CHAPTER 5

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.

The user defines a function with a list expressions known as an `EXPR`. An `EXPR` 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` would have two evaluated arguments, `X` and `Y`, and it would execute `(PRINT X)` and `(PRINT Y)` when evaluated. Other types of `EXPR`s are described below.

A function is defined by putting an `EXPR` 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 5.9). 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`s. The compiler (page 12.1) translates `EXPR` definitions into compiled code objects, which execute much faster. In Interlisp-10, many primitive system functions are defined with machine code objects known as `SUBR`s. Interlisp provides a number of ‘‘function type functions’’ which determine how a given function is defined (`EXPR`/compiled code/`SUBR`), the number and names of function arguments, etc. See page 5.6.

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`) 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 5.15).

The macro facility provides another way of specifying the behavior of a function (see page 5.17). 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.

5.1 FUNCTION TYPES

Interlisp functions are defined using list expressions called `EXPR`s. An `EXPR` 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` 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: The Lambdatran lispusers package provides facilities for creating new function types which evaluate/spread their arguments in different ways than those provided by Interlisp. See page 23.16.

5.1.1 Lambda-Spread Functions

Lambda-spread functions take a fixed number of evaluated arguments. This is the most common function type. A lambda-spread `EXPR` 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) (PRINT X) (PRINT Y))) )
(FOO)
_ (FOO 99 (PLUS 3 4))
99
7
7
_
```

In the above example, the function `FOO` defined by `(LAMBDA (X Y) (PRINT X) (PRINT 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, `(PRINT X)` is evaluated, printing 99, `(PRINT Y)` is evaluated, printing 7, and 7 (the *value* of `(PRINT Y)`) 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 5.11).

5.1.2 Nlambda-Spread Functions

Nlambda-spread functions take a fixed number of unevaluated arguments. An nlambda-spread `EXPR` 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) (PRINT X) (PRINT Y))) )
(FOO)
_ (FOO 99 (PLUS 3 4))
99
(PLUS 3 4)
(PLUS 3 4)
_
```

In the above example, the function `FOO` defined by `(NLAMBDA (X Y) (PRINT X) (PRINT Y))` is applied to the arguments `99` and `(PLUS 3 4)`, these arguments are bound unevaluated to `X` and `Y`, `(PRINT X)` is evaluated, printing `99`, `(PRINT Y)` is evaluated, printing `(PLUS 3 4)`, and the list `(PLUS 3 4)` 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 5.11). 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.

5.1.3 Lambda-Nospread Functions

Lambda-nospread functions take a variable number of evaluated arguments. A lambda-nospread EXPR 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 pushdown list. 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 Mth 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 Mth 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 print all of the evaluated arguments it is given, and return NIL (the value of the for statement).

```
_ (DEFINEQ (FOO
             (LAMBDA X
                (for ARGNUM from 1 to X do (PRINT (ARG X ARGNUM))))) )
(FOO)
_ (FOO 99 (PLUS 3 4))
99
7
NIL
_ (FOO 99 (PLUS 3 4) (TIMES 3 4))
99
7
12
NIL
```

_

5.1.4 Nlambda-Nospread Functions

Nlambda- nospread functions take a variable number of unevaluated arguments. An nlambda- nospread EXPR 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 EXPRs, an nlambda- nospread is evaluated quite di erently. 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 di erent arguments.

In the example below, FOO is de ned to print (and then return) the reverse of list of arguments it is given (unevaluated):

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

5.1.5 Compiled Functions

Functions de ned by EXPRs can be compiled by the Interlisp compiler (page 12.1), which produces compiled code objects, which execute more quickly than the corresponding EXPR code. Functions de ned by compiled code objects may have the same four types as EXPRs (lambda/nolambda, spread/nospread). Functions created by the compiler are referred to as compiled functions.

5.1.6 SUBRs

In Interlisp- 10, basic built- in functions such as CONS, CAR, and COND are handcoded in machine language. These functions are known as ‘‘SUBRs.’’ Functions de ned as SUBRs can be lambda/nolambda or spread/nospread, the same four function types as EXPR functions.

SUBRs are called in a special way, so their de nitions are stored di erently than those of compiled or interpreted functions. GETD of a SUBR returns a dotted pair, CAR of which is an encoding of the ARGTYPE and number of arguments of the SUBR, and CDR of which is the address of the rst instruction. Note that each GETD of a subr performs a CONS. Similarly, PUTD of a de nition of the form (NUMBER . ADDRESS), where NUMBER and ADDRESS are in the appropriate ranges, stores the de nition as a SUBR.

5.1.7 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 de nition from the litatom's de nition cell, or a function de nition itself.

(FNTYP FN) [Function]

Returns NIL if FN is not a function de nition or the name of a de ned function. Otherwise FNTYP returns one of the following twelve litatoms:

	Expressions	Compiled	Built-In
Lambda-Spread	EXPR	CEXPR	SUBR
Nlambda-Spread	FEXPR	CFEXPR	FSUBR
Lambda-Nospread	EXPR*	CEXPR*	SUBR*
Nlambda-Nospread	FEXPR*	CFEXPR*	FSUBR*

The types in the rst column are all de ned by EXPRs. The types in the second column are compiled versions of the types in the rst column, as indicated by the pre x C. In the third column are the parallel types for built-in subroutines (only in Interlisp-10). Functions of types in the rst two rows have a xed number of arguments, i.e., are spread functions. Functions in the third and fourth rows have an inde nite number of arguments, as indicated by the su x *. The pre x F indicates unevaluated arguments. Thus, for example, a CFEXPR* is a compiled nospread-nlambda function.

FNTYP returns the litatom FUNARG if FN is a FUNARG expression. See page 5.15.

(EXPRP FN) [Function]

Returns T if (FNTYP FN) is either EXPR, FEXPR, EXPR*, or FEXPR*, i.e., rst column of FNTYPs; NIL otherwise. However, (EXPRP FN) is also true if FN is (has) a list de nition that is not a SUBR, 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*, i.e., second column of FNTYPs; NIL otherwise.

(SUBRP FN) [Function]

Returns T if (FNTYP FN) is either SUBR, FSUBR, SUBR*, or FSUBR*, i.e., the third column of FNTYPs; NIL otherwise.

(ARGTYPE FN) [Function]

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

0 lambda-spread functions (EXPR, CEXPR, SUBR)

1 nlambda- spread functions (FEXPR, CFEXPR, FSUBR)

2 lambda- nospread functions (EXPR*, CEXPR*, SUBR*)

3 nlambda- nospread functions (FEXPR*, CFEXPR*, FSUBR*)

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 EXPRs, whose de nitions are lists beginning with LAMBDA or NLAMBDA, the argument list is simply CADR of GETD. If FN has a list de nition, and CAR of the de nition is not LAMBDA or NLAMBDA, ARGLIST will check to see if CAR of the de nition is a member of LAMBDASPLST (page 15.12). If it is, ARGLIST presumes this is a function object the user is de ning via DWIMUSERFORMS (page 15.10), and simply returns CADR of the de nition as its argument list. Otherwise ARGLIST generates an error as described above.

(Interlisp- 10) If FN is a spread SUBR, the ARGLIST returns (U), (U V), (U V W), etc. depending on the number of arguments; if a nospread SUBR, it returns U. This is merely a ''feature'' of ARGLIST; SUBRs do not actually store the names of their arguments(s) on the stack.

(SMARTARGLIST FN EXPLAINFL G TAIL) [Function]

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

If FN is not de ned as a function, SMARTARGLIST attempts spelling correction on FN by calling FNCHECK (page 15.19), passing TAIL to be used for the call to FIXSPELL. If unsuccessful, an error will be generated, FN NOT A FUNCTION.

If FN is known to the le package (page 11.1) but not loaded in, SMARTARGLIST will obtain the arglist information from the le.

In Interlisp- 10, if the HELPSYS help system is installed, SMARTARGLIST may use it to look up the arguments to FN in the Interlisp manual les. Speci cally, HELPSYS will be used if EXPLAINFL G = T and FN is a nospread function, or if FN is a spread SUBR, regardless of the value of EXPLAINFL G. For all other cases, and when HELPSYS is unde ned or unsuccessful in nding the arguments, SMARTARGLIST simply returns (ARGLIST FN).

In order to avoid repeated calls to HELPSYS, and also to provide the user with an override, SMARTARGLIST stores the arguments returned from HELPSYS on the property list of FN under the property ARGNAMES and checks for this property before calling HELPSYS. For spread functions, the argument list itself is stored.

For nospread, the form is `(NIL ARGLIST1 . ARGLIST2)` where ARGLIST1 is the value of `SMARTARGLIST` when EXPLAINFLG = `T`, and ARGLIST2 the value when EXPLAINFLG = `NIL`. For example, `(GETPROP 'DEFINEQ 'ARGNAMES) = (NIL (X1 XI ... XN) . X)`.

`SMARTARGLIST` is used by `BREAK` (page 10.4) and `ADVISE` (page 10.9) with EXPLAINFLG = `NIL` for constructing equivalent `EXPR` de nitions, and by the programmer's assistant command `?=` (page 9.5), with EXPLAINFLG = `T`.

5.2 FUNCTION DEFINITION

Function de nitions are stored in a ''function de nition cell'' associated with each litatom. This cell is directly accessible via the two functions `PUTD` and `GETD`, but it is usually easier to de ne functions with `DEFINEQ` (page 5.9).

`(GETD FN)` [Function]

Returns the function de nition of FN. Returns `NIL` if FN is not a litatom, or has no de nition.

`GETD` of a compiled function constructs a pointer to the de nition, with the result that two successive calls do not produce `EQ` results. `EQP` or `EQUAL` must be used to compare compiled de nitions.

(Interlisp- 10) `GETD` of a `SUBR` performs a `CONS`.

`(FGETD FN)` [Function]

Faster version of `GETD`. Interpreted, generates an error, `BAD ARGUMENT - FGETD`, if FN is not a litatom.

`FGETD` is intended primarily to check whether a function *has* a de nition, rather than to obtain the de nition. Therefore, in Interlisp- 10, `FGETD` of a `SUBR` returns just the address of the function de nition, not the dotted pair returned by `GETD`, thereby saving the `CONS`.

`(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`.

`(PUTDQ FN DEF)` [NLambda Function]

Nlambda version of `PUTD`; both arguments are unevaluated. Returns FN.

`(PUTDQ? FN DEF)` [NLambda Function]

If FN is not de ned, same as `PUTDQ`. Otherwise, does nothing and returns `NIL`.

`(MOVD FROM TO COPYFLG)` [Function]

Moves the de nition of FROM to TO, i.e., rede nes TO. If COPYFLG = `T`, a `COPY` of the de nition of FROM is used. COPYFLG = `T` is only meaningful for `EXPR`s, although `MOVD` works for compiled functions and `SUBR`s as well. `MOVD` returns

TO.

(MOVD? FROM TO COPYFLG) [Function]

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

(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 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 17.60), both DEFINE and DEFINEQ will stamp the definition with the user's initials and date.

DFNFLG [Variable]

DFNFLG is a global variable that effects the operation of DEFINE (and DEFINEQ, which calls 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 before redefining it, except if the old and new definitions are the same (i.e. EQUAL), 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 affects the operation of RPAQQ and RPAQ (page 11.37). DFNFLG is initially NIL.

DFNFLG is reset by LOAD (page 11.4) 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.

(SAVEDEF FN) [Function]

Saves the definition of FN on its property list under the property EXPR, CODE, or `SUBR` depending on its `FNTYP`. Returns the property name used. If `(GETD FN)` is non-`NIL`, but `(FNTYP FN)`= `NIL`, `SAVEDEF` saves the de nition on the property name `LIST`. This situation can arise when a function is rede ned which was originally de ned with `LAMBDA` misspelled or omitted.

If `FN` is a list, `SAVEDEF` operates on each function in the list, and returns a list of the individual values.

`(UNSAVEDEF FN PROP)` [Function]

Restores the de nition of `FN` from its property list under property `PROP` (see `SAVEDEF` above). Returns `PROP`. If nothing is saved under `PROP`, and `FN` is de ned, returns `(PROP NOT FOUND)`, otherwise generates an error, `NOT A FUNCTION`.

If `PROP` is not given, i.e., `NIL`, `UNSAVEDEF` looks under the properties `EXPR`, `CODE`, and `SUBR`, in that order. The value of `UNSAVEDEF` is the property name, or if nothing is found and `FN` is a function, the value is `(NOTHING FOUND)`; otherwise generates an error, `NOT A FUNCTION`.

If `DFNFLG`= `NIL`, the current de nition of `FN`, if any, is saved using `SAVEDEF`. Thus one can use `UNSAVEDEF` to switch back and forth between two de nitions of the same function, keeping one on its property list and the other in the function de nition cell.

If `FN` is a list, `UNSAVEDEF` operates on each function of the list, and its value is a list of the individual values.

Both `SAVEDEF` and `UNSAVEDEF` are rede ned in more general terms (see page 11.18) to operate on typed de nitions of which a function de nition is but one example. Thus, their actual argument lists in Interlisp are di erent than given here. However, when their extra arguments are defaulted to `NIL`, they operate as described above.

5.3 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 de nitions, or `EXPR` forms such as `(LAMBDA (X) )`, or `FUNARG` expressions (see page 5.15).

The following functions are useful when one wants to supply a functional argument which will always return `NIL`, `T`, or 0.

`(NILL)` [NoSpread Function]

Returns `NIL`.

(TRUE) [NoSpread Function]

Returns T.

(ZERO) [NoSpread Function]

Returns 0.

Note: When using EXPR expressions as functional arguments, they should be enclosed within the function FUNCTION (page 5.15) 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.

(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 rst evaluated, e.g.,

```
_(SETQ FOO '(ADD1 3))
(ADD1 3)
_(EVAL FOO)
4
_(EVAL 'FOO)
(ADD1 3)
```

Interlisp functions can either evaluate or not evaluate these arguments. For those cases where it is desirable to specify arguments unevaluated, one may use the QUOTE function:

(QUOTE X) [NLambda NoSpread Function]

This is a function that prevents its arguments from being evaluated. Its value is X itself, e.g., (QUOTE FOO) is FOO.

Note: 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, if the value of X is A and the value of Y is B, then (KWOTE (CONS X Y)) = (QUOTE (A . B)).

(DEFEVAL TYPE FN) [Function]

Speci es how a datum of a particular type is to be evaluated.[1] Intended primarily for user de ned 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

[1]COMPILETYPELST (page 12.9) permits the user to specify how a datum of a particular type is to be compiled.

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).

(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. Note that FN may still explicitly evaluate one or more of its arguments itself, as SETQ does. Thus, (APPLY 'SETQ '(FOO (ADD1 3))) will set FOO to 4, whereas (APPLY 'SET '(FOO (ADD1 3))) will set FOO to the expression (ADD1 3).

APPLY can be used for manipulating EXPRs, 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, equivalent to (APPLY FN (LIST ARG_1 ARG_2 ARG_N)).

(EVALA X A) [Function]

Simulates a-list evaluation as in LISP 1.5. X is a form, A is a list of the form:

(($NAME_1$. VAL_1) ($NAME_2$. VAL_2) ($NAME_N$. 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.

The functions below are used to evaluate a form or apply a function repeatedly. RPT, RPTQ, and FRPTQ evaluate a given form a speci ed number of times. MAP, MAPCAR, MAPLIST, etc. apply a given function repeatedly to di erent elements of a list, possibly constructing another list. These functions allow e cient iterative computations, but they are di cult to use. For programming iterative computations, it is usually better to use the CLISP Iterative Statement facility (page 4.5), 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 e ciency loss.

(RPT N FORM) [Function]

Evaluates the expression FORM, N times. Returns the value of the last evaluation. If N 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 FNTYPs for each element on X.

(MAPCON MAPX MAPFN1 MAPFN2) [Function]

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

(MAPCONC MAPX MAPFN1 MAPFN2) [Function]

Computes the same values as MAPC and MAPCAR, but NCONCs 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 NILs 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. It cycles through LST applying PFN (or PRIN1 if PFN not given) to each element of LST. 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 8.20) is used instead of PRIN1.

5.4 FUNCTIONAL ARGUMENTS

When using functional arguments, the following function is very useful:

(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
            (FIE LST (FUNCTION (LAMBDA (Z) (ITIMES Z Z))))
) )
```

FOO calls the function FIE with the value of LST and the EXPR expression (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 as an auxiliary function (See page 12.8). The compiled EXPR 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 7.1). 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)
20
_ (DO.TWICE [FUNCTION (LAMBDA (X) (IPLUS X VAL)) (VAL)]
            5)
7
```

DO.TWICE is de ned 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 expression (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 20 again, rather than 5+1+1 = 7. The problem is that when the EXPR 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 con ict), 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 12.4). This reduces problems with con icts 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:

```
(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 nished 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.

Note: In Interlisp-10, environment switching is expensive because it is a shallow-binding system (see page 7.1), so this may restrict the applications of `FUNARG` expressions.

5.5 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 `10MACRO` for Interlisp-10, `DMACRO` for Interlisp-D,[2] 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

[2] also `VAXMACRO` for Interlisp-VAX, and `JMACRO` for Interlisp-Jerico.

using the rst non-NIL property value as the macro de nition. In Interlisp-D this list contains DMACRO and MACRO in that order so that DMACROs 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 a ect both systems.

Macro de nitions can take the following forms:

(LAMBDA) or (NLAMBDA)

A function can be made to compile open by giving it a macro de nition of the form (LAMBDA) or (NLAMBDA), e.g., (LAMBDA (X) (COND ((GREATERP X 0) X) (T (MINUS X)))) for ABS. The e ect is as if the macro de nition 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) or (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 de nition of ADD1 is ((X) (IPLUS X 1)), then, (ADD1 (CAR Y)) is compiled as (IPLUS (CAR Y) 1).

Note that ABS could be de ned 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.

T

When a macro de nition is the atom T, it means that the compiler should ignore the macro, and compile the function de nition; this is a simple way of turning o other macros. For example, the user may have a function that runs in both Interlisp-D and Interlisp-10, but has a macro de nition that should only be used when compiling in Interlisp-10. If the MACRO property has the macro speci cation, 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 speci ed 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, FRPLACAs are treated as RPLACAs. 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 6.39).

The following function is quite useful for debugging macro definitions:

(EXPANDMACRO FORM 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.

5.5.1 MACROTRAN

Interpreted macros are implemented by the function MACROTRAN. When the interpreter encounters a form CAR of which is an undefined function,[3] MACROTRAN is called. 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 16.19) 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; MACROTRAN never even has to be invoked.

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

[3]In other words, if you have a macro on FOO, then typing (FOO 'A 'B) will work, but FOO(A B) will not work.

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 `MACROTRAN` 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.

Note: `MACROTRAN` is an entry on `DWIMUSERFORMS` (page 15.10) and thus will not work if DWIM is not enabled.