N$). *LAMBDA-WORD* determines whether the arguments to this function will be evaluated or not, *ARG-LIST* determines the number and names of arguments, and $FORM_1$ ... $FORM_N$ are a series of forms to be evaluated after the arguments are bound to the local variables in *ARG-LIST*.

If *LAMBDA-WORD* is the litatom **LAMBDA**, then the arguments to the function are evaluated. If *LAMBDA-WORD* is the litatom **NLAMBDA**, then the arguments to the function are not evaluated. Functions which evaluate or don't evaluate their arguments are therefore known as "lambda" or "nlambda" functions, respectively.

If *ARG-LIST* is **NIL** or a list of litatoms, this indicates a function with a fixed number of arguments. Each litatom is the name of an argument for the function defined by this expression. The process of binding these litatoms to the individual arguments is

called "spreading" the arguments, and the function is called a "spread" function. If the argument list is any litatom other than **NIL**, this indicates a function with a variable number of arguments, known as a "nospread" function.

If *ARG-LIST* is anything other than a litatom or a list of litatoms, such as **(LAMBDA "FOO" ...)**, attempting to use this expr definition will generate an **ARG NOT LITATOM** error. In addition, if **NIL** or **T** is used as an argument name, the error **ATTEMPT TO BIND NIL OR T** is generated.

These two parameters (lambda/nlambda and spread/nospread) may be specified independently, so there are four main function types, known as lambda-spread, nlambda-spread, lambda-nospread, and nlambda-nospread functions. Each one has a different form, and is used for a different purpose. These four function types are described more fully below.

Note: For lambda-spread, lambda-nospread, or nlambda-spread functions, there is an upper limit to the number of arguments that a function can have, based on the number of arguments that can be stored on the stack on any one function call. Currently, the limit is 80 arguments. If a function is called with more than that many arguments, the error **TOO MANY ARGUMENTS** occurs. However, nlambda-nospread functions can be called with an arbitrary number of arguments, since the arguments are not individually saved on the stack (see page 10.6).

10.1.1 Lambda-Spread Functions

Lambda-spread functions take a fixed number of evaluated arguments. This is the most common function type. A lambda-spread expr definition has the form:

(LAMBDA (ARG_1 ... ARG_M**)** $FORM_1$ ... $FORM_N$**)**

The argument list **(**ARG_1 ... ARG_M**)** is a list of litatoms that gives the number and names of the formal arguments to the function. If the argument list is **()** or **NIL**, this indicates that the function takes no arguments. When a lambda-spread function is applied to some arguments, the arguments are evaluated, and bound to the local variables ARG_1 ... ARG_M. Then, $FORM_1$ ... $FORM_N$ are evaluated in order, and the value of the function is the value of $FORM_N$.

← (DEFINEQ (FOO (LAMBDA (X Y) (LIST X Y))))
(FOO)
← (FOO 99 (PLUS 3 4))
(99 7)

In the above example, the function **FOO** defined by **(LAMBDA (X Y) (LIST X Y))** is applied to the arguments **99** and **(PLUS 3 4)**, these arguments are evaluated (giving 99 and 7), the local variable **X** is bound to 99 and **Y** to 7, **(LIST X Y)** is evaluated, returning **(99 7)**, and this is returned as the value of the function.

A standard feature of the Interlisp system is that no error occurs if a spread function is called with too many or too few arguments. If a function is called with too many arguments, the extra arguments are evaluated but ignored. If a function is called with too few arguments, the unsupplied ones will be delivered as **NIL**. In fact, a spread function cannot distinguish between being given **NIL** as an argument, and not being given that argument, e.g., **(FOO)** and **(FOO NIL)** are *exactly* the same for spread functions. If it is necessary to distinguish between these two cases, use an nlambda function and explicitly evaluate the arguments with the **EVAL** function (page 10.12).

10.1.2 Nlambda-Spread Functions

Nlambda-spread functions take a fixed number of unevaluated arguments. An nlambda-spread expr definition has the form:

(NLAMBDA (ARG_1 ... ARG_M**)** $FORM_1$ ... $FORM_N$**)**

Nlambda-spread functions are evaluated similarly to lambda-spread functions, except that the arguments are not evaluated before being bound to the variables ARG_1 ... ARG_M.

← (DEFINEQ (FOO (NLAMBDA (X Y) (LIST X Y))))
(FOO)
← (FOO 99 (PLUS 3 4))
(99 (PLUS 3 4))

In the above example, the function **FOO** defined by **(NLAMBDA (X Y) (LIST X Y))** is applied to the arguments **99** and **(PLUS 3 4)**, these arguments are bound unevaluated to **X** and **Y**, **(LIST X Y)** is evaluated, returning **(99 (PLUS 3 4))**, and this is returned as the value of the function.

Note: Functions can be defined so that all of their arguments are evaluated (lambda functions) or none are evaluated (nlambda functions). If it is desirable to write a function which only evaluates *some* of its arguments (e.g. **SETQ**), the function should be defined as an nlambda, with some arguments explicitly evaluated using the function **EVAL** (page 10.12). If this is done, the user should put the litatom **EVAL** on the property list of the function under the property **INFO**. This informs various system packages such as DWIM, CLISP, and Masterscope that this function in fact *does* evaluate its arguments, even though it is an nlambda.

Warning: A frequent problem that occurs when evaluating arguments to nlambda functions with **EVAL** (page 10.12) is that the form being evaluated may reference variables that are not accessable within the nlambda function. This is usually not a problem when interpreting code, but when the code is compiled, the values of "local" variables may not be accessable on the stack (see page 18.5). The system nlambda functions that evaluate their arguments (such as **SETQ**) are expanded in-line by the compiler, so this is not a problem. Using the macro facility (page 10.21) is recommended in cases where it is necessary to evaluate some arguments to an nlambda function.

10.1.3 Lambda-Nospread Functions

Lambda-nospread functions take a variable number of evaluated arguments. A lambda-nospread expr definition has the form:

(LAMBDA *VAR* $FORM_1$ ... $FORM_N$**)**

VAR may be any litatom, except **NIL** and **T**. When a lambda-nospread function is applied to some arguments, each of these arguments is evaluated and the values stored on the stack. *VAR* is then bound to the *number* of arguments which have been evaluated. For example, if **FOO** is defined by **(LAMBDA X ...)**, when **(FOO A B C)** is evaluated, **A**, **B**, and **C** are evaluated and **X** is bound to 3. *VAR* should *never* be reset.

The following functions are used for accessing the arguments of lambda-nospread functions:

(ARG *VAR M***)** [NLambda Function]

Returns the *M*th argument for the lambda-nospread function whose argument list is *VAR*. *VAR* is the *name* of the atomic argument list to a lambda-nospread function, and is not evaluated; *M* is the number of the desired argument, and is evaluated. The value of **ARG** is undefined for *M* less than or equal to 0 or greater than the *value* of *VAR*.

(SETARG *VAR M X***)** [NLambda Function]

Sets the *M*th argument for the lambda-nospread function whose argument list is *VAR* to *X*. *VAR* is not evaluated; *M* and *X* are evaluated. *M* should be between 1 and the value of *VAR*.

In the example below, the function **FOO** is defined to collect and return a list of all of the evaluated arguments it is given (the value of the **for** statement).

```
←(DEFINEQ (FOO
      (LAMBDA X
        (for ARGNUM from 1 to X collect (ARG X ARGNUM)]
```

```
(FOO)
← (FOO 99 (PLUS 3 4))
(99 7)
← (FOO 99 (PLUS 3 4) (TIMES 3 4))
(99 7 12)
```

10.1.4 Nlambda-Nospread Functions

Nlambda-nospread functions take a variable number of unevaluated arguments. An nlambda-nospread expr definition has the form:

(NLAMBDA *VAR* $FORM_1$ ... $FORM_N$**)**

VAR may be any litatom, except **NIL** and **T**. Though similar in form to lambda-nospread expr definitions, an nlambda-nospread is evaluated quite differently. When an nlambda-nospread function is applied to some arguments, *VAR* is simply bound to a list of the unevaluated arguments. The user may pick apart this list, and evaluate different arguments.

In the example below, **FOO** is defined to return the reverse of the list of arguments it is given (unevaluated):

```
← (DEFINEQ (FOO (NLAMBDA X (REVERSE X))))
(FOO)
← (FOO 99 (PLUS 3 4))
((PLUS 3 4) 99)
← (FOO 99 (PLUS 3 4) (TIMES 3 4))
((TIMES 3 4) (PLUS 3 4) 99)
```

Note: The warning about evaluating arguments to nlambda functions (page 10.5) also applies to nlambda-nospread function.

10.1.5 Compiled Functions

Functions defined by expr definitions can be compiled by the Interlisp compiler (page 18.1). The compiler produces compiled code objects (of data type **CCODEP**) which execute more quickly than the corresponding expr definition code. Functions defined by compiled code objects may have the same four types as expr definitions (lambda/nolambda, spread/nospread). Functions created by the compiler are referred to as compiled functions.

10.1.6 Function Type Functions

There are a variety of functions used for examining the type, argument list, etc. of functions. These functions may be given either a litatom, in which case they obtain the function

definition from the litatom's definition cell, or a function definition itself.

(FNTYP *FN*) [Function]

Returns **NIL** if *FN* is not a function definition or the name of a defined function. Otherwise **FNTYP** returns one of the following litatoms, depending on the type of function definition:

EXPR	Lambda-spread expr definition.
CEXPR	Lambda-spread compiled definition.
FEXPR	Nlambda-spread expr definition.
CFEXPR	Nlambda-spread compiled definition.
EXPR*	Lambda-nospread expr definition.
CEXPR*	Lambda-nospread compiled definition.
FEXPR*	Nlambda-nospread expr definition.
CFEXPR*	Nlambda-nospread compiled definition.
FUNARG	**FNTYP** returns the litatom **FUNARG** if *FN* is a **FUNARG** expression. See page 10.18.

EXPR, **FEXPR**, **EXPR***, and **FEXPR*** indicate that *FN* is defined by an expr definition. **CEXPR**, **CFEXPR**, **CEXPR***, and **CFEXPR*** indicate that *FN* is defined by a compiled definition, as indicated by the prefix **C**. The suffix * indicates that *FN* has an indefinite number of arguments, i.e., is a nospread functions. The prefix **F** indicates unevaluated arguments. Thus, for example, a **CFEXPR*** is a compiled nospread-nlambda function.

(EXPRP *FN*) [Function]

Returns **T** if **(FNTYP *FN*)** is either **EXPR**, **FEXPR**, **EXPR***, or **FEXPR***; **NIL** otherwise. However, **(EXPRP *FN*)** is also true if *FN* is (has) a list definition, even if it does not begin with **LAMBDA** or **NLAMBDA**. In other words, **EXPRP** is not quite as selective as **FNTYP**.

(CCODEP *FN*) [Function]

Returns **T** if **(FNTYP *FN*)** is either **CEXPR**, **CFEXPR**, **CEXPR***, or **CFEXPR***; **NIL** otherwise.

(ARGTYPE *FN*) [Function]

FN is the name of a function or its definition. **ARGTYPE** returns 0, 1, 2, or 3, or **NIL** if *FN* is not a function. The interpretation of this value is:

0	Lambda-spread function (**EXPR**, **CEXPR**)
1	Nlambda-spread function (**FEXPR**, **CFEXPR**)
2	Lambda-nospread function (**EXPR***, **CEXPR***)

3 Nlambda-nospread function (**FEXPR***, **CFEXPR***)

i.e., **ARGTYPE** corresponds to the *rows* of **FNTYP**'s.

(NARGS *FN***)** [Function]

Returns the number of arguments of *FN*, or **NIL** if *FN* is not a function. If *FN* is a nospread function, the value of **NARGS** is 1.

(ARGLIST *FN***)** [Function]

Returns the "argument list" for *FN*. Note that the "argument list" is a litatom for nospread functions. Since **NIL** is a possible value for **ARGLIST**, an error is generated, **ARGS NOT AVAILABLE**, if *FN* is not a function.

If *FN* is a compiled function, the argument list is constructed, i.e., each call to **ARGLIST** requires making a new list. For functions defined by expr definitions, lists beginning with **LAMBDA** or **NLAMBDA**, the argument list is simply **CADR** of **GETD**. If *FN* has an expr definition, and **CAR** of the definition is not **LAMBDA** or **NLAMBDA**, **ARGLIST** will check to see if **CAR** of the definition is a member of **LAMBDASPLST** (page 20.14). If it is, **ARGLIST** presumes this is a function object the user is defining via **DWIMUSERFORMS** (page 20.11), and simply returns **CADR** of the definition as its argument list. Otherwise **ARGLIST** generates an error as described above.

(SMARTARGLIST *FN EXPLAINFLG TAIL***)** [Function]

A "smart" version of **ARGLIST** that tries various strategies to get the arglist of *FN*.

First, **SMARTARGLIST** checks the property list of *FN* under the property **ARGNAMES**. For spread functions, the argument list itself is stored. For nospread functions, the form is **(NIL** $ARGLIST_1$ **.** $ARGLIST_2$**)**, where $ARGLIST_1$ is the value **SMARTARGLIST** should return when *EXPLAINFLG* = **T**, and $ARGLIST_2$ the value when *EXPLAINFLG* = **NIL**. For example, **(GETPROP 'DEFINEQ 'ARGNAMES) = (NIL (X1 XI ... XN) . X)**. This allows the user to specify special argument lists.

Second, if *FN* is not defined as a function, **SMARTARGLIST** attempts spelling correction on *FN* by calling **FNCHECK** (page 20.23), passing *TAIL* to be used for the call to **FIXSPELL**. If unsuccessful, an error will be generated, *FN* **NOT A FUNCTION**.

Third, if *FN* is known to the file package (page 17.1) but not loaded in, **SMARTARGLIST** will obtain the arglist information from the file.

Otherwise, **SMARTARGLIST** simply returns **(ARGLIST** *FN***)**.

SMARTARGLIST is used by **BREAK** (page 15.5) and **ADVISE** (page 15.11) with *EXPLAINFLG* = **NIL** for constructing equivalent expr

definitions, and by the TTYIN in-line command ?= (page 26.33), with *EXPLAINFLG* = T.

10.2 Defining Functions

Function definitions are stored in a "function definition cell" associated with each litatom. This cell is directly accessible via the two functions **PUTD** and **GETD** (page 10.11), but it is usually easier to define functions with **DEFINEQ**:

(DEFINEQ X_1 X_2 ... X_N**)** [NLambda NoSpread Function]

DEFINEQ is the function normally used for defining functions. It takes an indefinite number of arguments which are not evaluated. Each X_i must be a list defining one function, of the form (*NAME DEFINITION*). For example:

(DEFINEQ (DOUBLE (LAMBDA (X) (IPLUS X X))))

The above expression will define the function **DOUBLE** with the expr definition **(LAMBDA (X) (IPLUS X X))**. X_i may also have the form (*NAME ARGS . DEF-BODY*), in which case an appropriate lambda expr definition will be constructed. Therefore, the above expression is exactly the same as:

(DEFINEQ (DOUBLE (X) (IPLUS X X)))

Note that this alternate form can only be used for Lambda functions. The first form must be used to define an nlambda function.

DEFINEQ returns a list of the names of the functions defined.

(DEFINE *X* —**)** [Function]

Lambda-spread version of **DEFINEQ**. Each element of the list *X* is itself a list either of the form (*NAME DEFINITION*) or (*NAME ARGS . DEF-BODY*). **DEFINE** will generate an error, **INCORRECT DEFINING FORM**, on encountering an atom where a defining list is expected.

Note: **DEFINE** and **DEFINEQ** will operate correctly if the function is already defined and **BROKEN**, **ADVISED**, or **BROKEN-IN**.

For expressions involving type-in only, if the time stamp facility is enabled (page 16.76), both **DEFINE** and **DEFINEQ** will stamp the definition with the user's initials and date.

UNSAFE.TO.MODIFY.FNS [Variable]

Value is a list of functions that should not be redefined, because doing so may cause unusual bugs (or crash the system!). If the user tries to modify a function on this list (using **DEFINEQ**, **TRACE**, etc), the system will print "**Warning:** *XXX* **may be safe to modify -- continue?**" If the users types "Yes", the function is modified, otherwise an error occurs. This provides a measure of safety for novices who may accidently redefine important system functions. Users can add their own functions onto this list.

Note: By convention, all functions starting with the character backslash ("\") are system internal functions, which should never be redefined or modified by the user. Backslash functions are not on **UNSAFE.TO.MODIFY.FNS**, so trying to redefine them will not cause a warning.

DFNFLG [Variable]

DFNFLG is a global variable that effects the operation of **DEFINEQ** and **DEFINE**. If **DFNFLG = NIL**, an attempt to *redefine* a function *FN* will cause **DEFINE** to print the message **(***FN* **REDEFINED)** and to save the old definition of *FN* using **SAVEDEF** (page 17.27) before redefining it (except if the old and new definitions are **EQUAL**, inwhich case the effect is simply a no-op). If **DFNFLG = T**, the function is simply redefined. If **DFNFLG = PROP** or **ALLPROP**, the new definition is stored on the property list under the property **EXPR**. **ALLPROP** also affects the operation of **RPAQQ** and **RPAQ** (page 17.54). **DFNFLG** is initially **NIL**.

DFNFLG is reset by **LOAD** (page 17.6) to enable various ways of handling the defining of functions and setting of variables when loading a file. For most applications, the user will not reset **DFNFLG** directly.

Note: The compiler does NOT respect the value of **DFNFLG** when it redefines functions to their compiled definitions (see page 18.1). Therefore, if you set **DFNFLG** to **PROP** to completely avoid inadvertantly redefining something in your running system, you *must* use compile mode **F**, not **ST**.

Note: The functions **SAVEDEF** and **UNSAVEDEF** (page 17.27) can be useful for "saving" and restoring function definitions from property lists.

(GETD *FN***)** [Function]

Returns the function definition of *FN*. Returns **NIL** if *FN* is not a litatom, or has no definition.

GETD of a compiled function constructs a pointer to the definition, with the result that two successive calls do not

necessarily produce **EQ** results. **EQP** or **EQUAL** must be used to compare compiled definitions.

(PUTD *FN DEF* —**)** [Function]

Puts *DEF* into *FN*'s function cell, and returns *DEF*. Generates an error, **ARG NOT LITATOM,** if *FN* is not a litatom. Generates an error, **ILLEGAL ARG,** if *DEF* is a string, number, or a litatom other than **NIL**.

(MOVD *FROM TO COPYFLG* —**)** [Function]

Moves the definition of *FROM* to *TO*, i.e., redefines *TO*. If *COPYFLG* = **T**, a **COPY** of the definition of *FROM* is used. *COPYFLG* = **T** is only meaningful for expr definitions, although **MOVD** works for compiled functions as well. **MOVD** returns *TO*.

COPYDEF (page 17.27) is a higher-level function that only moves expr definitions, but works also for variables, records, etc.

(MOVD? *FROM TO COPYFLG* —**)** [Function]

If *TO* is not defined, same as **(MOVD** *FROM TO COPYFLG***)**. Otherwise, does nothing and returns **NIL**.

10.3 Function Evaluation

Usually, function application is done automatically by the Interlisp interpreter. If a form is typed into Interlisp whose **CAR** is a function, this function is applied to the arguments in the **CDR** of the form. These arguments are evaluated or not, and bound to the function parameters, as determined by the type of the function, and the body of the function is evaluated. This sequence is repeated as each form in the body of the function is evaluated.

There are some situations where it is necessary to explicitly call the evaluator, and Interlisp supplies a number of functions that will do this. These functions take "functional arguments", which may either be litatoms with function definitions, or expr definition forms such as **(LAMBDA (X) ...)**, or **FUNARG** expressions (see page 10.18).

(APPLY *FN ARGLIST* —**)** [Function]

Applies the function *FN* to the arguments in the list *ARGLIST*, and returns its value. **APPLY** is a lambda function, so its arguments are evaluated, but the individual elements of *ARGLIST* are not evaluated. Therefore, lambda and nlambda functions are treated the same by **APPLY**—lambda functions take their

arguments from *ARGLIST* without evaluating them. For example:

←(APPLY 'APPEND '((PLUS 1 2 3) (4 5 6)))
(PLUS 1 2 3 4 5 6)

Note that *FN* may explicitly evaluate one or more of its arguments itself. For example, the system function **SETQ** is an nlambda function that explicitly evaluates its second argument. Therefore, **(APPLY 'SETQ '(FOO (ADD1 3)))** will set **FOO** to 4, instead of setting it to the expression **(ADD1 3)**.

APPLY can be used for manipulating expr definitions, for example:

←(APPLY '(LAMBDA (X Y) (ITIMES X Y)) '(3 4))
12

(APPLY* *FN* ARG_1 ARG_2 ... ARG_N) [NoSpread Function]

Nospread version of **APPLY**. Applies the function *FN* to the arguments ARG_1 ARG_2 ... ARG_N. For example,

←(APPLY* 'APPEND '(PLUS 1 2 3) '(4 5 6))
(PLUS 1 2 3 4 5 6)

(EVAL *X* —) [Function]

EVAL evaluates the expression *X* and returns this value, i.e., **EVAL** provides a way of calling the Interlisp interpreter. Note that **EVAL** is itself a lambda function, so *its* argument is first evaluated, e.g.,

←(SETQ FOO '(ADD1 3))
(ADD1 3)
←(EVAL FOO)
4
←(EVAL 'FOO)
(ADD1 3)

(QUOTE *X*) [NLambda NoSpread Function]

QUOTE prevents its arguments from being evaluated. Its value is *X* itself, e.g., **(QUOTE FOO)** is **FOO**.

Interlisp functions can either evaluate or not evaluate their arguments. **QUOTE** can be used in those cases where it is desirable to specify arguments unevaluated.

Note: The character single-quote (') is defined with a read macro so it returns the next expression, wrapped in a call to **QUOTE** (page 25.42). For example, **'FOO** reads as **(QUOTE FOO)**. This is the form used for examples in this manual.

Since giving **QUOTE** more than one argument is almost always a parentheses error, and one that would otherwise go undetected,

QUOTE itself generates an error in this case, **PARENTHESIS ERROR**.

(KWOTE *X*) [Function]

Value is an expression which when evaluated yields *X*. If *X* is **NIL** or a number, this is *X* itself. Otherwise, **(LIST (QUOTE QUOTE)** *X*). For example,

(KWOTE 5) => 5

(KWOTE (CONS 'A 'B)) => (QUOTE (A . B))

(NLAMBDA.ARGS *X*) [Function]

This function interprets its argument as a list of unevaluated Nlambda arguments. If any of the elements in this list are of the form **(QUOTE ...)**, the enclosing **QUOTE** is stripped off. Actually, **NLAMBDA.ARGS** stops processing the list after the first non-quoted argument. Therefore, whereas **(NLAMBDA.ARGS '((QUOTE FOO) BAR)) -> (FOO BAR), (NLAMBDA.ARGS '(FOO (QUOTE BAR))) -> (FOO (QUOTE BAR))**.

NLAMBDA.ARGS is called by a number of nlambda functions in the system, to interpret their arguments. For instance, the function **BREAK** calls **NLAMBDA.ARGS** so that **(BREAK 'FOO)** will break the function **FOO**, rather than the function **QUOTE**.

(EVALA *X A*) [Function]

Simulates association list variable lookup. *X* is a form, *A* is a list of the form:

$((NAME_1 . VAL_1) (NAME_2 . VAL_2) \ldots (NAME_N . VAL_N))$

The variable names and values in *A* are "spread" on the stack, and then *X* is evaluated. Therefore, any variables appearing free in *X*, that also appears as **CAR** of an element of *A* will be given the value in the **CDR** of that element.

(DEFEVAL *TYPE FN*) [Function]

Specifies how a datum of a particular type is to be evaluated. Intended primarily for user defined data types, but works for all data types except lists, literal atoms, and numbers. *TYPE* is a type name. *FN* is a function object, i.e. name of a function or a lambda expression. Whenever the interpreter encounters a datum of the indicated type, *FN* is applied to the datum and its value returned as the result of the evaluation. **DEFEVAL** returns the previous evaling function for this type. If *FN* = **NIL**, **DEFEVAL** returns the current evaling function without changing it. If *FN* = **T**, the evaling function is set back to the system default (which for all data types except lists is to return the datum itself).

Note: **COMPILETYPELST** (page 18.11) permits the user to specify how a datum of a particular type is to be compiled.

(EVALHOOK *FORM EVALHOOKFN*) [Function]

EVALHOOK evaluates the expression *FORM*, and returns its value. While evaluating *FORM*, the function **EVAL** behaves in a special way. Whenever a list other than *FORM* itself is to be evaluated, whether implicitly or via an explicit call to **EVAL**, *EVALHOOKFN* is invoked (it should be a function), with the form to be evaluated as its argument. *EVALHOOKFN* is then responsible for evaluating the form; whatever is returned is assumed to be the result of evaluating the form. During the execution of *EVALHOOKFN*, this special evaluation is turned off. (Note that **EVALHOOK** does not effect the evaluations of variables, only of lists).

Here is an example of a simple tracing routine that uses the **EVALHOOK** feature:

```
←(DEFINEQ (PRINTHOOK (FORM)
   (printout T "eval: " FORM T)
   (EVALHOOK FORM (FUNCTION PRINTHOOK]
(PRINTHOOK)
```

Using **PRINTHOOK**, one might see the following interaction:

```
←(EVALHOOK '(LIST (CONS 1 2) (CONS 3 4)) 'PRINTHOOK)
eval: (CONS 1 2)
eval: (CONS 3 4)
((1 . 2) (3 . 4))
```

10.4 Iterating and Mapping Functions

The functions below are used to evaluate a form or apply a function repeatedly. **RPT**, **RPTQ**, and **FRPTQ** evaluate an expression a specified number of times. **MAP**, **MAPCAR**, **MAPLIST**, etc. apply a given function repeatedly to different elements of a list, possibly constructing another list.

These functions allow efficient iterative computations, but they are difficult to use. For programming iterative computations, it is usually better to use the CLISP Iterative Statement facility (page 9.9), which provides a more general and complete facility for expressing iterative statements. Whenever possible, CLISP translates iterative statements into expressions using the functions below, so there is no efficiency loss.

(RPT *N FORM*) [Function]

Evaluates the expression *FORM*, *N* times. Returns the value of the last evaluation. If *N* less than or equal to 0, *FORM* is not evaluated, and **RPT** returns **NIL**.

Before each evaluation, the local variable **RPTN** is bound to the number of evaluations yet to take place. This variable can be referenced within *FORM*. For example, **(RPT 10 '(PRINT RPTN))** will print the numbers 10, 9, ... 1, and return 1.

(RPTQ *N* $FORM_1$ $FORM_2$ ... $FORM_N$) [NLambda NoSpread Function]

Nlambda-nospread version of **RPT**: *N* is evaluated, $FORM_i$ are not. Returns the value of the last evaluation of $FORM_N$.

(FRPTQ *N* $FORM_1$ $FORM_2$ ... $FORM_N$) [NLambda NoSpread Function]

Faster version of **RPTQ**. Does not bind **RPTN**.

(MAP *MAPX MAPFN1 MAPFN2*) [Function]

If *MAPFN2* is **NIL**, **MAP** applies the function *MAPFN1* to successive tails of the list *MAPX*. That is, first it computes **(***MAPFN1 MAPX***)**, and then **(***MAPFN1* **(CDR** *MAPX***))**, etc., until *MAPX* becomes a non-list. If *MAPFN2* is provided, **(***MAPFN2 MAPX***)** is used instead of **(CDR** *MAPX***)** for the next call for *MAPFN1*, e.g., if *MAPFN2* were **CDDR**, alternate elements of the list would be skipped. **MAP** returns **NIL**.

(MAPC *MAPX MAPFN1 MAPFN2*) [Function]

Identical to **MAP**, except that **(***MAPFN1* **(CAR** *MAPX***))** is computed at each iteration instead of **(***MAPFN1 MAPX***)**, i.e., **MAPC** works on elements, **MAP** on tails. **MAPC** returns **NIL**.

(MAPLIST *MAPX MAPFN1 MAPFN2*) [Function]

Successively computes the same values that **MAP** would compute, and returns a list consisting of those values.

(MAPCAR *MAPX MAPFN1 MAPFN2*) [Function]

Computes the same values that **MAPC** would compute, and returns a list consisting of those values, e.g., **(MAPCAR X 'FNTYP)** is a list of **FNTYP**s for each element on **X**.

(MAPCON *MAPX MAPFN1 MAPFN2*) [Function]

Computes the same values as **MAP** and **MAPLIST** but **NCONC**s these values to form a list which it returns.

(MAPCONC *MAPX MAPFN1 MAPFN2***)** [Function]

Computes the same values as **MAPC** and **MAPCAR**, but **NCONC**s the values to form a list which it returns.

Note that **MAPCAR** creates a new list which is a mapping of the old list in that each element of the new list is the result of applying a function to the corresponding element on the original list. **MAPCONC** is used when there are a *variable* number of elements (including none) to be inserted at each iteration. Examples:

```
(MAPCONC '(A B C NIL D NIL)
    '(LAMBDA (Y) (if (NULL Y) then NIL else (LIST Y))))
 => (A B C D)
```

This **MAPCONC** returns a list consisting of *MAPX* with all **NIL**s removed.

```
(MAPCONC '((A B) C (D E F) (G) H I)
    '(LAMBDA (Y) (if (LISTP Y) then Y else NIL)))
 => (A B D E F G)
```

This **MAPCONC** returns a linear list consisting of all the lists on *MAPX*.

Since **MAPCONC** uses **NCONC** to string the corresponding lists together, in this example the original list will be altered to be **((A B D E F G) C (D E F G) (G) H I)**. If this is an undesirable side effect, the functional argument to **MAPCONC** should return instead a top level copy of the lists, i.e. **(LAMBDA (Y) (if (LISTP Y) then (APPEND Y) else NIL))**.

(MAP2C *MAPX MAPY MAPFN1 MAPFN2***)** [Function]

Identical to **MAPC** except *MAPFN1* is a function of two arguments, and (*MAPFN1* **(CAR** *MAPX***) (CAR** *MAPY***)**) is computed at each iteration. Terminates when either *MAPX* or *MAPY* is a non-list.

MAPFN2 is still a function of one argument, and is applied twice on each iteration; (*MAPFN2 MAPX*) gives the new *MAPX*, (*MAPFN2 MAPY*) the new *MAPY*. **CDR** is used if *MAPFN2* is not supplied, i.e., is **NIL**.

(MAP2CAR *MAPX MAPY MAPFN1 MAPFN2***)** [Function]

Identical to **MAPCAR** except *MAPFN1* is a function of two arguments and (*MAPFN1* **(CAR** *MAPX***) (CAR** *MAPY***)**) is used to assemble the new list. Terminates when either *MAPX* or *MAPY* is a non-list.

(SUBSET *MAPX MAPFN1 MAPFN2***)** [Function]

Applies *MAPFN1* to elements of *MAPX* and returns a list of those elements for which this application is non-**NIL**, e.g.,

(SUBSET '(A B 3 C 4) 'NUMBERP) = **(3 4)**.

MAPFN2 plays the same role as with **MAP**, **MAPC**, et al.

(EVERY *EVERYX EVERYFN1 EVERYFN2***)** [Function]

Returns **T** if the result of applying *EVERYFN1* to each element in *EVERYX* is true, otherwise **NIL**. For example, **(EVERY '(X Y Z) 'ATOM)** = > **T**.

EVERY operates by evaluating (*EVERYFN1* **(CAR** *EVERYX***)** *EVERYX*). The second argument is passed to *EVERYFN1* so that it can look at the next element on *EVERYX* if necessary. If *EVERYFN1* yields **NIL**, **EVERY** immediately returns **NIL**. Otherwise, **EVERY** computes (*EVERYFN2 EVERYX*), or **(CDR** *EVERYX***)** if *EVERYFN2* = **NIL**, and uses this as the "new" *EVERYX*, and the process continues. For example, **(EVERY** *X* **'ATOM 'CDDR)** is true if every *other* element of *X* is atomic.

(SOME *SOMEX SOMEFN1 SOMEFN2***)** [Function]

Returns the tail of *SOMEX* beginning with the first element that satisfies *SOMEFN1*, i.e., for which *SOMEFN1* applied to that element is true. Value is **NIL** if no such element exists. **(SOME X '(LAMBDA (Z) (EQUAL Z Y)))** is equivalent to **(MEMBER Y X)**. **SOME** operates analogously to **EVERY**. At each stage, (*SOMEFN1* **(CAR** *SOMEX***)** *SOMEX*) is computed, and if this is not **NIL**, *SOMEX* is returned as the value of **SOME**. Otherwise, (*SOMEFN2 SOMEX*) is computed, or **(CDR** *SOMEX***)** if *SOMEFN2* = **NIL**, and used for the next *SOMEX*.

(NOTANY *SOMEX SOMEFN1 SOMEFN2***)** [Function]

(NOT (SOME *SOMEX SOMEFN1 SOMEFN2***))**

(NOTEVERY *EVERYX EVERYFN1 EVERYFN2***)** [Function]

(NOT (EVERY *EVERYX EVERYFN1 EVERYFN2***))**

(MAPRINT *LST FILE LEFT RIGHT SEP PFN LISPXPRINTFLG***)** [Function]

A general printing function. For each element of the list *LST*, applies *PFN* to the element, and *FILE*. If *PFN* is **NIL**, **PRIN1** is used. Between each application, **MAPRINT** performs **PRIN1** of *SEP* (or " " if *SEP* = **NIL**). If *LEFT* is given, it is printed (using **PRIN1**) initially; if *RIGHT* is given it is printed (using **PRIN1**) at the end.

For example, **(MAPRINT X NIL '%('%))** is equivalent to **PRIN1** for lists. To print a list with commas between each element and a final "." one could use **(MAPRINT X T NIL '%. '%,)**.

If *LISPXPRINTFLG* = T, **LISPXPRIN1** (page 13.25) is used instead of **PRIN1**.

10.5 Functional Arguments

The functions that call the Interlisp-D evaluator take "functional arguments", which may either be litatoms with function definitions, or expr definition forms such as **(LAMBDA (X) ...)**, or **FUNARG** expressions (below).

The following functions are useful when one wants to supply a functional argument which will always return **NIL**, **T**, or 0. Note that the arguments X_1 ... X_N to these functions are evaluated, though they are not used.

(NILL X_1 ... X_N**)** [NoSpread Function]

Returns **NIL**.

(TRUE X_1 ... X_N**)** [NoSpread Function]

Returns **T**.

(ZERO X_1 ... X_N**)** [NoSpread Function]

Returns 0.

When using expr definitions as functional arguments, they should be enclosed within the function **FUNCTION** rather than **QUOTE**, so that they will be compiled as separate functions. **FUNCTION** can also be used to create **FUNARG** expressions, which can be used to solve some problems with referencing free variables, or to create functional arguments which carry "state" along with them.

(FUNCTION *FN ENV***)** [NLambda Function]

If *ENV* = **NIL**, **FUNCTION** is the same as **QUOTE**, except that it is treated differently when compiled. Consider the function definition:

```
(DEFINEQ (FOO (LST)
        (FIE LST (FUNCTION (LAMBDA (Z) (ITIMES Z Z))]
```

FOO calls the function **FIE** with the value of **LST** and the expr definition **(LAMBDA (Z) (LIST (CAR Z)))**.

If **FOO** is run interpreted, it doesn't make any difference whether **FUNCTION** or **QUOTE** is used. However, when **FOO** is compiled, if **FUNCTION** is used the compiler will define and compile the expr

definition as an auxiliary function (See page 18.10). The compiled expr definition will run considerably faster, which can make a big difference if it is applied repeatedly.

Note: Compiling **FUNCTION** will *not* create an auxiliary function if it is a functional argument to a function that compiles open, such as most of the mapping functions (**MAPCAR**, **MAPLIST**, etc.).

If *ENV* is not **NIL**, it can be a list of variables that are (presumably) used freely by *FN*. In this case, the value of **FUNCTION** is an expression of the form **(FUNARG** *FN POS***)**, where *POS* is a stack pointer to a frame that contains the variable bindings for those variables on *ENV*. *ENV* can also be a stack pointer itself, in which case the value of **FUNCTION** is **(FUNARG** *FN ENV***)**. Finally, *ENV* can be an atom, in which case it is evaluated, and the value interpreted as described above.

As explained above, one of the possible values that **FUNCTION** can return is the form **(FUNARG** *FN POS***)**, where *FN* is a function and *POS* is a stack pointer. **FUNARG** is not a function itself. Like **LAMBDA** and **NLAMBDA**, it has meaning and is specially recognized by Interlisp only in the context of applying a function to arguments. In other words, the expression **(FUNARG** *FN POS***)** is used exactly like a function. When a **FUNARG** expression is applied or is **CAR** of a form being **EVAL**'ed, the **APPLY** or **EVAL** takes place in the access environment specified by *ENV* (see page 11.1). Consider the following example:

```
←(DEFINEQ (DO.TWICE (FN VAL)
        (APPLY* FN (APPLY* FN VAL)))  )
(DO.TWICE)
←(DO.TWICE [FUNCTION (LAMBDA (X) (IPLUS X X))]
      5)
20
←(SETQ VAL 1)
1
←(DO.TWICE [FUNCTION (LAMBDA (X) (IPLUS X VAL))]
      5)
15
←(DO.TWICE [FUNCTION (LAMBDA (X) (IPLUS X VAL)) (VAL)]
      5)
7
```

DO.TWICE is defined to apply a function **FN** to a value **VAL**, and apply **FN** again to the value returned; in other words it calculates **(FN (FN VAL))**. Given the expr definition **(LAMBDA (X) (IPLUS X X))**, which doubles a given value, it correctly calculates **(FN (FN 5))** = **(FN 10)** = 20. However, when given **(LAMBDA (X) (IPLUS X VAL))**, which should add the value of the global variable **VAL** to the argument **X**, it does something unexpected, returning 15, rather than 5 + 1 + 1 = 7. The problem is that when the expr

definition is evaluated, it is evaluated in the context of **DO.TWICE**, where **VAL** is bound to the second argument of **DO.TWICE**, namely 5. In this case, one solution is to use the *ENV* argument to **FUNCTION** to construct a **FUNARG** expression which contains the value of **VAL** at the time that the **FUNCTION** is executed. Now, when **(LAMBDA (X) (IPLUS X VAL))** is evaluated, it is evaluated in an environment where the global value of **VAL** is accessable. Admittedly, this is a somewhat contrived example (it would be easy enough to change the argument names to **DO.TWICE** so there would be no conflict), but this situation arises occasionally with large systems of programs that construct functions, and pass them around.

Note: System functions with functional arguments (**APPLY**, **MAPCAR**, etc.) are compiled so that their arguments are local, and not accessable (see page 18.5). This reduces problems with conflicts with free variables used in functional arguments.

FUNARG expressions can be used for more than just circumventing the clashing of variables. For example, a **FUNARG** expression can be returned as the value of a computation, and then used "higher up". Furthermore, if the function in a **FUNARG** expression *sets* any of the variables contained in the frame, only the frame would be changed. For example, consider the following function:

```
←(DEFINEQ (MAKECOUNTER (CNT)
  (FUNCTION [LAMBDA NIL
        (PROG1 CNT (SETQ CNT (ADD1 CNT]
      (CNT)))]
```

The function **MAKECOUNTER** returns a **FUNARG** that increments and returns the previous value of the counter **CNT**. However, this is done within the environment of the call to **MAKECOUNTER** where **FUNCTION** was executed, which the **FUNARG** expression "carries around" with it, even after **MAKECOUNTER** has finished executing. Note that each call to **MAKECOUNTER** creates a **FUNARG** expression with a new, independent environment, so that multiple counters can be generated and used:

```
←(SETQ C1 (MAKECOUNTER 1))
(FUNARG (LAMBDA NIL (PROG1 CNT (SETQ CNT (ADD1 CNT))))
#1,13724/*FUNARG)
←(APPLY C1)
1
←(APPLY C1)
2
←(SETQ C2 (MAKECOUNTER 17))
(FUNARG (LAMBDA NIL (PROG1 CNT (SETQ CNT (ADD1 CNT))))
#1,13736/*FUNARG)
←(APPLY C2)
17
```

```
← (APPLY C2)
18
← (APPLY C1)
3
← (APPLY C2)
19
```

By creating a **FUNARG** expression with **FUNCTION**, a program can create a function object which has updateable binding(s) associated with the object which last *between* calls to it, but are only accessible through that instance of the function. For example, using the **FUNARG** device, a program could maintain two different instances of the same random number generator in different states, and run them independently.

10.6 Macros

Macros provide an alternative way of specifying the action of a function. Whereas function definitions are evaluated with a "function call", which involves binding variables and other housekeeping tasks, macros are evaluated by *translating* one Interlisp form into another, which is then evaluated.

A litatom may have both a function definition and a macro definition. When a form is evaluated by the interpreter, if the **CAR** has a function definition, it is used (with a function call), otherwise if it has a macro definition, then that is used. However, when a form is compiled, the **CAR** is checked for a macro definition first, and only if there isn't one is the function definition compiled. This allows functions that behave differently when compiled and interpreted. For example, it is possible to define a function that, when interpreted, has a function definition that is slow and has a lot of error checks, for use when debugging a system. This function could also have a macro definition that defines a fast version of the function, which is used when the debugged system is compiled.

Macro definitions are represented by lists that are stored on the property list of a litatom. Macros are often used for functions that should be compiled differently in different Interlisp implementations, and the exact property name a macro definition is stored under determines whether it should be used in a particular implementation. The global variable **MACROPROPS** contains a list of all possible macro property names which should be saved by the **MACROS** file package command. Typical macro property names are **DMACRO** for Interlisp-D, **10MACRO** for Interlisp-10, **VAXMACRO** for Interlisp-VAX, **JMACRO** for Interlisp-Jerico, and **MACRO** for

"implementation independent" macros. The global variable **COMPILERMACROPROPS** is a list of macro property names. Interlisp determines whether a litatom has a macro definition by checking these property names, in order, and using the first non-**NIL** property value as the macro definition. In Interlisp-D this list contains **DMACRO** and **MACRO** in that order so that **DMACRO**s will override the implementation-independent **MACRO** properties. In general, use a **DMACRO** property for macros that are to be used only in Interlisp-D, use **10MACRO** for macros that are to be used only in Interlisp-10, and use **MACRO** for macros that are to affect both systems.

Macro definitions can take the following forms:

(LAMBDA ...)
(NLAMBDA ...)

A function can be made to compile open by giving it a macro definition of the form **(LAMBDA ...)** or **(NLAMBDA ...)**, e.g., **(LAMBDA (X) (COND ((GREATERP X 0) X) (T (MINUS X))))** for **ABS**. The effect is as if the macro definition were written in place of the function wherever it appears in a function being compiled, i.e., it compiles as a lambda or nlambda expression. This saves the time necessary to call the function at the price of more compiled code generated in-line.

(NIL *EXPRESSION***)**
(*LIST EXPRESSION***)**

"Substitution" macro. Each argument in the form being evaluated or compiled is substituted for the corresponding atom in *LIST*, and the result of the substitution is used instead of the form. For example, if the macro definition of **ADD1** is **((X) (IPLUS X 1))**, then, **(ADD1 (CAR Y))** is compiled as **(IPLUS (CAR Y) 1)**.

Note that **ABS** could be defined by the substitution macro **((X) (COND ((GREATERP X 0) X) (T (MINUS X))))**. In this case, however, **(ABS (FOO X))** would compile as

```
(COND ((GREATERP (FOO X) 0)
       (FOO X))
      (T (MINUS (FOO X))))
```

and **(FOO X)** would be evaluated two times. (Code to evaluate **(FOO X)** would be generated three times.)

(OPENLAMBDA *ARGS BODY***)**

This is a cross between substitution and **LAMBDA** macros. When the compiler processes an **OPENLAMBDA**, it attempts to substitute the actual arguments for the formals wherever this preserves the frequency and order of evaluation that would have resulted from a **LAMBDA** expression, and produces a **LAMBDA** binding only for those that require it.

Note: **OPENLAMBDA** assumes that it can substitute literally the actual arguments for the formal arguments in the body of the macro if the actual is side-effect free or a constant. Thus, you should be careful to use names in *ARGS* which don't occur in

BODY (except as variable references). For example, if **FOO** has a macro definition of

(OPENLAMBDA (ENV) (FETCH (MY-RECORD-TYPE ENV) OF BAR))

then **(FOO NIL)** will expand to

(FETCH (MY-RECORD-TYPE NIL) OF BAR)

T When a macro definition is the atom **T**, it means that the compiler should ignore the macro, and compile the function definition; this is a simple way of turning off other macros. For example, the user may have a function that runs in both Interlisp-D and Interlisp-10, but has a macro definition that should only be used when compiling in Interlisp-10. If the **MACRO** property has the macro specification, a **DMACRO** of **T** will cause it to be ignored by the Interlisp-D compiler. Note that this **DMACRO** would not be necessary if the macro were specified by a **10MACRO** instead of a **MACRO**.

(= . *OTHER-FUNCTION***)** A simple way to tell the compiler to compile one function exactly as it would compile another. For example, when compiling in Interlisp-D, **FRPLACA**s are treated as **RPLACA**s. This is achieved by having **FRPLACA** have a **DMACRO** of **(= . RPLACA)**.

(*LITATOM EXPRESSION***)** If a macro definition begins with a litatom other than those given above, this allows *computation* of the Interlisp expression to be evaluated or compiled in place of the form. *LITATOM* is bound to the **CDR** of the calling form, *EXPRESSION* is evaluated, and the result of this evaluation is evaluated or compiled in place of the form. For example, **LIST** could be compiled using the computed macro:

```
[X (LIST 'CONS
    (CAR X)
    (AND (CDR X)
      (CONS 'LIST
        (CDR X]
```

This would cause **(LIST X Y Z)** to compile as **(CONS X (CONS Y (CONS Z NIL)))**. Note the recursion in the macro expansion.

If the result of the evaluation is the litatom **IGNOREMACRO**, the macro is ignored and the compilation of the expression proceeds as if there were no macro definition. If the litatom in question is normally treated specially by the compiler (**CAR**, **CDR**, **COND**, **AND**, etc.), and also has a macro, if the macro expansion returns **IGNOREMACRO**, the litatom will still be treated specially.

In Interlisp-10, if the result of the evaluation is the atom **INSTRUCTIONS**, no code will be generated by the compiler. It is then assumed the evaluation was done for effect and the necessary code, if any, has been added. This is a way of giving direct instructions to the compiler if you understand it.

Note: It is often useful, when constructing complex macro expressions, to use the **BQUOTE** facility (see page 25.42).

The following function is quite useful for debugging macro definitions:

(EXPANDMACRO *EXP QUIETFLG* — —**)** [Function]

Takes a form whose **CAR** has a macro definition and expands the form as it would be compiled. The result is prettyprinted, unless *QUIETFLG* = **T**, in which case the result is simply returned.

10.6.1 DEFMACRO

Macros defined with the function **DEFMACRO** are much like "computed" macros (page 10.23), in that they are defined with a form that is evaluated, and the result of the evaluation is used (evaluated or compiled) in place of the macro call. However, **DEFMACRO** macros support complex argument lists with optional arguments, default values, and keyword arguments. In addition, argument list destructuring is supported.

(DEFMACRO *NAME ARGS FORM***)** [NLambda NoSpread Function]

Defines *NAME* as a macro with the arguments *ARGS* and the definition form *FORM* (*NAME*, *ARGS*, and *FORM* are unevaluated). If an expression starting with *NAME* is evaluated or compiled, arguments are bound according to *ARGS*, *FORM* is evaluated, and the value of *FORM* is evaluated or compiled instead. The interpretation of *ARGS* is described below.

Note: Unlike the function **DEFMACRO** in Common Lisp, this function currently does not remove any function definition for *NAME*.

ARGS is a list that defines how the argument list passed to the macro *NAME* is interpreted. Specifically, *ARGS* defines a set of variables that are set to various arguments in the macro call (unevaluated), that *FORM* can reference to construct the macro form.

In the simplest case, *ARGS* is a simple list of variable names that are set to the corresponding elements of the macro call (unevaluated). For example, given:

(DEFMACRO FOO (A B) (LIST 'PLUS A B B))

The macro call **(FOO X (BAR Y Z))** will expand to **(PLUS X (BAR Y Z) (BAR Y Z))**.

The list *ARGS* can include any of a number of special "&-keywords" (beginning with the character "**&**") that are used

to set variables to particular items from the macro call form, as follows:

&OPTIONAL Used to define optional arguments, possibly with default values. Each element on *ARGS* after **&OPTIONAL** until the next &-keyword or the end of the list defines an optional argument, which can either be a litatom or a list, interpreted as follows:

VAR

If an optional argument is specified as a litatom, that variable is set to the corresponding element of the macro call (unevaluated).

(*VAR DEFAULT*)

If an optional argument is specified as a two element list, *VAR* is the variable to be set, and *DEFAULT* is a form that is evaluated and used as the default if there is no corresponding element in the macro call.

(*VAR DEFAULT VARSETP*)

If an optional argument is specified as a three element list, *VAR* and *DEFAULT* are the variable to be set and the default form, and *VARSETP* is a variable that is set to **T** if the optional argument is given in the macro call, **NIL** otherwise. This can be used to determine whether the argument was not given, or whether it was specified with the default value.

For example, after

(DEFMACRO FOO (&OPTIONAL A (B 5) (C 6 CSET)) *FORM*)

expanding the macro call **(FOO)** would cause *FORM* to be evaluated with **A** set to **NIL**, **B** set to **5**, **C** set to **6**, and **CSET** set to **NIL**. **(FOO 4 5 6)** would be the same, except that **A** would be set to **4** and **CSET** would be set to **T**.

&REST
&BODY Used to get a list of all additional arguments from the macro call. Either **&REST** or **&BODY** should be followed by a single litatom, which is set to a list of all arguments to the macro after the position of the &-keyword. For example, given

(DEFMACRO FOO (A B &REST C) *FORM*)

expanding the macro call **(FOO 1 2 3 4 5)** would cause *FORM* to be evaluated with **A** set to **1**, **B** set to **2**, and **C** set to **(3 4 5)**.

Note: If the macro calling form contains keyword arguments (see **&KEY** below) these are included in the **&REST** list.

&KEY Used to define keyword arguments, that are specified in the macro call by including a "keyword" (a litatom starting with the character ":") followed by a value.

Each element on *ARGS* after **&KEY** until the next &-keyword or the end of the list defines a keyword argument, which can either be a litatom or a list, interpreted as follows:

VAR

(*VAR*)

((*KEYWORD VAR*))

If a keyword argument is specified by a single litatom *VAR*, or a one-element list containing *VAR*, it is set to the value of a keyword argument, where the keyword used is created by adding the character ":" to the front of *VAR*. If a keyword argument is specified by a single-element list containing a two-element list, *KEYWORD* is interpreted as the keyword (which should start with the letter ":"), and *VAR* is the variable to set.

(*VAR DEFAULT*)

((*KEYWORD VAR*) *DEFAULT*)

(*VAR DEFAULT VARSETP*)

((*KEYWORD VAR*) *DEFAULT VARSETP*)

If a keyword argument is specified by a two or three-element list, the first element of the list specifies the keyword and variable to set as above. Similar to **&OPTIONAL** (above), the second element *DEFAULT* is a form that is evaluated and used as the default if there is no corresponding element in the macro call, and the third element *VARSETP* is a variable that is set to **T** if the optional argument is given in the macro call, **NIL** otherwise.

For example, the form

(DEFMACRO FOO (&KEY A (B 5 BSET) ((:BAR C) 6 CSET)) *FORM*)

Defines a macro with keys **:A**, **:B** (defaulting to 5), and **:BAR**. Expanding the macro call **(FOO :BAR 2 :A 1)** would cause *FORM* to be evaluated with **A** set to **1**, **B** set to **5**, **BSET** set to **NIL**, **C** set to 2, and **CSET** set to **T**.

&ALLOW-OTHER-KEYS It is an error for any keywords to be suplied in a macro call that are not defined as keywords in the macro argument list, unless either the &-keyword **&ALLOW-OTHER-KEYS** appears in *ARGS*, or the keyword **:ALLOW-OTHER-KEYS** (with a non-**NIL** value) appears in the macro call.

&AUX Used to bind and initialize auxiliary varables, using a syntax similar to **PROG** (page 9.8). Any elements after **&AUX** should be either litatoms or lists, interpreted as follows:

VAR

Single litatoms are interpreted as auxiliary variables that are initially bound to **NIL**.

(*VAR EXP*)

If an auxiliary variable is specified as a two element list, *VAR* is a variable initially bound to the result of evaluating the form *EXP*.

For example, given

(DEFMACRO FOO (A B &AUX C (D 5)) *FORM*)

C will be bound to **NIL** and **D** to **5** when *FORM* is evaluated.

&WHOLE Used to get the whole macro calling form. Should be the first element of *ARGS*, and should be followed by a single litatom, which is set to the entire macro calling form. Other &-keywords or arguments can follow. For example, given

(DEFMACRO FOO (&WHOLE X A B) *FORM*)

Expanding the macro call **(FOO 1 2)** would cause *FORM* to be evaluated with **X** set to **(FOO 1 2)**, **A** set to **1**, and **B** set to **2**.

DEFMACRO macros also support argument list "destructuring," a facility for accessing the structure of individual arguments to a macro. Any place in an argument list where a litatom is expected, an argument list (in the form described above) can appear instead. Such an embedded argument list is used to match the corresponding parts of that particular argument, which should be a list structure in the same form. In the simplest case, where the embedded argument list does not include &-keywords, this provides a simple way of picking apart list structures passed as arguments to a macro. For example, given

(DEFMACRO FOO (A (B (C . D)) E) *FORM*)

Expanding the macro call **(FOO 1 (2 (3 4 5)) 6)** would cause *FORM* to be evaluated with with **A** set to **1**, **B** set to **2**, **C** set to **3**, **D** set to **(4 5)**, and **E** set to **6**. Note that the embedded argument list **(B (C . D))** has an embedded argument list **(C . D)**. Also notice that if an argument list ends in a dotted pair, that the final litatom matches the rest of the arguments in the macro call.

An embedded argument list can also include &-keywords, for interpreting parts of embedded list structures as if they appeared in a top-level macro call. For example, given

(DEFMACRO FOO (A (B &OPTIONAL (C 6)) D) *FORM*)

Expanding the macro call **(FOO 1 (2) 3)** would cause *FORM* to be evaluated with with **A** set to **1**, **B** set to **2**, **C** set to **6** (because of the default value), and **D** set to **3**.

Warning: Embedded argument lists can only appear in positions in an argument list where a list is otherwise not accepted. In the above example, it would not be possible to specify an embedded argument list after the **&OPTIONAL** keyword, because it would be interpreted as an optional argument specification (with variable name, default value, set variable). However, it would be possible to specify an embedded argument list as the first element of an optional argument specification list, as so:

(DEFMACRO FOO (A (B &OPTIONAL ((X (Y) Z) '(1 (2) 3))) D) *FORM*)

In this case, **X**, **Y**, and **Z** default to **1**, **2**, and **3**, respectively. Note that the "default" value has to be an appropriate list structure. Also, in this case either the whole structure **(X (Y) Z)** can be

supplied, or it can be defaulted (i.e. is not possible to specify X while letting Y default).

10.6.2 Interpreting Macros

When the interpreter encounters a form **CAR** of which is an undefined function, it tries interpreting it as a macro. If **CAR** of the form has a macro definition, the macro is expanded, and the result of this expansion is evaluated in place of the original form. **CLISPTRAN** (page 21.25) is used to save the result of this expansion so that the expansion only has to be done once. On subsequent occasions, the translation (expansion) is retrieved from **CLISPARRAY** the same as for other CLISP constructs.

Note: Because of the way that the evaluator processes macros, if you have a macro on **FOO**, then typing **(FOO 'A 'B)** will work, but **FOO(A B)** will not work.

Sometimes, macros contain calls to functions that assume that the macro is being compiled. The variable **SHOULDCOMPILEMACROATOMS** is a list of functions that should be compiled to work correctly (initially **(OPCODES)** in Interlisp-D, **(ASSEMBLE LOC)** in Interlisp-10). **UNSAFEMACROATOMS** is a list of functions which effect the operation of the compiler, so such macro forms shouldn't even be expanded except by the compiler (initially **NIL** in Interlisp-D, **(C2EXP STORIN CEXP COMP)** in Interlisp-10). If the interpreter encounters a macro containing calls to functions on these two lists, instead of the macro being expanded, a dummy function is created with the form as its definition, and the dummy function is then compiled. A form consisting of a call to this dummy function with no arguments is then evaluated in place of the original form, and **CLISPTRAN** is used to save the translation as described above. There are some situations for which this procedure is not amenable, e.g. a **GO** inside the form which is being compiled will cause the compiler to give an **UNDEFINED TAG** error message because it is not compiling the entire function, just a part of it.