9. CONDITIONALS AND ITERATIVE STATEMENTS

In order to do any but the simplest computations, it is necessary to test values and execute expressions conditionally, and to execute a series of expressions. Interlisp supplies a large number of predicates, conditional functions, and control functions. Also, there is a complex "iterative statement" facility which allows the user to easily create complex loops and iterative constructs (page 9.9).

9.1 Data Type Predicates

Interlisp provides separate functions for testing whether objects are of certain commonly-used types:

(LITATOM *X***)** [Function]

Returns **T** if *X* is a litatom (see page 2.1) **NIL** otherwise. Note that a number is not a litatom.

(SMALLP *X***)** [Function]

Returns *X* if *X* is a small integer; **NIL** otherwise. (Note that the range of small integers is implementation-dependent. See page 7.1.)

(FIXP *X***)** [Function]

Returns *X* if *X* is a small or large integer; **NIL** otherwise.

(FLOATP *X***)** [Function]

Returns *X* if *X* is a floating point number; **NIL** otherwise.

(NUMBERP *X***)** [Function]

Returns *X* if *X* is a number of any type (**FIXP** or **FLOATP**), **NIL** otherwise.

(ATOM *X***)** [Function]

Returns **T** if *X* is an atom (i.e. a litatom or a number); **NIL** otherwise.

Warning: **(ATOM** *X*) is **NIL** if *X* is an array, string, etc. In many dialects of Lisp, the function **ATOM** is defined equivalent to the Interlisp function **NLISTP**.

(LISTP *X*) [Function]

Returns *X* if *X* is a list cell, e.g., something created by **CONS**; **NIL** otherwise.

(NLISTP *X*) [Function]

(NOT (LISTP X)). Returns **T** if *X* is not a list cell, **NIL** otherwise.

(STRINGP *X*) [Function]

Returns *X* if *X* is a string, **NIL** otherwise.

(ARRAYP *X*) [Function]

Returns *X* if *X* is an array, **NIL** otherwise.

Note: In some implementations of Interlisp (but not Interlisp-D), **ARRAYP** may also return *X* if it is of type **CCODEP** or **HARRAYP**.

(HARRAYP *X*) [Function]

Returns *X* if it is a hash array object; otherwise **NIL**.

Note that **HARRAYP** returns **NIL** if *X* is a list whose **CAR** is an **HARRAYP**, even though this is accepted by the hash array functions.

Note: The empty list, **()** or **NIL**, is considered to be a litatom, rather than a list. Therefore, **(LITATOM NIL)** = **(ATOM NIL)** = **T** and **(LISTP NIL)** = **NIL**. Care should be taken when using these functions if the object may be the empty list **NIL**.

9.2 Equality Predicates

A common operation when dealing with data objects is to test whether two objects are equal. In some cases, such as when comparing two small integers, equality can be easily determined. However, sometimes there is more than one type of equality. For instance, given two lists, one can ask whether they are exactly the same object, or whether they are two distinct lists which contain the same elements. Confusion between these two types of equality is often the source of program errors. Interlisp supplies an extensive set of functions for testing equality:

(EQ *X Y***)** [Function]

Returns **T** if *X* and *Y* are identical pointers; **NIL** otherwise. **EQ** should not be used to compare two numbers, unless they are small integers; use **EQP** instead.

(NEQ *X Y***)** [Function]

(NOT (EQ *X Y***))**

(NULL *X***)** [Function]

(NOT *X***)** [Function]

(EQ *X* **NIL)**

(EQP *X Y***)** [Function]

Returns **T** if *X* and *Y* are **EQ**, or if *X* and *Y* are numbers and are equal in value; **NIL** otherwise. For more discussion of **EQP** and other number functions, see page 7.1.

Note: **EQP** also can be used to compare stack pointers (page 11.4) and compiled code (page 10.10).

(EQUAL *X Y***)** [Function]

EQUAL returns **T** if *X* and *Y* are (1) **EQ**; or (2) **EQP**, i.e., numbers with equal value; or (3) **STREQUAL**, i.e., strings containing the same sequence of characters; or (4) lists and **CAR** of *X* is **EQUAL** to **CAR** of *Y*, and **CDR** of *X* is **EQUAL** to **CDR** of *Y*. **EQUAL** returns **NIL** otherwise. Note that **EQUAL** can be significantly slower than **EQ**.

A loose description of **EQUAL** might be to say that *X* and *Y* are **EQUAL** if they print out the same way.

(EQUALALL *X Y***)** [Function]

Like **EQUAL**, except it descends into the contents of arrays, hash arrays, user data types, etc. Two non-**EQ** arrays may be **EQUALALL** if their respective componants are **EQUALALL**.

9.3 Logical Predicates

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

Takes an indefinite number of arguments (including zero), that are evaluated in order. If any argument evaluates to **NIL**, **AND** immediately returns **NIL** (without evaluating the remaining arguments). If all of the arguments evaluate to non-**NIL**, the value of the last argument is returned. **(AND) => T**.

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

Takes an indefinite number of arguments (including zero), that are evaluated in order. If any argument is non-**NIL**, the value of that argument is returned by **OR** (without evaluating the remaining arguments). If all of the arguments evaluate to **NIL**, **NIL** is returned. **(OR) => NIL**.

AND and **OR** can be used as simple logical connectives, but note that they may not evaluate all of their arguments. This makes a difference if the evaluation of some of the arguments causes side-effects. Another result of this implementation of **AND** and **OR** is that they can be used as simple conditional statements. For example: **(AND (LISTP** *X***) (CDR** *X***))** returns the value of **(CDR** *X***)** if *X* is a list cell, otherwise it returns **NIL** without evaluating **(CDR** *X***)**. In general, this use of **AND** and **OR** should be avoided in favor of more explicit conditional statements in order to make programs more readable.

9.4 The COND Conditional Function

(COND $CLAUSE_1$ $CLAUSE_2$ ... $CLAUSE_K$**)** [NLambda NoSpread Function]

The conditional function of Interlisp, **COND**, takes an indefinite number of arguments, called clauses. Each $CLAUSE_i$ is a list of the form $(P_i\ C_{i1}\ ...\ C_{iN})$, where P_i is the predicate, and C_{i1} ... C_{iN} are the consequents. The operation of **COND** can be paraphrased as:

IF P_1 THEN C_{11} ... C_{1N} ELSEIF P_2 THEN C_{21} ... C_{2N} ELSEIF P_3 ...

The clauses are considered in sequence as follows: the predicate P_1 of the clause $CLAUSE_i$ is evaluated. If the value of P_1 is "true" (non-**NIL**), the consequents C_{i1} ... C_{iN} are evaluated in order, and the value of the **COND** is the value of C_{iN}, the last expression in the clause. If P_1 is "false" (**EQ** to **NIL**), then the remainder of $CLAUSE_i$ is ignored, and the next clause, $CLAUSE_{i+1}$, is considered. If no P_i is true for *any* clause, the value of the **COND** is **NIL**.

Note: If a clause has no consequents, and has the form (P_i), then if P_i evaluates to non-**NIL**, it is returned as the value of the **COND**. It is only evaluated once.

Example:

```
←(DEFINEQ (DOUBLE (X)
        (COND ((NUMBERP X) (PLUS X X))
```

```
        ((STRINGP X) (CONCAT X X))
        ((ATOM X) (PACK* X X))
        (T (PRINT "unknown") X)
        ((HORRIBLE-ERROR))]
(DOUBLE)
←(DOUBLE 5)
10
←(DOUBLE "FOO")
"FOOFOO"
←(DOUBLE 'BAR)
BARBAR
←(DOUBLE '(A B C))
"unknown"
(A B C)
```

A few points about this example: Notice that **5** is both a number and an atom, but it is "caught" by the **NUMBERP** clause before the **ATOM** clause. Also notice the predicate **T**, which is always true. This is the normal way to indicate a **COND** clause which will always be executed (if none of the preceeding clauses are true). **(HORRIBLE-ERROR)** will never be executed.

9.5 The IF Statement

The **IF** statement provides a way of way of specifying conditional expressions that is much easier and readable than using the **COND** function directly (page 9.4). CLISP translates expressions employing **IF**, **THEN**, **ELSEIF**, or **ELSE** (or their lowercase versions) into equivalent **COND** expressions. In general, statements of the form:

(if *AAA* **then** *BBB* **elseif** *CCC* **then** *DDD* **else** *EEE***)**

are translated to:

(COND (*AAA BBB***)**
(*CCC DDD***)**
(T *EEE***))**

The segment between **IF** or **ELSEIF** and the next **THEN** corresponds to the predicate of a **COND** clause, and the segment between **THEN** and the next **ELSE** or **ELSEIF** as the consequent(s). **ELSE** is the same as **ELSEIF T THEN**. These words are spelling corrected using the spelling list **CLISPIFWORDSPLST**. Lower case versions (**if**, **then**, **elseif**, **else**) may also be used.

If there is nothing following a **THEN**, or **THEN** is omitted entirely, then the resulting **COND** clause has a predicate but no consequent. For example, **(if X then elseif ...)** and **(if X elseif ...)**

both translate to **(COND (X) ...)**, which means that if **X** is not **NIL**, it is returned as the value of the **COND**.

Note that only one expression is allowed as the predicate, but multiple expressions are allowed as the consequents after **THEN** or **ELSE**. Multiple consequent expressions are implicitely wrapped in a **PROGN**, and the value of the last one is returned as the value of the consequent. For example:

(if X then (PRINT "FOO") (PRINT "BAR") elseif Y then (PRINT "BAZ"))

CLISP considers **IF**, **THEN**, **ELSE**, and **ELSEIF** to have lower precedence than all infix and prefix operators, as well as Interlisp forms, so it is sometimes possible to omit parentheses around predicate or consequent forms. For example, **(if FOO X Y then ...)** is equivalent to **(if (FOO X Y) then ...)**, and **(if X then FOO X Y else ...)** as equivalent to **(if X then (FOO X Y) else ...)**. Essentially, CLISP determines whether the segment between **THEN** and the next **ELSE** or **ELSEIF** corresponds to one form or several and acts accordingly, occasionally interacting with the user to resolve ambiguous cases. Note that if **FOO** is bound as a variable, **(if FOO then ...)** is translated as **(COND (FOO ...))**, so if a call to the *function* **FOO** is desired, use **(if (FOO) then ...)**.

9.6 Selection Functions

(SELECTQ *X CLAUSE*$_1$ *CLAUSE*$_2$ ... *CLAUSE*$_K$ *DEFAULT***)** [NLambda NoSpread Function]

Selects a form or sequence of forms based on the value of *X*. Each clause *CLAUSE*$_i$ is a list of the form $(S_i\ C_{i1}\ \ldots\ C_{iN})$ where S_i is the selection key. The operation of **SELECTQ** can be paraphrased as:

IF $X = S_1$ THEN C_{11} ... C_{1N} ELSEIF $X = S_2$ THEN ... ELSE *DEFAULT*.

If S_i is an atom, the value of X is tested to see if it is **EQ** to S_i (which is not evaluated). If so, the expressions C_{i1} ... C_{iN} are evaluated in sequence, and the value of the **SELECTQ** is the value of the last expression evaluated, i.e., C_{iN}.

If S_i is a list, the value of X is compared with each element (not evaluated) of S_i, and if X is **EQ** to any one of them, then C_{i1} ... C_{iN} are evaluated as above.

If *CLAUSE*$_i$ is not selected in one of the two ways described, *CLAUSE*$_{i+1}$ is tested, etc., until all the clauses have been tested. If none is selected, *DEFAULT* is evaluated, and its value is returned as the value of the **SELECTQ**. *DEFAULT* must be present.

An example of the form of a **SELECTQ** is:

```
[SELECTQ MONTH
  (FEBRUARY (if (LEAPYEARP) then 29 else 28))
  ((APRIL JUNE SEPTEMBER NOVEMBER) 30)
  31]
```

If the value of **MONTH** is the litatom **FEBRUARY**, the **SELECTQ** returns 28 or 29 (depending on **(LEAPYEARP)**); otherwise if **MONTH** is **APRIL, JUNE, SEPTEMBER**, or **NOVEMBER**, the **SELECTQ** returns 30; otherwise it returns 31.

SELECTQ compiles open, and is therefore very fast; however, it will not work if the value of *X* is a list, a large integer, or floating point number, since **SELECTQ** uses **EQ** for all comparisons.

Note: **SELCHARQ** (page 2.15) is a version of **SELECTQ** that recognizes **CHARCODE** litatoms.

(SELECTC *X CLAUSE*$_1$ *CLAUSE*$_2$ ... *CLAUSE*$_K$ *DEFAULT***)** [NLambda NoSpread Function]

"**SELECTQ**-on-Constant." Similar to **SELECTQ** except that the selection keys are evaluated, and the result used as a **SELECTQ**-style selection key.

SELECTC is compiled as a **SELECTQ**, with the selection keys evaluated at compile-time. Therefore, the selection keys act like compile-time constants (see page 18.7). For example:

```
[SELECTC NUM
  ((for X from 1 to 9 collect (TIMES X X)) "SQUARE")
  "HIP"]
```

compiles as:

```
[SELECTQ NUM
  ((1 4 9 16 25 36 49 64 81) "SQUARE")
  "HIP"]
```

9.7 PROG and Associated Control Functions

(PROG1 X_1 X_2 ... X_N**)** [NLambda NoSpread Function]

Evaluates its arguments in order, and returns the value of its first argument X_1. For example, **(PROG1 X (SETQ X Y))** sets **X** to **Y**, and returns **X**'s original value.

(PROG2 X_1 X_2 ... X_N**)** [NoSpread Function]

Similar to **PROG1**. Evaluates its arguments in order, and returns the value of its second argument X_2.

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

PROGN evaluates each of its arguments in order, and returns the value of its last argument. **PROGN** is used to specify more than one computation where the syntax allows only one, e.g., **(SELECTQ ... (PROGN ...))** allows evaluation of several expressions as the default condition for a **SELECTQ**.

(PROG *VARLST* E_1 E_2 ... E_N**)** [NLambda NoSpread Function]

This function allows the user to write an ALGOL-like program containing Interlisp expressions (forms) to be executed. The first argument, *VARLST*, is a list of local variables (must be **NIL** if no variables are used). Each atom in *VARLST* is treated as the name of a local variable and bound to **NIL**. *VARLST* can also contain lists of the form (*LITATOM FORM*). In this case, *LITATOM* is the name of the variable and is bound to the value of *FORM*. The evaluation takes place before any of the bindings are performed, e.g., **(PROG ((X Y) (Y X)) ...)** will bind local variable **X** to the value of **Y** (evaluated *outside* the **PROG**) and local variable **Y** to the value of **X** (outside the **PROG**). An attempt to use anything other than a litatom as a **PROG** variable will cause an error, **ARG NOT LITATOM**. An attempt to use **NIL** or **T** as a **PROG** variable will cause an error, **ATTEMPT TO BIND NIL OR T**.

The rest of the **PROG** is a sequence of non-atomic statements (forms) and litatoms (labels). The forms are evaluated sequentially; the labels serve only as markers. The two special functions **GO** and **RETURN** alter this flow of control as described below. The value of the **PROG** is usually specified by the function **RETURN**. If no **RETURN** is executed before the **PROG** "falls off the end," the value of the **PROG** is **NIL**.

(GO *U***)** [NLambda NoSpread Function]

GO is used to cause a transfer in a **PROG**. **(GO L)** will cause the **PROG** to evaluate forms starting at the label **L** (**GO** does not evaluate its argument). A **GO** can be used at any level in a **PROG**. If the label is not found, **GO** will search higher progs *within the same function,* e.g., **(PROG ... A ... (PROG ... (GO A)))**. If the label is not found in the function in which the **PROG** appears, an error is generated, **UNDEFINED OR ILLEGAL GO**.

(RETURN *X***)** [Function]

A **RETURN** is the normal exit for a **PROG**. Its argument is evaluated and is immediately returned the value of the **PROG** in which it appears.

Note: If a **GO** or **RETURN** is executed in an interpreted function which is not a **PROG**, the **GO** or **RETURN** will be executed in the last interpreted **PROG** entered if any, otherwise cause an error.

GO or **RETURN** inside of a compiled function that is not a **PROG** is not allowed, and will cause an error at compile time.

As a corollary, **GO** or **RETURN** in a functional argument, e.g., to **SORT**, will not work compiled. Also, since **NLSETQ**'s and **ERSETQ**'s compile as *separate* functions, a **GO** or **RETURN** *cannot* be used inside of a compiled **NLSETQ** or **ERSETQ** if the corresponding **PROG** is outside, i.e., above, the **NLSETQ** or **ERSETQ**.

(LET *VARLST* E_1 E_2 ... E_N**)** [Macro]

LET is essentially a **PROG** that can't contain **GO**'s or **RETURN**'s, and whose last form is the returned value.

(LET* *VARLST* E_1 E_2 ... E_N**)** [Macro]

(PROG* *VARLST* E_1 E_2 ... E_N**)** [Macro]

LET* and **PROG*** differ from **LET** and **PROG** only in that the binding of the bound variables is done "sequentially." Thus

```
(LET* ((A (LIST 5))
       (B (LIST A A)))
      (EQ A (CADR B)))
```

would evaluate to **T**; whereas the same form with **LET** might even find **A** an unbound variable when evaluating **(LIST A A)**.

9.8 The Iterative Statement

The iterative statement (i.s.) in its various forms permits the user to specify complicated iterative statements in a straightforward and visible manner. Rather than the user having to perform the mental transformations to an equivalent Interlisp form using **PROG, MAPC, MAPCAR,** etc., the system does it for him. The goal was to provide a robust and tolerant facility which could "make sense" out of a wide class of iterative statements. Accordingly, the user should not feel obliged to read and understand in detail the description of each operator given below in order to use iterative statements.

An iterative statement is a form consisting of a number of special words (known as i.s. operators or i.s.oprs), followed by operands. Many i.s.oprs (**FOR, DO, WHILE,** etc.) are similar to iterative statements in other programming languages; other i.s.oprs (**COLLECT, JOIN, IN,** etc.) specify useful operations in a Lisp environment. Lower case versions of i.s.oprs (**do, collect,** etc.)

can also be used. Here are some examples of iterative statements:

```
←(for X from 1 to 5 do (PRINT 'FOO))
FOO
FOO
FOO
FOO
FOO
NIL
←(for X from 2 to 10 by 2 collect (TIMES X X))
(4 16 36 64 100)
←(for X in '(A B 1 C 6.5 NIL (45)) count (NUMBERP X))
2
```

Iterative statements are implemented through CLISP, which translates the form into the appropriate **PROG**, **MAPCAR**, etc. Iterative statement forms are translated using all CLISP declarations in effect (standard/fast/undoable/ etc.); see page 21.12. Misspelled i.s.oprs are recognized and corrected using the spelling list **CLISPFORWORDSPLST**. The order of appearance of operators is never important; CLISP scans the entire statement before it begins to construct the equivalent Interlisp form. New i.s.oprs can be defined as described on page 9.20.

If the user defines a function by the same name as an i.s.opr (**WHILE**, **TO**, etc.), the i.s.opr will no longer have the CLISP interpretation when it appears as **CAR** of a form, although it will continue to be treated as an i.s.opr if it appears in the interior of an iterative statement. To alert the user, a warning message is printed, e.g., **(WHILE DEFINED, THEREFORE DISABLED IN CLISP)**.

9.8.1 I.s.types

The following i.s.oprs are examples of a certain kind of iterative statement operator called an i.s.type. The i.s.type specifies what is to be done at each iteration. Its operand is called the "body" of the iterative statement. Each iterative statement must have one and only one i.s.type.

DO *FORM* [I.S. Operator]

Specifies what is to be done at each iteration. **DO** with no other operator specifies an infinite loop. If some explicit or implicit terminating condition is specified, the value of the i.s. is **NIL**. Translates to **MAPC** or **MAP** whenever possible.

COLLECT *FORM* [I.S. Operator]

Specifies that the value of *FORM* at each iteration is to be collected in a list, which is returned as the value of the i.s. when it

terminates. Translates to **MAPCAR**, **MAPLIST** or **SUBSET** whenever possible.

When **COLLECT** translates to a **PROG** (e.g., if **UNTIL**, **WHILE**, etc. appear in the i.s.), the translation employs an open **TCONC** using two pointers similar to that used by the compiler for compiling **MAPCAR**. To disable this translation, perform **(CLDISABLE 'FCOLLECT)** (see page 21.26).

JOIN *FORM* [I.S. Operator]

Similar to **COLLECT**, except that the values of *FORM* at each iteration are **NCONC**ed. Translates to **MAPCONC** or **MAPCON** whenever possible. /**NCONC**, /**MAPCONC**, and /**MAPCON** are used when the CLISP declaration **UNDOABLE** is in effect.

SUM *FORM* [I.S. Operator]

Specifies that the values of *FORM* at each iteration be added together and returned as the value of the i.s., e.g., **(for I from 1 to 5 sum (TIMES I I))** returns 1 + 4 + 9 + 16 + 25 = 55. **IPLUS**, **FPLUS**, or **PLUS** will be used in the translation depending on the CLISP declarations in effect.

COUNT *FORM* [I.S. Operator]

Counts the number of times that *FORM* is true, and returns that count as its value.

ALWAYS *FORM* [I.S. Operator]

Returns **T** if the value of *FORM* is non-**NIL** for all iterations. (Note: returns **NIL** as soon as the value of *FORM* is **NIL**).

NEVER *FORM* [I.S. Operator]

Similar to **ALWAYS**, except returns **T** if the value of *FORM* is *never* true. (Note: returns **NIL** as soon as the value of *FORM* is non-**NIL**).

The following i.s.types explicitly refer to the iteration variable (i.v.) of the iterative statement, which is a variable set at each iteration. This is explained below under **FOR**.

THEREIS *FORM* [I.S. Operator]

Returns the first value of the i.v. for which *FORM* is non-**NIL**, e.g., **(for X in Y thereis (NUMBERP X))** returns the first number in **Y**. (Note: returns the value of the i.v. as soon as the value of *FORM* is non-**NIL**).

LARGEST *FORM* [I.S. Operator]

SMALLEST *FORM* [I.S. Operator]

Returns the value of the i.v. that provides the largest/smallest value of *FORM*. **$$EXTREME** is always bound to the current greatest/smallest value, **$$VAL** to the value of the i.v. from which it came.

9.8.2 Iteration Variable I.s.oprs

FOR *VAR* [I.S. Operator]

Specifies the iteration variable (i.v.) which is used in conjunction with **IN, ON, FROM, TO,** and **BY**. The variable is rebound within the i.s., so the value of the variable outside the i.s. is not effected. Example:

```
←(SETQ X 55)
55
←(for X from 1 to 5 collect (TIMES X X))
(1 4 9 16 25)
←X
55
```

FOR *VARS* [I.S. Operator]

VARS a list of variables, e.g., **(for (X Y Z) in ...)**. The first variable is the i.v., the rest are dummy variables. See **BIND** below.

FOR OLD *VAR* [I.S. Operator]

Similar to **FOR**, except that *VAR* is *not* rebound within the i.s., so the value of the i.v. outside of the i.s. is changed. Example:

```
←(SETQ X 55)
55
←(for old X from 1 to 5 collect (TIMES X X))
(1 4 9 16 25)
←X
6
```

BIND *VAR* [I.S. Operator]

BIND *VARS* [I.S. Operator]

Used to specify dummy variables, which are bound locally within the i.s.

Note: **FOR, FOR OLD,** and **BIND** variables can be initialized by using the form *VAR*←*FORM*:

(for old (X←*FORM*) bind (Y←*FORM*) ...)

IN *FORM* [I.S. Operator]

Specifies that the i.s. is to iterate down a list with the i.v. being reset to the corresponding element at each iteration. For example, **(for X in Y do ...)** corresponds to **(MAPC Y (FUNCTION (LAMBDA (X) ...)))**. If no i.v. has been specified, a dummy is supplied, e.g., **(in Y collect CADR)** is equivalent to **(MAPCAR Y (FUNCTION CADR))**.

ON *FORM* [I.S. Operator]

Same as **IN** except that the i.v. is reset to the corresponding *tail* at each iteration. Thus **IN** corresponds to **MAPC**, **MAPCAR**, and **MAPCONC**, while **ON** corresponds to **MAP**, **MAPLIST**, and **MAPCON**.

Note: for both **IN** and **ON**, *FORM* is evaluated before the main part of the i.s. is entered, i.e. *outside* of the scope of any of the bound variables of the i.s. For example, **(for X bind (Y←'(1 2 3)) in Y ...)** will map down the list which is the value of **Y** evaluated *outside* of the i.s., *not* **(1 2 3)**.

IN OLD *VAR* [I.S. Operator]

Specifies that the i.s. is to iterate down *VAR*, with *VAR* itself being reset to the corresponding tail at each iteration, e.g., after **(for X in old L do ... until ...)** finishes, **L** will be some tail of its original value.

IN OLD (*VAR*←*FORM***)** [I.S. Operator]

Same as **IN OLD** *VAR*, except *VAR* is first set to value of *FORM*.

ON OLD *VAR* [I.S. Operator]

Same as **IN OLD** *VAR* except the i.v. is reset to the current value of *VAR* at each iteration, instead of to **(CAR** *VAR***)**.

ON OLD (*VAR*←*FORM***)** [I.S. Operator]

Same as **ON OLD** *VAR*, except *VAR* is first set to value of *FORM*.

INSIDE *FORM* [I.S. Operator]

Similar to **IN**, except treats first non-list, non-**NIL** tail as the last element of the iteration, e.g., **INSIDE '(A B C D . E)** iterates five times with the i.v. set to **E** on the last iteration. **INSIDE 'A** is equivalent to **INSIDE '(A)**, which will iterate once.

FROM *FORM* [I.S. Operator]

Used to specify an initial value for a numerical i.v. The i.v. is automatically incremented by 1 after each iteration (unless **BY** is specified). If no i.v. has been specified, a dummy i.v. is supplied and initialized, e.g., **(from 2 to 5 collect SQRT)** returns **(1.414 1.732 2.0 2.236)**.

TO *FORM* [I.S. Operator]

Used to specify the final value for a numerical i.v. If **FROM** is not specified, the i.v. is initialized to 1. If no i.v. has been specified, a dummy i.v. is supplied and initialized. If **BY** is not specified, the i.v. is automatically incremented by 1 after each iteration. When the i.v. is definitely being *incremented*, i.e., either **BY** is not specified, or its operand is a positive number, the i.s. terminates when the i.v. exceeds the value of *FORM*. Similarly, when the i.v. is definitely being decremented the i.s. terminates when the i.v. becomes *less* than the value of *FORM* (see description of **BY**).

Note: *FORM* is evaluated only once, when the i.s. is first entered, and its value bound to a temporary variable against which the i.v. is checked each interation. If the user wishes to specify an i.s. in which the value of the boundary condition is recomputed each iteration, he should use **WHILE** or **UNTIL** instead of **TO**.

Note: When both the operands to **TO** and **FROM** are numbers, and **TO**'s operand is less than **FROM**'s operand, the i.v. is decremented by 1 after each iteration. In this case, the i.s. terminates when the i.v. becomes *less* than the value of *FORM*. For example, **(from 10 to 1 do PRINT)** prints the numbers from 10 down to 1.

BY *FORM* (with **IN/ON**) [I.S. Operator]

If **IN** or **ON** have been specified, the value of *FORM* determines the *tail* for the next iteration, which in turn determines the value for the i.v. as described earlier, i.e., the new i.v. is **CAR** of the tail for **IN**, the tail itself for **ON**. In conjunction with **IN**, the user can refer to the current tail within *FORM* by using the i.v. or the operand for **IN/ON**, e.g., **(for Z in L by (CDDR** *Z***) ...)** or **(for Z in L by (CDDR** *L***) ...)**. At translation time, the name of the internal variable which holds the value of the current tail is substituted for the i.v. throughout *FORM*. For example, **(for X in Y by (CDR (MEMB 'FOO (CDR X))) collect X)** specifies that after each iteration, **CDR** of the current tail is to be searched for the atom **FOO**, and (**CDR** of) this latter tail to be used for the next iteration.

BY *FORM* (without **IN/ON**) [I.S. Operator]

If **IN** or **ON** have not been used, **BY** specifies how the i.v. itself is reset at each iteration. If **FROM** or **TO** have been specified, the

i.v. is known to be numerical, so the new i.v. is computed by adding the value of *FORM* (which is reevaluated each iteration) to the current value of the i.v., e.g., **(for N from 1 to 10 by 2 collect N)** makes a list of the first five odd numbers.

If *FORM* is a positive number (*FORM* itself, not its value, which in general CLISP would have no way of knowing in advance), the i.s. terminates when the value of the i.v. *exceeds* the value of **TO**'s operand. If *FORM* is a negative number, the i.s. terminates when the value of the i.v. becomes *less* than **TO**'s operand, e.g., **(for I from N to M by -2 until (LESSP I M) ...)**. Otherwise, the terminating condition for each iteration depends on the value of *FORM* for that iteration: if *FORM*<0, the test is whether the i.v. is less than **TO**'s operand, if *FORM*>0 the test is whether the i.v. exceeds **TO**'s operand, otherwise if *FORM* = 0, the i.s. terminates unconditionally.

If **FROM** or **TO** have not been specified and *FORM* is not a number, the i.v. is simply reset to the value of *FORM* after each iteration, e.g., **(for I from N by M ...)** is equivalent to **(for I←N by (PLUS I M) ...)**.

AS *VAR* [I.S. Operator]

Used to specify an iterative statement involving more than one iterative variable, e.g., **(for X in Y as U in V do ...)** corresponds to **MAP2C** (page 10.16). The i.s. terminates when any of the terminating conditions are met, e.g., **(for X in Y as I from 1 to 10 collect X)** makes a list of the first ten elements of **Y**, or however many elements there are on **Y** if less than 10.

The operand to **AS**, *VAR*, specifies the new i.v. For the remainder of the i.s., or until another **AS** is encountered, all operators refer to the new i.v. For example, **(for I from 1 to N1 as J from 1 to N2 by 2 as K from N3 to 1 by -1 ...)** terminates when **I** exceeds **N1**, or **J** exceeds **N2**, or **K** becomes less than 1. After each iteration, **I** is incremented by 1, **J** by 2, and **K** by -1.

OUTOF *FORM* [I.S. Operator]

For use with generators (page 11.16). On each iteration, the i.v. is set to successive values returned by the generator. The i.s. terminates when the generator runs out.

9.8.3 Condition I.s.oprs

WHEN *FORM* [I.S. Operator]

Provides a way of excepting certain iterations. For example, **(for X in Y collect X when (NUMBERP X))** collects only the elements of **Y** that are numbers.

UNLESS *FORM* [I.S. Operator]

Same as **WHEN** except for the difference in sign, i.e., **WHEN Z** is the same as **UNLESS (NOT Z)**.

WHILE *FORM* [I.S. Operator]

Provides a way of terminating the i.s. **WHILE** *FORM* evaluates *FORM before* each iteration, and if the value is **NIL**, exits.

UNTIL *FORM* [I.S. Operator]

Same as **WHILE** except for difference in sign, i.e., **WHILE X** is equivalent to **UNTIL (NOT X)**.

UNTIL *N* (*N* a number) [I.S. Operator]

Equivalent to **UNTIL** *I.V.* > *N*.

REPEATWHILE *FORM* [I.S. Operator]

Same as **WHILE** except the test is performed after the evalution of the body, but before the i.v. is reset for the next iteration.

REPEATUNTIL *FORM* [I.S. Operator]

Same as **UNTIL**, except the test is performed after the evaluation of the body.

REPEATUNTIL *N* (*N* a number) [I.S. Operator]

Equivalent to **REPEATUNTIL** *I.V.* > *N*.

9.8.4 Other I.s.oprs

FIRST *FORM* [I.S. Operator]

FORM is evaluated once before the first iteration, e.g., **(for X Y Z in L first (FOO Y Z) ...)**, and **FOO** could be used to initialize **Y** and **Z**.

FINALLY *FORM* [I.S. Operator]

FORM is evaluated after the i.s. terminates. For example, **(for X in L bind Y←0 do (if (ATOM X) then (SETQ Y (PLUS Y 1))) finally (RETURN Y))** will return the number of atoms in **L**.

EACHTIME *FORM* [I.S. Operator]

FORM is evaluated at the beginning of each iteration before, and regardless of, any testing. For example, consider,

```
(for I from 1 to N
  do (... (FOO I) ...)
```

unless (... (FOO I) ...)
until (... (FOO I) ...))

The user might want to set a temporary variable to the value of **(FOO I)** in order to avoid computing it three times each iteration. However, without knowing the translation, he would not know whether to put the assignment in the operand to **DO**, **UNLESS**, or **UNTIL**, i.e., which one would be executed first. He can avoid this problem by simply writing **EACHTIME (SETQ J (FOO I))**.

DECLARE: *DECL* [I.S. Operator]

Inserts the form **(DECLARE** *DECL***)** immediately following the **PROG** variable list in the translation, or, in the case that the translation is a mapping function rather than a **PROG**, immediately following the argument list of the lambda expression in the translation. This can be used to declare variables bound in the iterative statement to be compiled as local or special variables (see page 18.5). For example **(for X in Y declare: (LOCALVARS X) ...)**. Several **DECLARE:**s can apppear in the same i.s.; the declarations are inserted in the order they appear.

DECLARE *DECL* [I.S. Operator]

Same as **DECLARE:**.

Note that since **DECLARE** is also the name of a function, **DECLARE** cannot be used as an i.s. operator when it appears as **CAR** of a form, i.e. as the first i.s. operator in an iterative statement. However, **declare** (lower-case version) *can* be the first i.s. operator.

ORIGINAL *I.S.OPR OPERAND* [I.S. Operator]

I.S.OPR will be translated using its original, built-in interpretation, independent of any user defined i.s. operators. See page 9.20.

There are also a number of i.s.oprs that make it easier to create iterative statements that use the clock, looping for a given period of time. See timers, page 12.16.

9.8.5 Miscellaneous Hints on I.S.Oprs

- Lowercase versions of all i.s. operators are equivalent to the uppercase, e.g., **(for X in Y ...)** is equivalent to **(FOR X IN Y ...)**.
- Each i.s. operator is of lower precedence than all Interlisp forms, so parentheses around the operands can be omitted, and will be supplied where necessary, e.g., **BIND (X Y Z)** can be written **BIND**

X Y Z, OLD (X←*FORM*) as **OLD X←*FORM*, WHEN (NUMBERP X)** as **WHEN NUMBERP X**, etc.

- **RETURN** or **GO** may be used in any operand. (In this case, the translation of the iterative statement will always be in the form of a **PROG**, never a mapping function.) **RETURN** means return from the i.s. (with the indicated value), *not* from the function in which the i.s appears. **GO** refers to a label elsewhere in the function in which the i.s. appears, except for the labels **$$LP**, **$$ITERATE**, and **$$OUT** which are reserved, as described below.

- In the case of **FIRST**, **FINALLY**, **EACHTIME**, **DECLARE:** or one of the i.s.types, e.g., **DO**, **COLLECT**, **SUM**, etc., the operand can consist of more than one form, e.g., **COLLECT (PRINT (CAR X)) (CDR X)**, in which case a **PROGN** is supplied.

- Each operand can be the name of a function, in which case it is applied to the (last) i.v., e.g., **(for X in Y do PRINT when NUMBERP)** is the same as **(for X in Y do (PRINT X) when (NUMBERP X))**. Note that the i.v. need not be explicitly specified, e.g., **(in Y do PRINT when NUMBERP)** will work.

  For i.s.types, e.g., **DO**, **COLLECT**, **JOIN**, the function is always applied to the first i.v. in the i.s., whether explicity named or not. For example, **(in Y as I from 1 to 10 do PRINT)** prints elements on **Y**, not integers between 1 and 10.

  Note that this feature does not make much sense for **FOR**, **OLD**, **BIND**, **IN**, or **ON**, since they "operate" before the loop starts, when the i.v. may not even be bound.

  In the case of **BY** in conjunction with **IN**, the function is applied to the current *tail* e.g., **(for X in Y by CDDR ...)** is the same as **(for X in Y by (CDDR X) ...)**.

- While the exact form of the translation of an iterative statement depends on which operators are present, a **PROG** will always be used whenever the i.s. specifies dummy variables, i.e., if a **BIND** operator appears, or there is more than one variable specified by a **FOR** operator, or a **GO**, **RETURN**, or a reference to the variable **$$VAL** appears in any of the operands. When a **PROG** is used, the form of the translation is:

```
(PROG VARIABLES
   {initialize}
$$LP {eachtime}
   {test}
   {body}
$$ITERATE
   {aftertest}
   {update}
   (GO $$LP)
$$OUT {finalize}
   (RETURN $$VAL))
```

where {test} corresponds to that portion of the loop that tests for termination and also for those iterations for which {body} is not going to be executed, (as indicated by a **WHEN** or **UNLESS**); {body} corresponds to the operand of the i.s.type, e.g., **DO**, **COLLECT**, etc.; {aftertest} corresponds to those tests for termination specified by **REPEATWHILE** or **REPEATUNTIL**; and {update} corresponds to that part that resets the tail, increments the counter, etc. in preparation for the next iteration. {initialize}, {finalize}, and {eachtime} correspond to the operands of **FIRST**, **FINALLY**, and **EACHTIME**, if any.

Note that since {body} always appears at the top level of the **PROG**, the user can insert labels in {body}, and **GO** to them from within {body} or from other i.s. operands, e.g., **(for X in Y first (GO A) do (FOO) A (FIE))**. However, since {body} is dwimified as a list of forms, the label(s) should be added to the dummy variables for the iterative statement in order to prevent their being dwimified and possibly "corrected", e.g., **(for X in Y bind A first (GO A) do (FOO) A (FIE))**. The user can also **GO** to **$$LP**, **$$ITERATE**, or **$$OUT**, or explicitly set **$$VAL**.

9.8.6 Errors in Iterative Statements

An error will be generated and an appropriate diagnostic printed if any of the following conditions hold:

(1) Operator with null operand, i.e., two adjacent operators, as in **(for X in Y until do ...)**

(2) Operand consisting of more than one form (except as operand to **FIRST**, **FINALLY**, or one of the i.s.types), e.g., **(for X in Y (PRINT X) collect ...)**.

(3) **IN**, **ON**, **FROM**, **TO**, or **BY** appear twice in same i.s.

(4) Both **IN** and **ON** used on same i.v.

(5) **FROM** or **TO** used with **IN** or **ON** on same i.v.

(6) More than one i.s.type, e.g., a **DO** and a **SUM**.

In 3, 4, or 5, an error is not generated if an intervening **AS** occurs.

If an error occurs, the i.s. is left unchanged.

If no **DO**, **COLLECT**, **JOIN** or any of the other i.s.types are specified, CLISP will first attempt to find an operand consisting of more than one form, e.g., **(for X in Y (PRINT X) when ATOM X ...)**, and in this case will insert a **DO** after the first form. (In this case, condition 2 is not considered to be met, and an error is not generated.) If CLISP cannot find such an operand, and no **WHILE** or **UNTIL** appears in the i.s., a warning message is printed: **NO DO, COLLECT, OR JOIN:** followed by the i.s.

Similarly, if no terminating condition is detected, i.e., no **IN**, **ON**, **WHILE**, **UNTIL**, **TO**, or a **RETURN** or **GO**, a warning message is printed: **POSSIBLE NON-TERMINATING ITERATIVE STATEMENT:** followed by the iterative statement. However, since the user may be planning to terminate the i.s. via an error, control-E, or a **RETFROM** from a lower function, the i.s. is still translated. Note: The error message is not printed if the value of **CLISPI.S.GAG** is **T** (initially **NIL**).

9.8.7 Defining New Iterative Statement Operators

The following function is available for defining new or redefining existing iterative statement operators:

(I.S.OPR *NAME FORM OTHERS EVALFLG***)** [Function]

NAME is the name of the new i.s.opr. If *FORM* is a list, *NAME* will be a new *i.s.type* (see page 9.10), and *FORM* its body.

OTHERS is an (optional) list of additional i.s. operators and operands which will be added to the i.s. at the place where *NAME* appears. If *FORM* is **NIL**, *NAME* is a new i.s.opr defined entirely by *OTHERS*.

In both *FORM* and *OTHERS*, the atom **$$VAL** can be used to reference the value to be returned by the i.s., **I.V.** to reference the current i.v., and **BODY** to reference *NAME*'s operand. In other words, the current i.v. will be substituted for all instances of **I.V.** and *NAME*'s operand will be substituted for all instances of **BODY** throughout *FORM* and *OTHERS*.

If *EVALFLG* is **T**, *FORM* and *OTHERS* are evaluated at translation time, and their values used as described above. A dummy variable for use in translation that does not clash with a dummy variable already used by some other i.s. operators can be obtained by calling **(GETDUMMYVAR)**. **(GETDUMMYVAR T)** will return a dummy variable and also insure that it is bound as a **PROG** variable in the translation.

If *NAME* was previously an i.s.opr and is being redefined, the message **(***NAME* **REDEFINED)** will be printed (unless **DFNFLG** = **T**), and all expressions using the i.s.opr *NAME* that have been translated will have their translations discarded.

The following are some examples of how **I.S.OPR** could be called to define some existing i.s.oprs, and create some new ones:

COLLECT

```
(I.S.OPR 'COLLECT
    '(SETQ $$VAL (NCONC1 $$VAL BODY)))
```

SUM

```
(I.S.OPR 'SUM
    '($$VAL←$$VAL + BODY)
    '(FIRST $$VAL←0))
```

Note: **$$VAL + BODY** is used instead of **(IPLUS $$VAL BODY)** so that the choice of function used in the translation, i.e., **IPLUS**, **FPLUS**, or **PLUS**, will be determined by the declarations then in effect.

NEVER

```
(I.S.OPR 'NEVER
    '(if BODY then $$VAL←NIL (GO $$OUT)))
```

Note: **(if BODY then (RETURN NIL))** would exit from the i.s. immediately and therefore not execute the operations specified via a **FINALLY** (if any).

THEREIS

```
(I.S.OPR 'THEREIS
    '(if BODY then $$VAL←I.V. (GO $$OUT)))
```

RCOLLECT To define **RCOLLECT**, a version of **COLLECT** which uses **CONS** instead of **NCONC1** and then reverses the list of values:

```
(I.S.OPR 'RCOLLECT
    '($$VAL←(CONS BODY $$VAL))
    '(FINALLY (RETURN (DREVERSE $$VAL))))]
```

TCOLLECT To define **TCOLLECT**, a version of **COLLECT** which uses **TCONC**:

```
(I.S.OPR 'TCOLLECT
    '(TCONC $$VAL BODY)
    '(FIRST $$VAL←(CONS) FINALLY (RETURN (CAR $$VAL))))]
```

PRODUCT

```
(I.S.OPR 'PRODUCT
    '($$VAL←$$VAL*BODY)
    '(FIRST $$VAL←1)]
```

UPTO To define **UPTO**, a version of **TO** whose operand is evaluated only once:

```
(I.S.OPR 'UPTO
    NIL
    '(BIND $$FOO←BODY TO $$FOO)]
```

TO To redefine **TO** so that instead of recomputing *FORM* each iteration, a variable is bound to the value of *FORM*, and then that variable is used:

```
(I.S.OPR 'TO
    NIL
    '(BIND $$END FIRST $$END←BODY ORIGINAL TO $$END)]
```

Note the use of **ORIGINAL** to redefine **TO** in terms of its original definition. **ORIGINAL** is intended for use in redefining built-in operators, since their definitions are not accessible, and hence

not directly modifiable. Thus if the operator had been defined by the user via **I.S.OPR**, **ORIGINAL** would not obtain its original definition. In this case, one presumably would simply modify the i.s.opr definition.

I.S.OPR can also be used to define synonyms for already defined i.s. operators by calling **I.S.OPR** with *FORM* an atom, e.g., **(I.S.OPR 'WHERE 'WHEN)** makes **WHERE** be the same as **WHEN**. Similarly, following **(I.S.OPR 'ISTHERE 'THEREIS)**, one can write **(ISTHERE ATOM IN Y)**, and following **(I.S.OPR 'FIND 'FOR)** and **(I.S.OPR 'SUCHTHAT 'THEREIS)**, one can write **(find X in Y suchthat X member Z)**. In the current system, **WHERE** is synonymous with **WHEN**, **SUCHTHAT** and **ISTHERE** with **THEREIS**, **FIND** with **FOR**, and **THRU** with **TO**.

If *FORM* is the atom **MODIFIER**, then *NAME* is defined as an i.s.opr which can immediately follow another i.s. operator (i.e., an error will not be generated, as described previously). *NAME* will not terminate the scope of the previous operator, and will be stripped off when **DWIMIFY** is called on its operand. **OLD** is an example of a **MODIFIER** type of operator. The **MODIFIER** feature allows the user to define i.s. operators similar to **OLD**, for use in conjunction with some other user defined i.s.opr which will produce the appropriate translation.

The file package command **I.S.OPRS** (page 17.39) will dump the definition of i.s.oprs. **(I.S.OPRS PRODUCT UPTO)** as a file package command will print suitable expressions so that these iterative statement operators will be (re)defined when the file is loaded.