M T S

The Michigan Terminal System

# UTILISP in MTS

**Volume 22**

**May 1988**

University of Michigan
Information Technology Division
Consulting and Support Services

DISCLAIMER

The MTS Manual is intended to represent the current state of the Michigan Terminal System (MTS), but because the system is constantly being developed, extended, and refined, sections of this volume will become obsolete. The user should refer to the *U–M Computing News*, Computing Center Memos, and future Updates to this volume for the latest information about changes to MTS.

# CONTENTS

# PREFACE

The software developed by the Information Technology Division staff for the operation of the high-speed IBM 370-compatible computers can be described as a multiprogramming supervisor that handles a number of resident, reentrant programs. Among them is a large subsystem, called MTS (Michigan Terminal System), for command interpretation, execution control, file management, and accounting maintenance. Most users interact with the computer's resources through MTS.

The MTS Manual is a series of volumes that describe in detail the facilities provided by the Michigan Terminal System. Administrative policies of the Information Technology Division and the physical facilities provided are described in other publications.

The MTS volumes now in print are listed below. The date indicates the most recent edition of each volume; however, since volumes are periodically updated, users should check the file *CCPUBLICATIONS, or watch for announcements in the *U–M Computing News*, to ensure that their MTS volumes are fully up to date.

| | |
|---|---|
| Volume 1 | *The Michigan Terminal System*, January 1984 |
| Volume 2 | *Public File Descriptions*, January 1987 |
| Volume 3 | *System Subroutine Descriptions*, April 1981 |
| Volume 4 | *Terminals and Networks in MTS*, March 1984 |
| Volume 5 | *System Services*, May 1983 |
| Volume 6 | *FORTRAN in MTS*, October 1983 |
| Volume 7 | *PL/I in MTS*, September 1982 |
| Volume 8 | *LISP and SLIP in MTS*, June 1976 |
| Volume 9 | *SNOBOL4 in MTS*, September 1975 |
| Volume 10 | *BASIC in MTS*, December 1980 |
| Volume 11 | *Plot Description System*, August 1978 |
| Volume 12 | *PIL/2 in MTS*, December 1974 |
| Volume 13 | *The Symbolic Debugging System*, September 1985 |
| Volume 14 | *360/370 Assemblers in MTS*, May 1983 |
| Volume 15 | *FORMAT and TEXT360*, April 1977 |
| Volume 16 | *ALGOL W in MTS*, September 1980 |
| Volume 17 | *Integrated Graphics System*, December 1980 |
| Volume 18 | *The MTS File Editor*, February 1988 |
| Volume 19 | *Tapes and Floppy Disks*, November 1986 |
| Volume 20 | *Pascal in MTS*, December 1985 |
| Volume 21 | *MTS Command Extensions and Macros*, April 1986 |
| Volume 22 | *Utilisp in MTS*, May 1988 |
| Volume 23 | *UTILISP in MTS*, March 1987 |

Other volumes are in preparation. The numerical order of the volumes does not necessarily reflect the chronological order of their appearance; however, in general, the higher the number, the more specialized the volume. Volume 1, for example, introduces the user to MTS and describes in general the MTS operating system, while Volume 10 deals exclusively with BASIC.

The attempt to make each volume complete in itself and reasonably independent of others in the series naturally results in a certain amount of repetition. Public file descriptions, for example, may appear in more than one volume. However, this arrangement permits the user to buy only those volumes that serve his or her immediate needs.

Richard A. Salisbury
General Editor

# PREFACE TO VOLUME 22

This is an interim edition of MTS Volume 22. In the future, sections will be added describing in further detail the use of Utilisp with MTS.[1]

The characters up-arrow and grave which are used in a few places in this volume currently are missing from the Textform fonts used to print the document. They will be supplied in the near future.

------------------
[1]If you believe this, you probably also believe in the Great Pumpkin.

# UTILISP

The UTILISP (University of Tokyo Interactive LISt Processor) system is designed for highly interactive programming and debugging of sophisticated programs.

This document is intended to serve both as a User's Guide and as a Reference Manual for the language and the system. It is hoped that those who are familiar with the Lisp language can acquire a complete knowledge of the system from this manual.

This is a preliminary version. Several sections are missing and these will be supplied in the near future. Please consult the author for information concerning these missing sections.

The author's address is as follows:

Takashi Chikayama
ICOT Institute for New Generation Computer Technology
Mita Kokusai Bldg. 21F
4–28 Mita 1–Chome
Minato–ku, Tokyo 108, Japan

Telephone: 03–456–3195

# INTRODUCTION

## GENERAL INFORMATION

The UTILISP (University of Tokyo Interactive LISt Processor) system is designed for highly interactive programming and debugging of sophisticated programs.

At present, the UTILISP system is in operation at the University of Michigan Computing Center, and is organized as an MTS program. The standard command sequence for running the system is currently as follows:

```
#Run *UTILISP
> (Lisp input)
```

Several optional parameters exist for the command LISP:

| | |
|---|---|
| STACK | STACK=n specifies that the size of the stack area is to be “n” pages (1 page = 4 kilobytes). The default value for “n” is 16. |
| FIX | FIX=n specifies that the size of the “fixed heap” area (for compiled codes) is to be “n” pages. The default value for “n” is 8. |
| SIZE | SIZE=n specifies that “n” pages are to be allocated for use by UTILISP. The program will not get more space if this is not enough. The default value for “n” is 64. |

If the logical I/O unit 0 has been assigned, the file associated with it is executed first. This execution is identical with that of the standard top-level Lisp loop, except that the results are not displayed.

After the execution of the unit 0 file (if any), the system enters the top-level loop. Each S-expression read in is evaluated and the result is displayed. Note that the top-level evaluator is EVAL, not EVALQUOTE.

The session can be terminated by calling the function QUIT. If one wishes to terminate the session abnormally, the function ABEND can be used.

There are cases in which these system functions are not recognized by the Lisp reader, e.g., when the READTABLE or OBVECTOR has been destroyed. In such cases, the Utilisp session can be terminated by ten consecutive exclamation marks (!!!!!!!!!!) at the beginning of an input line from the terminal.

In case an endless or unexpectedly long computation should occur, an attention interrupt from the terminal (usually by means of the break key) will stop the current computation and the system enters the BREAK loop. For details, see the section *Errors and Debugging*.

## NOTATIONAL CONVENTIONS AND NOTES ON SYNTAX

There are several conventions concerning notation which should be understood before reading the manual in order to avoid confusion.

In this manual, Lisp symbols are printed in uppercase characters. Lowercase words appearing in S-expressions represent certain Lisp objects, the details of which are irrelevant or explained elsewhere. They appear between the symbols "<" and ">", e.g., <sexpr>.

In what follows, a Lisp object whose "car" is <a> and "cdr" is <b> may sometimes be written in the form (a . b). However, note that <a> and, especially, <b> are not necessarily atoms. Thus, a list beginning with the symbol PROGN may be written in the form (PROGN . body), where <body> is a list following PROGN. Similarly, in titles of descriptions of functions, "PLUS . args" for example, <args> indicates a list of arguments following the function PLUS.

Lisp symbols appearing as titles are followed by a term indicating its category within curly brackets, "{" and "}", (if it is not an ordinary function) and a description of its arguments (if it is not a variable). Specifically, the categories are "Special Form", "Macro", and "Variable". The following examples illustrate the manner in which the arguments are described:

QUOTE {Special Form} arg

QUOTE is a "special form" and takes one argument.

CONS x y

CONS is an ordinary function and requires two arguments, <x> and <y>, and their absence generates an error.

GENSYM (prefix) (begin)

GENSYM may take zero to two arguments; <prefix> and <begin> are optional.

PLUS . args

PLUS may take arbitrarily many (possibly zero) arguments.

– arg . args

– may take arbitrarily many (but at least one) arguments.

As in the examples, argument names appear between "<" and ">" in the description of the function.

The symbol "=>" will be used to indicate evaluation in examples, e.g., "FOO => NIL" means that "the result of evaluating FOO is NIL".

There are several terms which are widely used in this manual but will not be rigorously defined. They are: "S-expression", which means a Lisp object, especially in its printed representation; "dotted pair", which means a "cons"; and "atom", which means a Lisp object other than a "cons". Note that an atom does not necessarily mean a symbolic atom; it may be a number, string, etc. It is recommended that those who are not familiar with these terms consult an appropriate Lisp textbook.

Several characters have special meanings in UTILISP, i.e., single quote ('), double quote ("), backquote (`), comma (,), semicolon (;), and slant (/).

Semicolons are used for comments. When the Lisp reader encounters a semicolon, it ignores all the characters remaining on the current line and resumes reading from the beginning of the next line. In such a case, a blank space is automatically introduced between the last symbol preceding the semicolon and the first symbol on the next line. However, a semicolon can occur as an element of a string (see remarks on double quotes below).

A single quote "'" has the same effect as the special form QUOTE (see below). For example, 'FOO is read as (QUOTE FOO), and '(CONS 'FOO 'BAR) is read as (QUOTE (CONS (QUOTE FOO) (QUOTE BAR))), etc.

Slants are used for quoting characters possessing special functions so that they are merely interpreted as normal alphabetic characters. For example, /'FOO is read as a symbol whose print name is "'FOO" and not as (QUOTE FOO). Thus, one must type "//" to convey the symbol "/" to the Lisp reader.

Double quotes are used for indicating strings. Any characters occurring between a double quote and the next double quote are read as a string. Double quotes occurring inside strings should be typed twice. For example, """" represents a string consisting of one double quote. A string may extend beyond the ends of a line.

Concerning backquotes and commas, see the section *Macros*.

## DATA TYPES

There are nine data types in this system, i.e., symbol, cons, fixnum, flonum, string, vector, reference, stream, and code piece.

A symbol has a "print name", a "value" (sometimes called a "binding"), a definition, and a property list. The "print name" is a string which is the value of the function PNAME when applied to the symbol in question; this string serves as the printed representation of the symbol. The "value" may be any Lisp object, and is interpreted as the value of the symbol when the latter is used as a variable. The symbol may also be in "unbound" state, in which case, it has no value at all. Access to the value of a symbol is effected by evaluating the symbol, and the value may be updated by using the function SET. The definition is functional attributes of the symbol; access is effected by GETD and updating by PUTD. The property list contains an even number of elements (possibly zero); direct access and updating can be effected by PLIST and SETPLIST, respectively, but it is usually more convenient to use the functions GET (for access), PUTPROP (for adding and updating properties), and REMPROP (for removing properties). SYMBOL is the basic function for creating a new symbol with a certain print name. All symbols which are normally read in are registered in a table called "obvector", and any of these which bear the same name are identified by means of the function INTERN (for details, see the section *Input and Output*). The function GENSYM serves to generate a sequence of distinct symbols.

A "cons" is a Lisp object possessing two components, "car" and "cdr", which can be any Lisp objects. Access to these two components is effected by the functions CAR and CDR, respectively, and updating by RPLACA and RPLACD, respectively. A "cons" may be constructed by means of the function CONS.

There are two kinds of numerical objects in this system, upon which arithmetical operations may be performed; one is called “fixnum” which possesses 24-bit signed integer value; the other is called “flonum” which possesses 64-bit floating-point value with the accuracy of about 15 decimal digits.

A “string” is a finite sequence of characters. Each character has an 8-bit code value which is usually interpreted in terms of the EBCDIK code. Independent access to and updating of these characters can be effected by means of the functions SREF and SSET, respectively. The length of a string can be obtained by applying the function STRING–LENGTH.

A “vector” is a finite ordered set of Lisp objects. Vectors can be created by means of the function VECTOR. The elements of such a vector may be any Lisp objects; access is effected by means of the function VREF and updating by the function VSET. The length of a vector can be obtained by using the function VECTOR–LENGTH.

A “reference” is a pointer indicating an element of a vector. It is often useful to have access to and update elements of vectors. A reference can be created by the function REFERENCE; access to and updating of the corresponding element can be effected by means of the functions DEREF and SETREF, respectively.

A “stream” is an object related to I/O. All the I/O operations in this system are carried out by means of such intermediary streams, which are created by the function STREAM.

A “code piece” is a segment of machine code which constitutes the body of a predefined or a compiled function. Code pieces have names, normally a symbol, access to which can be effected by means of the function FUNCNAME.

## LAMBDA LISTS

A LAMBDA-expression is the format specifying an interpreted function in Lisp, and is of the form

```
    (LAMBDA lambda-list . body)
```

where <body> is a list of forms. Usually, <lambda-list> is a list of symbols which corresponds to the so-called formal parameter list in certain other programming languages. When a LAMBDA-expression is applied to given values of the arguments (actual parameters), the symbols are bound to these values, and the forms constituting <body> are evaluated sequentially and the result of the last of these evaluations is returned as the final result of the application. The formal parameters are then unbound and revert to the state they were in prior to the application. If the number of actual arguments is not equal to the length of <lambda-list>, an error is generated.

In UTILISP, an element of <lambda-list> can be either a symbol or a list of the form

```
    (symbol . defaults)
```

When the number of actual arguments to which the function is applied is less than the length of <lambda-list>, the given actual arguments are first bound to the corresponding symbols. The remaining elements of <lambda-list> must have the form (symbol . defaults). Here, <defaults> is a list of forms which are evaluated sequentially and the result of the last one (or NIL in the case when <defaults> is empty) is bound to <symbol>. If an actual argument corresponding to a symbol associated with a list <defaults> is supplied, the symbol is bound to this actual argument and the associated list <defaults> is merely ignored.

Default values are evaluated after the binding of the preceding arguments, hence, they may depend upon the results of the preceding bindings.

Some examples of lambda-lists are as follows:

(A B C)
A, B, and C are all required.

(A B (C))
A and B are required but C is optional; the default value of C is NIL.

(A B (C 0))
and B are required but C is optional; the default value of C is 0.

(A B (C (PRINT "Default value is used for C.") 0))
A and B are required and C is optional; when the default value is used, the indicated message is printed.

(A B (C (CONS A B)))
A and B are required and C is optional; the default value of C depends upon A and B.

# PREDICATES

A predicate is a function which tests the validity of some condition involving its arguments and returns the symbol T if the condition holds, and the symbol NIL if it does not.

When a Lisp object is used as a logical value, it is interpreted as “false” if, and only if, it is NIL; all Lisp objects other than NIL are interpreted as “true”.

## PREDICATES ON DATA TYPES

The following predicates are for testing data types. These predicates return T if the argument is of the type indicated by the name of the function, NIL if it is of some other type.

SYMBOLP arg

Returns T if <arg> is a symbol, otherwise NIL.

CONSP arg

Returns T if <arg> is a “cons”, otherwise NIL.

LISTP arg

LISTP is an alias of CONSP. This is incorporated mainly for compatibility with other Lisp systems.

ATOM arg

Returns T if <arg> is not a “cons”, otherwise NIL.

FIXP arg

Returns T if <arg> is a “fixnum” object, i.e., an integer number, otherwise NIL.

FLOATP arg

Returns T if <arg> is a “flonum” object, i.e., a floating-point number, otherwise NIL.

NUMBERP arg

Returns T if <arg> is a numerical object, i.e., either a “fixnum” or a “flonum”, otherwise NIL.

STRINGP arg

Returns T if <arg> is a string, otherwise NIL.

VECTORP arg

Returns T if <arg> is a vector, otherwise NIL.

REFERENCEP arg

Returns T if <arg> is a reference pointer, otherwise NIL.

STREAMP arg

Returns T if <arg> is a stream, otherwise NIL.

CODEP arg

Returns T if <arg> is a code piece, otherwise NIL.

## GENERAL-PURPOSE PREDICATES

The following functions are some other general-purpose predicates.

EQ x y

Returns T if <x> and <y> denote the same Lisp object, otherwise NIL. Lisp objects which have the same printed representation are not necessarily identical. However, the "interning" process ensures that two symbols with the same print name are identical (see the section *Input and Output* for details). Unlike some other Lisp systems, equality of values of integer numbers ("fixnums") can also be compared using EQ.

Note: In this manual, the expression "two Lisp objects are EQ" means that they are the same object.

NEQ x y

(NEQ x y) = (NOT (EQ x y)). This function is incorporated primarily for convenience in typing.

EQUAL x y

EQUAL returns T if <x> and <y> are "similar" Lisp objects, otherwise NIL. That is, two strings are EQUAL if they have the same length and all the characters in corresponding positions are the same, two "flonums" are EQUAL if they have the same floating-point value, and, inductively, two "cons" cells are EQUAL if their respective "cars" and "cdrs" are EQUAL. In all other cases, two objects are EQUAL if, and only if, they are EQ.

If two Lisp objects are EQUAL, they have the same printed representation, however, the reverse does not necessarily hold (e.g., for symbols which have not been "interned").

NOT x
NULL x

NOT returns the symbol T if <x> is EQ to NIL, and the symbol NIL otherwise. NULL is the same as NOT; both functions are incorporated for the sake of readability. It is recommended that NULL be used for checking whether a given value is NIL, and that

NOT be used for inverting a logical value.

The system also includes various predicates in addition to those introduced in this section. These will be described in the sections on the various data types accepted by these predicates; for example, the predicate ZEROP is described in the section *Numbers*.

# EVALUATION

## THE EVALUATOR

The process of evaluation of a Lisp form is as follows:

If the form is neither a symbol nor a “cons”, i.e., if it is a “fixnum”, “flonum”, a string, a code piece, a vector, a reference, or a stream, the result of its evaluation is simply the form itself.

If the form is a symbol, the result is the value to which that symbol is bound. If the symbol is unbound, an error is generated.

A so-called “special form” (i.e., a “cons” identified by a distinguished symbol in its “car”) is evaluated in a manner which depends upon the particular form in question. All of these special forms will be individually described in this manual.

If the form in question is not a so-called “special form”, it requires the application of a function or a macro to its arguments. The “car” of the form is a LAMBDA-expression or the name of a function. If the function is not a “macro”, the “cdr” of the form is a list of forms which are evaluated sequentially, from left to right, and the resulting arguments are then supplied to the function; the value finally returned is the result of applying the function to these arguments.

The evaluation process for macro forms is described in the section *Macros*.

A more detailed and accurate description of the evaluator will be supplied after various improvements of present implementation have been carried out.

## VARIOUS FUNCTIONS CONCERNED WITH EVALUATION

EVAL x

Evaluates <x>, and returns the result. Ordinarily, EVAL is not used explicitly, since evaluation is usually carried out implicitly. EVAL is primarily useful in programs concerning Lisp itself, rather than in its applications.

APPLY fn arglist

Applies the function <fn> to the set of arguments given by <arglist>, and returns the resulting value.

FUNCALL fn . args

Applies the function <fn> to the set of arguments <args>, and returns the resulting value. Note that the functional argument <fn> is evaluated in the usual way, while function, which constitutes the “car” of an ordinary Lisp application, is not. For example,

```
(SETQ CONS 'PLUS) => PLUS
(FUNCALL CONS 1 2) => 3
(CONS 1 2) => (1 . 2)
```

Thus, explicit application using FUNCALL instead of simple implicit function application, should be used for functional arguments, since the binding of the function is not examined by the evaluator in simple implicit function applications, whereas when FUNCALL is used the functional argument symbol is evaluated first yielding a function which is then applied in the ordinary manner.

QUOTE {Special Form} arg

Simply returns the argument <arg>. Its usefulness largely consists in the fact that its argument is not evaluated by the evaluator. For example,

```
(QUOTE X) => X
(SETQ X (QUOTE (CONS 1 2))) X => (CONS 1 2)
```

Since QUOTE is very frequently used, the Lisp Reader allows the user to reduce the burden of keying in the program by converting S-expressions preceded by a single quote character "'" into QUOTEd forms. For example,

```
(SETQ X '(CONS 1 2))
```

is converted into

```
(SETQ X (QUOTE (CONS 1 2)))
```

FUNCTION {Special Form} fn

The form (FUNCTION x) has precisely the same effect as (QUOTE x); these alternative forms are available for the sake of clarity in reading and writing programs. It is recommended that FUNCTION be used to quote a piece of a program, and that QUOTE be used for segments of data. The compiler utilizes this information to generate efficient object codes.

Note: Function-valued arguments in Lisp functions should be evoked using FUNCALL. See the description of FUNCALL (above) for details.

COMMENT {Special Form} . args

COMMENT ignores its arguments and always returns NIL; it is useful for inserting explanatory remarks.

PROGN {Special Form} . args

The arguments <args> are evaluated sequentially, from left to right, and the value of the final argument is returned. This operation is useful in cases where it is necessary to evaluate a number of forms for the sake of the concomitant side effects but only the value of the last form is required. Note that LAMBDA-expressions, COND forms, and many other control structure forms incorporate this property of PROGN implicitly (in the sense that multiple forms are handled in a similar manner).

PROG1 {Special Form} . args

PROG1 functions in the same manner as PROGN, except that it returns the value of the first argument rather than the last. PROG1 is most commonly used to evaluate a number of expressions, with possible occurrence of the side effects, and return a value which must be computed before the side effects occur. For example,

```
(SETQ X (PROG1 Y (SETQ Y X)))
```

This form interchanges the values of the variables X and Y.

PROG2 {Special Form} . args

The action of PROG2 is the same as that of PROGN and PROG1, except that it returns the value of its second argument. It is incorporated mainly for compatibility with other Lisp systems.

```
(PROG2 x y . z)
```

is equivalent to

```
(PROGN x (PROG1 y . z))
```

LET {Macro} bindings . body

A LET form has the syntax

```
(LET ((var1 vform1)
      (var2 vform2)
      ...)
    bform1
    bform2
    ...)
```

which is automatically converted into and effectively equivalent to the following form:

```
((LAMBDA (var1 var2 ...)
         bform1
         bform2
         ...)
    vform1
    vform2
    ...)
```

It is often preferable to use LET rather than to directly use LAMBDA, since the variables and the corresponding forms appear textually close to one another, which increases the readability of the program.

As LET forms are converted into LAMBDA application forms, all the values of the <vform>s are computed before binding any of these values to to the corresponding <var>s. For example, <vform2> cannot depend upon <var1>; that is, if <var1> appears in <vform2>, a variable named <var1> must have been bound somewhere outside this LET form.

LETS {Macro} bindings . body

LETS is similar to LET except that LETS binds its variables sequentially, one by one, while LET, as mentioned above, binds them simultaneously. (LETS is a contraction of “LET Sequentially”).

A LETS form has the syntax

```
(LETS ((var1 . vforms1)
       (var2 . vforms2)
       ...)
    bform1
    bform2
    ...)
```

which is effectively equivalent to:

```
((LAMBDA ((var1 . vforms1)
          (var2 . vforms2)
          ...)
    bform1
    bform2
    ...))
```

Each list <vform-i> constitutes the default value list for the corresponding <var-i>, and therefore can depend upon the preceding <var>s (see the section *Lambda Lists* for details).

Note: The interpretation of LETS is faster than that of LET. However, once compiled, their speeds become identical.

# FLOW OF CONTROL

The present system provides a variety of structures for the flow of control.

Function application is the basic method for constructing programs. Moreover, the definition of a function may always call the function being defined. This process is known as "recursion".

Both explicit and implicit PROGN structures can be used for sequential execution of programs. The forms in a PROGN structure are evaluated sequentially from left to right.

In this section, some even more flexible control structures are described. Conditional constructs are useful for making decisions, while iteration and mapping constructs may be convenient for repetition. There are also more flexible control structures known as non-local exits.

## CONDITIONALS

A conditional construct incorporates a decision in a program, resulting in the execution of one of several alternatives in accordance with certain logical conditions.

AND {Special Form} . args

Evaluates the arguments sequentially, from left to right. If the value of any argument is NIL, NIL is returned and the remaining arguments are not evaluated. If the value of all the arguments are non-NIL, the value of the last argument is returned. AND can be interpreted for logical operation, where NIL stands for "false" and non-NIL for "true". For example,

```
(AND X Y)
(AND (SETQ TEMP (ASSQ X Y))
     (RPLACD TEMP Z))
(AND ERROR-EXISTS (PRINC "There is an error!"))
```

Note: (AND) => T

OR {Special Form} . args

Evaluates the arguments sequentially, from left to right. If the value of any argument is NIL, the next argument is evaluated. If there are no remaining arguments, NIL is returned. However, if the value of any argument is non-NIL, that value is immediately returned and the remaining arguments, if any, are not evaluated. OR can be interpreted as a logical operation, where NIL stands for "false" and non-NIL for "true".

Note: (OR) => NIL

COND {Special Form} . clauses

The arguments of COND are usually referred to as "clauses". Each clause consists of a predicate followed by a number (possibly zero) of forms. The predicate is called the "antecedent" and the forms are called the "consequents".

Thus, a COND-form might have the following syntax:

```
(COND (antecedent consequent consequent ...)
      (antecedent)
      (antecedent consequent ...)
      ...)
```

Each clause represents an alternative which is selected if its antecedent is satisfied and the antecedents of all preceding clauses were not satisfied when evaluated.

The clauses are processed sequentially from left to right. First, the antecedent of the current clause is evaluated. If the result is NIL, the process advances to the next clause. Otherwise, the consequents are evaluated sequentially from left to right (in a PROGN manner), the value of the last consequent is returned, and the remaining clauses (if any) are not processed. If there were no consequents in the selected clause, the value of the antecedent is returned. If the clauses are exhausted, that is the value of every antecedent is NIL, the value of the COND form is NIL.

SELECTQ {Special Form} key-form . clauses

Many programs require multiplex branchings which depend on the value of some form. A typical example is as follows:

```
(COND ((EQ X 'FOO) ...)
      ((EQ X 'BAR) ...)
      ((MEMQ X '(BAZ QUUX MUM)) ...)
      (T ...))
```

SELECTQ is incorporated for convenience in such situations. A SELECTQ form has the following syntax:

```
(SELECTQ key-form
      (pattern consequent consequent ...)
      (pattern consequent consequent ...)
      (pattern consequent consequent ...)
      ...)
```

The first argument <key-form> is evaluated first (just once). The resulting value is called the “key”. The key form is followed by a number of “clauses”, each of which consists of a “pattern” followed by a number (possibly zero) of “consequent” forms. The pattern of each clause is compared with the key, and if it “matches”, the consequents of this clause are evaluated, and SELECTQ returns the value of the last consequent. If there are no “matches”, or if there is no consequent in the selected clause, NIL is returned. Note that the patterns are not evaluated.

The objects which may be used as the patterns and their “matching” conditions are as follows:

(1) Any atom (symbol, number, etc.), except the symbol T — In this case, the key matches if it is EQ to the atom.
(2) A list — In this case, the key matches if it is EQ to one of the top-level elements of the list.
(3) The symbol T — The symbol T constitutes a special pattern which matches anything.

The preceding example can be expressed as follows:

```
(SELECTQ X
     (FOO ...)
     (BAR ...)
     ((BAZ QUUX MUM) ...)
     (T ...))
```

Note: The symbol T itself may be used as the first component of a clause, in a non-trivial manner, by selecting (T) as the pattern.

## ITERATION

PROG {Special Form} locals . body

PROG is a special form which provides temporary variables, sequential evaluation of forms, and "goto" operations. A typical PROG form might have, for example, the following structure:

```
     (PROG (var1 (var2 . inits2) var3 (var4 . inits4))
tag1       statement1
           statement2
tag2       statement3
           ...)
```

<var1>, <var2>, ... are temporarily bound variables. The binding of these variables prior to the execution of the PROG are recorded, and when the execution of the PROG has been completed, the original bindings are restored. If a variable is associated with an initial value list <inits>, the elements of the list are evaluated sequentially, from left to right, and the value of the last one becomes the initial value of the variable. If there are no initial value forms, the variable is initialized to NIL. For example,

```
(PROG ((A T)
       B
       (C (PRINT "C is bound") (CAR '(FOO . BAR))))
            . body)
```

Here, the initial value of A is T, that of B is NIL, and that of C is the symbol FOO. Before the binding of C is executed, the indicated message is printed. The bindings are processed sequentially, and the value of each form may depend upon previous bindings.

The portion of a PROG which follows the variable list is called the "body". The elements of <body> may be atoms, which are called "tags", or "cons" cells, which are called "statements."

After the temporary variables have been bound, the forms in the body are processed sequentially. Tags are not evaluated, whereas statements are evaluated and their values discarded. If the process reaches the end of the body, NIL is returned. However, two special devices (described below) may be used to alter the flow of control in the body of a PROG.

If (RETURN x1 ... xn) is evaluated, processing of the body is terminated and the value of the last argument <xn> is returned as the value of the PROG. If n=0, i.e., if no arguments are present, the value returned will be NIL. Only those RETURN statements which are explicitly included in the body of a PROG form can legitimately be used in this manner (for example, a RETURN statement occurring within the definition of a function called during the execution of a PROG will generate an error when the program is compiled.)

If (GO tag) is evaluated, the evaluation process is resumed from the statement labelled with <tag> (in case there is no statement associated with <tag>, i.e., when <tag> is at the end of a PROG body, the PROG routine is simply terminated); <tag> is not evaluated. If the label <tag> does not occur in the body of the PROG form currently being executed, the body of the innermost PROG form properly including the current one is searched, and so forth; if <tag> is found, the execution sequence leaves the current PROG form and the program execution is resumed from the point labelled with <tag>. If the label <tag> does not occur in any PROG form which contains (GO tag), an error is generated. Any statement of the form (GO tag) must be explicitly included in the PROG form containing the destination indicated by <tag>.

GO {Special Form} tag

See the explanation under the entry for PROG above.

Note: <tag> can be an atom of any type including symbolic atoms or “fixnums”. Since the process of searching is effected using the equality criterion EQ, “flonums”, strings, vectors, etc. are generally not appropriate as labels.

RETURN . args

See the explanation under the entry for “PROG” above.

LOOP {Special Form} . body

LOOP is a special form used for simple iteration. The arguments of LOOP are evaluated sequentially from left to right. As long as EXIT is not evoked during these evaluations, this process is interminably repeated. However, if an EXIT form is encountered, the innermost LOOP containing it is terminated and the value of the last argument of this EXIT is returned as the value of that LOOP form.

For example, the top-level LOOP of the system, although actually defined in terms of machine language, could have been defined as follows:

```
(LOOP (PRINT (EVAL (READ))))
```

EXIT . args

See the explanation under the entry for LOOP above. EXIT being an ordinary function, its arguments are evaluated sequentially, from left to right, in the usual manner.

When a LOOP form is to be compiled, the corresponding EXIT forms must be explicitly contained in the LOOP.

DO {Macro} index-part exit-part . body

DO is a control form which facilitates iteration using so-called “index variables”. The first argument <index-part> is a list, the elements of which have the form

```
(var init next)
```

where <var> is a symbol employed as an index variable, <init> is the initial value assigned to <var>, and <next> is a form which is computed after each iteration, whereupon the resulting value is assigned to <var>.

The initial values are computed sequentially, and only after this process is completed are they bound to the corresponding variables; the same applies to subsequent assignments arising from the <next> forms.

The syntax of the second argument <exit-part> is

```
(end-test . exit-forms)
```

After initially binding the index variables, and after each round of <next> value assignments, the form <end-test> is evaluated. If the result is non-NIL, the termination process begins; the forms constituting the list <exit-forms> are evaluated sequentially, from left to right, and the value of the last one (or NIL, if the list <exit-forms> is empty) will be returned as the value of the DO form. The index variables are then unbound, their original values are restored, and the evaluation of the DO form terminates.

Otherwise, if the evaluation of <end-test> yields NIL, execution of <body> begins; <body> is a list of forms which are evaluated sequentially, from left to right, and the results are discarded. When <body> is exhausted, the evaluation process proceeds to the evaluation of the <next> forms.

Any <next> form may be omitted from <index-part> when no assignment of the corresponding variable is required after iteration; in this case, <var> merely serves as an ordinary local variable. Any initiation form <init> may also be omitted; in this case, NIL becomes the initial value of the corresponding <var>.

A DO form, being a macro, is automatically converted into an equivalent combination of LET and LOOP. Thus, to depart from a DO form, the function EXIT can be used in its <body>, <exit-part>, or the <next> forms of its <index-part>. It should be borne in mind that since the <init> forms are evaluated outside the LOOP, the use of EXIT in an <init> form will terminate the evaluation of a still “larger” DO or LOOP form than the one under consideration.

The following examples illustrate the DO function. Printing all the elements of a list X separated by a space can be performed by the following program:

```
(DO ((L X (CDR L)))
     ((ATOM L))
     (PRIN1 (CAR L))
     (PRINC " "))
```

When each element of a vector V is a number, their sum can be computed by the following program:

```
(DO ((I 0 (1+ I))
     (L (VECTOR-LENGTH V))
     (SUM 0 (PLUS SUM (VREF V I))))
    ((= I L) SUM))
```

Note that in this example, the body of DO is empty. This is in fact the case in many applications since the index and exit parts of a DO control form can, in themselves, be quite powerful. Also note that when (VREF V I) is computed, the variable I still retains its previous value, that is the <next> value (1+ I) has not yet been assigned to it. L does not have a <next> part, and is merely a temporary variable which facilitates the computation of <end-test>.

## NON-LOCAL EXITS

CATCH tag . forms

CATCH is a function primarily utilized for non-local exits. (CATCH tag . forms) evaluates the elements of the list <forms> and returns the value of the last form, unless an expression of the form (THROW tag . values) with the same <tag> is encountered during the evaluation of <forms>, in which case the arguments in <values> are evaluated, and CATCH immediately returns the last of the <values> (or NIL when <values> is empty) and performs no further evaluation.

Note: The argument <tag> is evaluated, which is not the case in some other Lisp systems. However, no repeated evaluation is applied to the elements of the list <forms>, which are evaluated just once as the normal arguments of a function. The special action of CATCH occurs during the evaluation of its arguments, rather than during the execution of CATCH itself; the function CATCH, in itself, only returns its last argument (or NIL when there is only one argument <tag>) if the evaluation of its arguments is completed without calling THROW. For example,

```
(CATCH 'ATOMIC
      (MAPCAR L
           (FUNCTION (LAMBDA (X)
                (COND ((ATOM X) (THROW 'ATOMIC X))
                      (T (CAR X)))))))
```

This program returns a list of the “car”s of the elements of the list L if the latter are all non-atomic; otherwise, the first atomic element of L is returned.

THROW tag . values

As described above, THROW is used in conjunction with CATCH for (primarily non-local) exits. THROW conveys the value of the last argument in <values> (or NIL when <values> is empty) back to the closest preceding CATCH in the execution sequence which possesses the same <tag> and has not yet been evoked. Any CATCH forms (or other control forms or functions) which may be nested between the THROW form under consideration and the corresponding CATCH are effectively ignored. See the above description of CATCH for further details.

Note: As in the case of CATCH, both <tag> and forms in <values> are evaluated, unlike the corresponding function THROW in some other Lisp systems.

For example, the following program returns A rather than B.

```
        (CATCH 'A (CATCH 'B (THROW 'A 'A)) 'B)
```

## MAPPING

Mapping is a type of iteration in which a certain function is successively applied to portions of a list or a vector given as an argument. There are several options for the manner in which the portions of the list or the vector are chosen and the results returned by the application of the function are presented.

The table below shows the relations between the six map functions on list structures.

| | Applies function to | |
|---|---|---|
| Returns | Successive sublists | Successive elements |
| Its own first argument | MAP | MAPC |
| List of the function results | MAPLIST | MAPCAR |
| NCONC of the function results | MAPCON | MAPCAN |

MAP list fn

The function <fn> is applied to the successive sublists of <list>, i.e., first <list> itself, its “cdr”, “cddr”, etc. The value returned is its original argument <list> (possibly modified by <fn>). For example,

```
        (MAP '(A B C) (FUNCTION PRIN1))
```

This program prints out “(A B C)(B C)(C)” and returns (A B C).

MAPC list fn

The function <fn> is applied to the successive elements of <list>, i.e., first the “car” of <list>, then its “cadr”, then “caddr”, etc. The value returned is its original argument <list> (possibly modified by <fn>). For example,

```
        (MAPC '(A B C) (FUNCTION PRIN1))
```

This program prints out “ABC” and returns (A B C).

MAPLIST list fn

The function <fn> is applied to the successive sublists of <list>, i.e., first <list> itself, then its “cdr”, then “cddr”, etc. The value returned is a newly created list of the results of these applications. For example,

```
(MAPC '(A B C) (FUNCTION PRIN1))
```

This program prints out "(A B C)(B C)(C)" and returns ((A B C) (B C) (C)).

MAPCAR list fn

The function <fn> is applied to the successive elements of <list>, i.e., first "car" of <list>, then its "cadr", then "caddr", etc. The value returned is a newly created list of the results of these applications. For example,

```
(MAPCAR '(A B C) (FUNCTION PRIN1))
```

This program prints out "ABC" and returns (A B C), which appears to be the same as the original arguments, but, actually, has been newly created.

MAPCON list fn

The function <fn> is applied to the successive sublists of <list>, i.e., first <list> itself, then its "cdr", then "cddr", etc. The value returned is the concatenation of the results of these applications. For example,

```
(MAPCON '(A B C) (FUNCTION NCONS))
```

This program returns (A B C B C C).

MAPCAN list fn

The function <fn> is applied to the successive elements of <list>, i.e., first "car" of <list>, then its "cadr", then "caddr", etc. The value returned is the concatenation of the results of these applications. For example,

```
(MAPCAN '(A B C) (FUNCTION NCONS))
```

This program returns (A B C), which appears to be the same as the original argument, but, actually, has been newly created.

MAPV vector fn

MAPV successively applies <fn> to all the elements of <vector>, in increasing order of indices. The arguments presented to the function <fn> are "reference" objects "pointing" to the elements of <vector>. See the section *Vectors* for further information about "reference". The value returned by MAPV is simply the original argument <vector> (possibly modified by the execution of the function <fn>). For example,

```
(MAPV (VECTOR 5)
      (FUNCTION (LAMBDA (R)
           (SETREF R (READ)))))
```

This will return a vector of five Lisp objects read in consecutively.

MAPVECTOR vector fn

MAPVECTOR also applies <fn> to all the elements of <vector>, in increasing order of indices, however, in this case, the arguments presented to <fn> are the elements themselves, rather than references “pointing” to them (see the description of MAPV). MAPVECTOR returns a new vector, the components of which are the corresponding results of these applications.

# MANIPULATING LIST STRUCTURES

## CONS MANIPULATION

CAR x

CAR returns the “car” of <x>. If <x> is an atom, an error is generated.

CDR x

CDR returns the “cdr” of <x>. If <x> is an atom, an error is generated.

C...R x

All the compositions of CAR and CDR, up to a total of four, are defined as so-called “built-in” functions. The names of these functions begin with “C”, followed by a sequence of “A”s and “D”s corresponding to the indicated composition of functions, and ending with “R”. For example, (CDDAR x) is effectively the same as (CDR (CDR (CAR x)))

CR x

CR returns <x> itself, and is the function in the C...R group for which the total number of “A”s and “D”s is zero. This function is sometimes useful when dealing with mapping functions. For example, (MAPCAR list (FUNCTION CR)) may be used to obtain a top-level copy of <list>.

CONS x y

CONS is a primitive function which creates a new “cons” cell, the “car” and “cdr” of which are <x> and <y>, respectively. For example,

```
(CONS 'A 'B) => (A . B)
(CONS 'A '(B C D)) => (A B C D)
```

NCONS x

(NCONS x) is effectively the same as (CONS x NIL).

XCONS y x

XCONS (an abbreviation of “eXchange CONS”) differs from CONS only in that the order of the arguments is reversed. XCONS is useful when the “cdr” part of the result should be evaluated before the “car” part. For example,

```
(XCONS 'A 'B) => (B . A)
```

COPY x

COPY creates and returns a “copy” of <x>. The atoms constituting the copy are the same as those constituting the original argument <x>, but all the “cons” cells of the copy are newly created.

Note: List structures in which a non-atomic node is indicated by more than one pointer are not copied faithfully; such nodes will be duplicated in the “copy”. “Copying” a cyclic structure in this manner results in an endless computation.

## LIST MANIPULATION

The following section explains some of the basic functions provided for manipulating lists. A list is defined recursively as either the symbol NIL, which represents an empty list, or a “cons” whose “cdr” is a list. However, it should be noted that although their arguments are denoted by the word “list”, the functions described below are applicable whether or not the final atom is NIL.

LAST list

LAST returns the last top-level “cons” of <list>. If <list> is an atom, an error is generated; if the top-level structure of <list> is cyclic, an endless computation occurs. For example,

```
(LAST '(A (B C) D E)) => (E)
```

LENGTH list

LENGTH returns the length of <list>. The length of a list is the number of its top-level elements. As in the case of LAST, if the top-level structure of <list> is cyclic, an endless computation occurs. For example,

```
(LENGTH '(A (B C) D E)) => 4
(LENGTH NIL) => 0
```

FIRST x

==> (CAR x)

SECOND x

==> (CADR x)

THIRD x

==> (CADDR x)

FOURTH x

==> (CADDDR x)

FIFTH x

==> (CAR (CDDDDR x))

SIXTH x

==> (CADR (CDDDDR x))

SEVENTH x

==> (CADDR (CDDDDR x))

NTH n list

NTH returns the <n>th top-level element of <list>, where (CAR list) is counted as the zero element. If <n> is negative or not less than the length of <list>, an error is generated. Note that (NTH 2 x) is actually (THIRD x) rather than (SECOND x). For example,

```
(NTH 2 '(A B C D E)) => C
```

NTHCDR n list

NTHCDR applies CDR to the second argument for <n> times, and returns the result; for <n>=0, the result is simply <list> itself. If <n> is negative or not less than the length of <list>, an error is generated. For example,

```
(NTHCDR 2 '(A B C D E)) => (C D E)
```

LIST . args

LIST constructs and returns a list of its arguments, ordered in the same manner as the arguments themselves. For example,

```
(LIST 1 2 (CAR '(3 5)) (+ 2 2)) => (1 2 3 4)
(LIST) => NIL
```

APPEND . lists

The result of APPEND is essentially a concatenation of its arguments, however, avoiding physical alteration, the arguments are copied (except for the last one; see also the description of NCONC below). The tail of the resulting list is physically identical with that of the last argument. For example,

```
(APPEND '(A B C) '(D E) NIL '(F G H))
         => (A B C D E F G H)
(APPEND) => NIL
```

Note: When several lists are to be APPENDed, and the order of the resulting list is not essential, the longest argument should be entered last since it is not copied; this reduces both computing time and required memory space.

REVERSE list

REVERSE creates a new list, the top-level elements of which are the same as those of <list> but arranged in reverse order. REVERSE, unlike NREVERSE (see below), does not modify its argument. For example,

```
        (REVERSE '(A (B C) D)) => (D (B C) A)
```

NCONC . lists

NCONC returns a list which is the concatenation of the arguments. The arguments (except the last one) are physically altered in the manner of RPLACD rather than copied (see also the description of APPEND above.) For example,

```
        (SETQ X '(A B C))
        (SETQ Y '(D E F))
        (NCONC X Y) => (A B C D E F)
        X => (A B C D E F)
```

Note that the value of X itself has been altered, since the "cdr" of its last "cons" has been replaced by the value of Y.

NREVERSE list

NREVERSE reverses its argument <list>, which is altered in the RPLACD manner throughout the list (see also the description of REVERSE). For example,

```
        (SETQ X '(A B C))
        (NREVERSE X) => (C B A)
        X => (A)
```

Note that the value of X itself has been altered, since the original list has been modified in RPLACD fashion.

PUSH {Special Form} item var

(PUSH item var) has the same effect and value as

```
        (SETQ var (CONS item var))
```

but is more readable. <var> must be a bound variable. PUSH is useful, along with POP (see below), in maintaining a list in the manner of a push-down stack.

POP {Special Form} var

(POP var) has the same effect and value as

```
        (PROG1 (CAR var) (SETQ var (CDR var)))
```

but is more readable. <var> must be a symbol which is bound to a non-atomic value prior to the execution of POP. POP is useful, along with PUSH (see above), in maintaining a list in the manner of a push-down stack.

## ALTERATION OF LIST STRUCTURES

The functions RPLACA and RPLACD serve to alter existing list structures; that is, to change the "cars" and "cdrs" of existing "cons" cells.

Since structure is physically altered rather than copied, caution should be exercised when using these functions, as unexpected side-effects may occur if portions of the affected list structures are common to several Lisp objects. The functions NCONC and NREVERSE also alter list structure, however, they are not normally used to obtain such side-effects, rather, the concomitant list-structure modification is effected purely for the sake of efficiency and corresponding non-destructive functions are also available.

RPLACA x y

RPLACA replaces the "car" of <x> by <y> and returns (modified) <x>. <x> must be a "cons", while <y> may be any Lisp object. For example,

```
(SETQ X '(A B C))
(RPLACA X 'N) => (N B C)
X => (N B C)
```

RPLACD x y

RPLACD replaces the "cdr" of <x> by <y> and returns (modified) <x>. <x> must be a "cons", while <y> may be any Lisp object. For example,

```
(SETQ X '(A B C))
(RPLACD X 'C) => (A . C)
X => (A . C)
```

SUBST x y z

(SUBST x y z) substitutes <x> for all occurrences of <y> in <z> (using EQ for testing equality) and returns the modified copy of <z>. The original <z> is not altered, as SUBST recursively copies all the "cons" cells of <z>, replacing by <x> all elements which are EQ to <y>. For example,

```
(SUBST 'A 'B '(A B (C B))) => (A A (C A))
```

Note: List structures in which a non-atomic node is designated by more than one pointer are not copied faithfully; such nodes will be duplicated in the "copy". Applying SUBST to a cyclic structure results in an endless computation.

## TABLES

The system provides several functions which simplify the maintenance of several varieties of tabular data structures assembled from "cons" cells.

The simplest of these structures is just an ordinary list of items, which represents an ordered set.

An association list is a list in which the elements are "cons" cells. The "car" of each such "cons" is called a "key" and the "cdr" represents an associated datum.

Although these simple data structures are convenient for small data bases, their form is such that search time increases linearly with the size of the data base, and consequently they are inefficient when handling large amounts of data. Large-scale data bases are best maintained using vectors and hashing functions (see the section *Hashing* for details).

MEMQ item list

(MEMQ item list) returns NIL if <item> is not identical (with respect to the function EQ) with one of the elements of <list>, otherwise it returns the portion of <list> beginning with the first occurrence of <item>. The procedure searches <list> on the top-level only. Since MEMQ returns NIL if <item> is not found, and a non-NIL object if it is found, MEMQ can be used as a predicate. For example,

```
(SETQ X '(A B C D E))
(MEMQ 'C X) => (C D E)
(MEMQ 'FOO X) => NIL
```

MEMBER item list

MEMBER functions in the same manner as MEMQ, except that EQUAL, rather than EQ, is used for comparison.

MEM predicate item list

MEM functions in the same manner as MEMQ, except that it takes an additional argument <predicate>. The latter may be any predicate taking two arguments. (MEM (FUNCTION EQ) a b) is effectively identical with (MEMQ a b) and (MEM (FUNCTION EQUAL) a b) with (MEMBER a b). For example,

```
(MEM (FUNCTION (LAMBDA (X Y) (0= (+ X Y))))
      13
      '(1 3 -4 -13 7 -6))
     => (-13 7 -6)
```

DELQ item list (n)

When the optional argument <n> is absent, DELQ returns <list> with all top-level occurrences of <item> deleted; EQ is used for comparison. The argument <list> is actually altered in the RPLACD manner when occurrences of <item> are excised, except that any initial segment of <list>, all the elements of which are EQ to <item>, is not altered in this manner (see the example below). If <n> is present, it must be a “fixnum” and only the first <n> top-level occurrences of <item> are deleted. <n> may be zero, in which case, <list> itself is returned without any alteration. For example,

```
(SETQ X '(A B A B))
(DELQ X 'B) => (A A)
X => (A A)
```

Note: DELQ should be used for value, not for effect. Thus, the two pairs of operations

```
(SETQ Y '(A B A B))
(SETQ Y (DELQ 'A Y))
```

and

```
(SETQ Y '(A B A B))
(DELQ 'A Y)
```

result in different values of Y. The value returned by DELQ is (B B) in both cases. However, Y is given the value (B B) in the former case and (A B B) in the latter.

REMQ item list (n)

REMQ yields the same result as DELQ, except that <list> itself is not altered; what is returned is a copy of the original argument <list> with the first <m> top-level occurrences of <item> removed, where <m> is the minimum of <n> and the number of top-level occurrences of <item> in <list>.

EVERY list predicate

EVERY applies <predicate>, a predicate function of one argument, to the top-level elements of <list> sequentially, from left to right. If <predicate> returns non-NIL for every element, EVERY returns T. If any of these applications yields NIL, EVERY returns NIL immediately, and no further applications are executed.

SOME list predicate

SOME applies <predicate>, a predicate function of one argument, to the top-level elements of <list> sequentially, from left to right. If <predicate> returns non-NIL for some element, SOME immediately returns the portion of <list> beginning with the element which yielded non-NIL, and no further applications are executed. If all the applications yield NIL, SOME returns NIL.

ASSQ item alist

ASSQ searches for and returns the first element in the association list <alist>, the “car” of which is EQ to <item>, if such an element exists; otherwise, the value NIL is returned. The association list may be updated by applying RPLACD to the result of ASSQ, if the latter is not NIL. For example,

```
(ASSQ 'C '((A B) (C D) (E F))) => (C D)
```

ASSOC item alist

ASSOC functions in the same manner as ASSQ, except that EQUAL instead of EQ is used for comparison.

ASS predicate item alist

ASS functions in the same manner as ASSQ, except that it takes an additional argument <predicate>, a predicate taking two arguments, which is used for comparison. In the special case where <predicate> is EQ, this function effectively reduces to ASSQ.

## SORTING

SORT table predicate

> The list <table> is arranged in increasing order, using the ordering relation corresponding to <predicate>, and the resulting ordered list is returned. <predicate> should be a function of two arguments, which returns non-NIL if, and only if, the first argument is strictly less than the second in the sense of total ordering relation.

## HASHING

Some hashing scheme is desirable in order to reduce the computing time required for data retrieval in large-scale data bases. The search time required for an item remains constant using hashing, as long as the hash table is large enough, compared with the number of its entries.

The present system provides a standard hashing function for Lisp objects to facilitate the maintainance of hashed data bases.

HASH x

> HASH computes hash value for <x> and returns it as an integer number "fixnum". The result may be positive, negative, or zero. Its guaranteed properties are:
>
> (1) Objects which are EQUAL are hashed to the equal value.
> (2) A "fixnum" is hashed to itself.
> (3) A string is hashed to non-negative value.
> (4) A symbol is hashed to the same value as its print-name.

# SYMBOLS

Symbolic atoms such as X or CONS are called “symbols” in this system. A symbol is associated with four Lisp objects:

(1) the “binding” is the value of the symbol when it is used as a variable;

(2) the “definition” is the functional definition of the symbol when it is used as the name of a function or a macro;

(3) the “property list” is used to retain various Lisp objects associated with the symbol;

(4) the “print name” is used for input and output operations.

## THE VALUE

A symbol can be associated with its “value”, which may be a Lisp object of any type, and is returned as the result of evaluating the symbol. The symbol may be in “unbound” state, in which case the symbol has no value at all; when an “unbound” symbol is evaluated, an error is generated. Newly created symbols (by INTERN, GENSYM, etc.) are initially in the “unbound” state. A symbol is called a “variable” when the primary concern is its value.

The value of a variable can be changed either by “lambda-binding” or by “assignment”; when a symbol is “lambda-bound”, its previous value is saved and will be restored later, whereas “assignment” discards the previous value. “Lambda-binding” is sometimes called simply “binding” in this manual.

The symbols NIL and T are always bound to themselves; they may not be assigned nor lambda-bound. (The error of changing the value of T or NIL is not detected!)

SET variable new-value

“Assignment” to <variable> can be effected by the function SET. The value of <variable> is changed to <new-value> which may be any Lisp object. The previous value of <variable>, if any, is discarded. SET returns the newly assigned value <new-value>.

SETQ {Special Form} . args

(SETQ x y) is effectively the same as (SET 'x y).

Additional feature of SETQ is concurrent assignment of variables without explicit temporary variables. A SETQ form such as

```
(SETQ var1 form1 var2 form2 ...)
```

is used for this purpose. <form1>, <form2>, ... are all evaluated first, sequentially, in this order. Then their resulting values are assigned to <var1>, <var2>, ....

For example, the values of two variables X and Y can be exchanged by

```
        (SETQ X Y Y X)
```

BOUNDP variable

BOUNDP returns T if <variable> is bound to some value; if it is unbound, NIL is returned.

MAKE–UNBOUND variable

MAKE–UNBOUND makes <variable> unbound. The current value of <variable>, if any, is discarded. MAKE–UNBOUND returns the symbol <variable> as its value.

## THE DEFINITION

A symbol can be associated with its “functional definition”, or “definition”, for short. When a function is called via its name, that is, when the first argument of FUNCALL or APPLY is a symbol, or a symbol appears as the “car” of a form to be evaluated, the “definition” of that symbol is called as a function. When a symbol is not defined as a function or a macro, the symbol is said to be “undefined”; an error is generated when an undefined symbol is used as a function.

DEFUN {Macro} name lambda-list . body

DEFUN can be used for defining functions. <name> should be a symbol. A list

```
        (LAMBDA lambda-list . body)
```

will be the new definition of <name>. The previous definition of <name>, if any, is discarded. DEFUN returns <name> as its value.

MACRO {Macro} name lambda-list . body

MACRO can be used for defining macros. <name> should be a symbol. A list

```
        (MACRO LAMBDA (arg) . body)
```

will be the new definition of <name>. The previous definition of <name>, if any, is discarded. MACRO returns <name> as its value.

Note: Macros can more elegantly be defined using DEFMACRO. See the section *Macros* for details.

GETD sym

GETD returns the definition of a symbol <sym>. If <sym> is undefined, an error is generated.

PUTD sym def

PUTD makes the definition of <sym> be <def>. <sym> must be a symbol while <def> may be any Lisp object. It returns <sym> as its value.

DEFINEDP sym

> DEFINEDP returns T if <sym> is defined as a function or a macro; if it is undefined, NIL is returned.
>
> Note: DEFINEDP returns NIL for special form indicators such as COND, since they are not defined as an ordinary function or a macro. Use the function SPECIALP (see below) to discriminate special form indicators.

SPECIALP sym

> SPECIALP returns T if <sym> is a special form indicator (such as COND or PROG); otherwise, it returns NIL.

MAKE–UNDEFINED sym

> MAKE–UNDEFINED makes the symbol <sym> undefined. Current definition of <sym>, if any, is discarded. It returns <sym> as its value.

## THE PROPERTY LIST

Every symbol is associated with its “property list”, which is a list used for associating certain Lisp objects with symbols. A property list has an even number of elements; each pair of elements constitutes a property. The first of the pair is called the “indicator” or the “name” of the property, and the second is a Lisp object called the “value” of the property.

For example, a property list which have the form

```
    (JAPAN TOKYO ENGLAND LONDON FRANCE PARIS)
```

indicates that there are three properties named JAPAN, ENGLAND, and FRANCE, and their values are TOKYO, LONDON, and PARIS, respectively.

When a symbol is created, its property list is initially NIL.

Note: Print-names, bindings and functional definitions are often implemented as properties of symbols in various Lisp systems, however, they are not implemented as usual “properties” in this system.

GET sym name

> GET searches for a property of <sym> named <name>. If it finds such a property, it returns the value of that property; otherwise, it returns NIL.
>
> Note: If the value of a property is NIL, it is impossible to distinguish whether the property exists or not, only from the result of GET.

PUTPROP sym value name

> If the symbol <sym> has no property with its name being <name>, PUTPROP adds a new property named <name> with the value <value>; otherwise, the value of the existing

property is updated to <value>. PUTPROP returns <value> as its resulting value.

DEFPROP {Macro} sym name value

(DEFPROP x y z) ==> (PUTPROP 'x 'y 'z)

REMPROP sym name

REMPROP removes the property of <sym> with its name being <name>. If <sym> has no such property, it merely does nothing. REMPROP returns NIL as its value.

LIST sym

PLIST returns the property list of <sym>.

SETPLIST sym property-list

SETPLIST sets the property list of <sym> to <property-list>. It returns <property-list> as its value.

## THE PRINT NAME

Every symbol has an associated string called the “print-name”, or “pname” for short. This string is used as the printed representation of the symbol in input and output operations.

Though print-names are normal character string objects (see the section *Strings* for more information about strings), modifying them (by SSET, etc.) requires certain care, since they are used to “hash” symbols into the Lisp name table, “obvector” (see the section *Input and Output* for details).

PNAME sym

PNAME returns the print-name of the symbol <sym>.

## CREATION OF SYMBOLS

SYMBOL pname

SYMBOL creates and returns a new symbol with its print-name being <pname>.

GENSYM (prefix) (begin)

GENSYM generates a new print-name, and creates a new “uninterned” symbol with that print-name (see the section *Input and Output* for “interning”).

The generated print-name is prefixed by a string, which is initially “G” but can be changed by supplying GENSYM a string argument <prefix>. The prefix string is followed by a 4-digit decimal representation of an integer number. This number is incremented by one every time GENSYM is called and only the least significant 4 digits are used. This number can also be initiated by giving a “fixnum” to GENSYM as its second argument <begin>. For example,

```
(GENSYM) => G0034
(GENSYM "GEN") => GEN0035
(GENSYM "ABC" 15) => ABC0015
(GENSYM) => ABC0016
```

Note: Print-names of symbols generated by GENSYM are primarily for ease of their inspection in printed representations. After ten thousand GENSYM calls, the print-name of the generated symbol will be the same as the first one, but they are not the same symbol.

See also the section *Input and Output* for INTERN which may create a symbol with given print-name.

# NUMBERS

There are two types of numbers in this system, namely “fixnums” and “flonums”.

The “fixnums” have a signed integer value of 24 bits. No overflow checking is made on arithmetical operations on “fixnums”. All the results are treated modulo 2 ** 24.

The “flonums” have a 64-bit floating point value with the accuracy of about 15 decimal digits. Overflows are checked on arithmetical operations on “flonums”, but underflows are not.

“Fixnums” are denoted using conventional decimal notation (e.g., 15) and “flonums” using decimal notation with a decimal point (e.g., 15.0); “flonums” can also have an “exponential part” indicated by the character “↑” (e.g., 1.5↑1).

Functions described in this section expect numbers of appropriate types as their arguments; if an argument of an illegal type is supplied, an error is generated.

Functions on numbers can be grouped into three categories by the type of the numbers they accept: Functions the name of which includes alphabetic characters (e.g., PLUS) can be applied to both “fixnums” and “flonums”. If all the arguments are “fixnums”, the result will be a “fixnum”; otherwise, the result will be a “flonum”. Functions consisting only of non-alphabetic characters are special purpose functions. If their names end with the character “$” (e.g., +$), they are for “flonums” only; otherwise (e.g., +), for “fixnums” only. These rules apply to all the functions described in this section except explicitly stated otherwise.

Special-purpose arithmetic functions can be computed more efficiently than general-purpose ones, especially when the functions using them are compiled.

## NUMERIC PREDICATES

ZEROP x
0= x
0=$ x

ZEROP, 0=, and 0=$ return T if <x> is zero (of proper type); otherwise, they return NIL.

PLUSP x
0< x
0<$ x

PLUSP, 0<, and 0<$ return T if <x> is a positive number (of proper type); otherwise, they return NIL.

MINUSP x
0> x
0>$ x

MINUSP, 0>, and 0>$ return T if <x> is a negative number (of proper type); otherwise, they return NIL.

ODDP x

ODDP returns T if <x> is odd; otherwise, it returns NIL. <x> must be a “fixnum”.

= x y
=$ x y

= and =$ return T if <x> and <y> are equal numbers (of proper type); otherwise, they return NIL.

Note: Equality of numbers can also be tested using the function EQUAL; equality of “fixnums” can also be tested using EQ.

# x y
<> x y
#$ x y
<>$ x y

#, <>, #$, and <>$ return T if <x> and <y> are not equal numbers; otherwise, they return NIL.

LESSP arg . args
< arg . args
<$ arg . args

LESSP, <, and <$ returns T if every argument (except the last one) is strictly less than the next argument; otherwise, it returns NIL.

GREATERP arg . args
> arg . args
>$ arg . args

GREATERP, >, and >$ returns T if every argument (except the last one) is strictly greater than the next argument; otherwise, it returns NIL.

<= arg . args
<=$ arg . args

<= and <=$ returns T if every argument (except the last one) is less than or equal to the next argument; otherwise, it returns NIL.

>= arg . args
>=$ arg . args

>= and >=$ returns T if every argument (except the last one) is greater than or equal to the next argument; otherwise, it returns NIL.

## CONVERSION FUNCTIONS

FIX x

FIX converts a “flonum” <x> into a “fixnum” and returns it; rounding is used for the conversion.

FLOAT x

FLOAT converts a “fixnum” <x> into a “flonum” and returns it.

## ARITHMETICS

PLUS . args
+ . args
+$ . args

PLUS, +, and +$ return the sum of its arguments. With no argument, PLUS and + return 0, and +$ returns 0.0.

MINUS x

MINUS returns the negative of <x>.

DIFFERENCE arg . args

DIFFERENCE returns its first argument minus all the rest of its arguments.

– arg . args
–$ arg . args

With only one argument, – and –$ behave the same as MINUS; they return the negative of its argument. With more than one argument, – and –$ are effectively the same as DIFFERENCE; they return their first argument minus all of the rest of the arguments.

TIMES . args
* . args
*$ . args

TIMES, *, and *$ return the product of its arguments. With no argument, TIMES and * return 1, and *$ returns 1.0.

QUOTIENT arg . args
// arg . args
//$ arg . args

QUOTIENT, //, and //$ return the first argument divided by all of the rest of its arguments. For //, the division performed is integer division with truncation; for //$, floating-point division; for QUOTIENT, the type of the division performed depends on the type of the arguments.

// is written here as “//” rather than “/” since “/” is the quoting character in Lisp syntax and must be doubled.

ADD1 x

(ADD1 x) ==> (PLUS x 1)

1+ x

(1+ x) ==> (+ x 1)

1+$ x

(1+$ x) ==> (+$ x 1.0)

SUB1 x

(SUB1 x) ==> (DIFFERENCE x 1)

1– x

(1– x) ==> (– x 1)

1–$ x

(1–$ x) ==> (–$ x 1.0)

INCR {Macro} var amount

(INCR var amount) ==> (SETQ var (+ var amount))

DECR {Macro} var amount

(DECR var amount) ==> (SETQ var (– var amount))

REMAINDER x y
\ x y
\$ x y

REMAINDER, \, and \$ return the remainder of <x> divided by <y>. The sign of the result is the same with <x> (if not zero).

MAX arg . args

MAX returns the largest of its arguments.

MIN arg . args

MIN returns the smallest of its arguments.

ABS x

ABS returns |<x>|, the absolute value of the number <x>.

EXPT x y
↑ x y
↑$ x y

EXPT, ↑, and ↑$ return the <y>th power of <x>. <y> must be a “fixnum”. When <x> is a “fixnum” and <y> is non-negative, the result will be a “fixnum”; in all the other cases, the result will be a “flonum”.

SIN x

SIN computes and returns sin(x). <x> can be either a “flonum” or a “fixnum”, and the result is a “flonum”.

COS x

COS computes and returns cos(x). <x> can be either a “flonum” or a “fixnum”, and the result is a “flonum”.

TAN x

TAN computes and returns tan(x). <x> can be either a “flonum” or a “fixnum”, and the result is a “flonum”.

ARCSIN x

ARCSIN computes and returns arc sin(x). <x> can be either a “flonum” or a “fixnum”, and the result is a “flonum”.

ARCCOS x

ARCCOS computes and returns arc cos(x). <x> can be either a “flonum” or a “fixnum”, and the result is a “flonum”.

ARCTAN x

ARCTAN computes and returns arc tan(x). <x> can be either a “flonum” or a “fixnum”, and the result is a “flonum”.

SQRT x

SQRT computes and returns the square root of <x>. <x> can be either a “flonum” or a “fixnum”, and the result is a “flonum”.

LOG x

LOG computes and returns the natural logarithm of <x>. <x> can be either a “flonum” or a “fixnum”, and the result is a “flonum”.

LOG10 x

LOG10 computes and returns the ordinary logarithm of <x>. <x> can be either a “flonum” or a “fixnum”, and the result is a “flonum”.

EXP x

EXP computes and returns a “flonum”, the natural logarithm of which is <x>.

## LOGICAL OPERATIONS ON NUMBERS

The following functions treat “fixnums” as bit sequences of 24-bit long. If a non-fixnum argument is supplied, an error is generated.

LOGOR . args

LOGOR returns bitwise logical “or” of the arguments. When no arguments are supplied, 0 is returned.

LOGAND . args

LOGAND returns bitwise logical “and” of the arguments. When no arguments are supplied, −1 is returned.

LOGXOR . args

LOGXOR returns bitwise logical “xor” of the arguments. When no arguments are supplied, 0 is returned.

LOGSHIFT x y

LOGSHIFT returns <x> logically shifted <y> bits. If <y> is positive, <x> is shifted left; if <y> is negative, <x> is shifted right. Absolute value of <y> should be less than 24.

# STRINGS

A string is a Lisp object consisting of a sequence of zero or more characters. They are primarily used for manipulating texts. Print-names of symbols are also represented using strings.

Characters of a string can be independently referenced and updated using SREF and SSET, respectively. The subscript origin for strings is zero. If an subscript value specified is not appropriate, i.e., if it is negative or greater than or equal to the length of the corresponding string, an error is signalled.

Characters are a “fixnum” which resides between 0 and 255, i.e., representable in one byte (8 bits). They are usually treated as EBCDIK character codes in input and output operations.

Strings can also be used as vectors of small non-negative integer “fixnums” ranging from 0 through 255. This kind of usage may save a considerable memory space, compared with the use of normal vectors which requires 4 bytes for each component.

## CHARACTERS

Characters are a “fixnum” which resides between 0 and 255. They are treated as EBCDIK codes in input and output operations.

CHARACTER x

Some of the functions manipulating strings require their arguments to be a character. Though most of the functions introduced in this section automatically coerce strings or symbols to characters, there are certain cases in which explicit conversion is required.

CHARACTER coerces <x> to a single character, represented as a “fixnum”. If <x> is a character, i.e., a “fixnum” which resides between 0 and 255, <x> itself is returned. If <x> is a non-null string, its first character is returned. If <x> is a symbol, the first character of its print-name is returned. Otherwise, an error is generated.

## STRING MANIPULATION

Note that the subscript origin for strings is zero.

STRING x

Functions manipulating strings require string arguments. Though most of the functions introduced in this section automatically coerce symbols to strings, there are certain cases in which explicit conversion is required.

STRING coerces <x> into a string. If <x> is a string, <x> itself is returned. If <x> is a symbol, its pname is returned. If <x> is a character, a one-character string containing <x> is returned. Otherwise, an error is generated.

MAKE–STRING length (char)

MAKE–STRING allocates and returns a new string of the length given by <length>. If the optional argument <char>, which must be a character, is supplied, all the characters of the allocated string will be initiated to <char>; otherwise, to the “fixnum” 0 (not the character code for “0”).

STRING–LENGTH string

STRING–LENGTH returns the number of characters in <string>, which is one more than the largest subscript value for <string>.

STRING–EQUAL string1 string2

STRING–EQUAL compares two strings and returns T if two strings have the same length and all the corresponding characters are the same; otherwise, it returns NIL.

Comparison of equality of two strings can also be effected by the function EQUAL though, STRING–EQUAL is more specific and, therefore, more efficient.

STRING–LESSP string1 string2

STRING–LESSP compares two strings using dictionary order. The result is T if <string1> is the lesser, and NIL if they are equal or <string2> is the lesser. For example,

```
(STRING-LESSP "ABC" "ABD") => T
(STRING-LESSP "ABC" "AB")  => NIL
```

SUBSTRING string (start) (end)

SUBSTRING extracts a substring of <string>, starting at the character specified by <start> up to but not including the character specified by <end>. Thus, the length of the string returned will be <end> minus <start>. The default value for <end> is the length of <string>, and that of <start> is zero. For example,

```
(SUBSTRING "ABCDE" 1 3) => "BC"
(SUBSTRING "ABCDE" 1)   => "BCDE"
(SUBSTRING "ABCDE")     => "ABCDE"
```

Note: Even if both <start> and <end> are omitted, SUBSTRING makes a new copy of <string> and returns it.

STRING–APPEND . strings

Any number of strings are copied and concatenated into a single string. If no arguments are supplied, STRING–APPEND returns a null string "".

STRING–REVERSE string

Returns a copy of <string> with the order of characters reversed. The original string is not physically altered (see the description of STRING–NREVERSE below).

STRING–NREVERSE string

Returns <string> with the order of characters reversed. The reversing is made on the argument <string> directly, physically altering the order of characters in <string> (see the description of STRING–REVERSE above).

STRING–SEARCH–CHAR char string (from)

STRING–SEARCH–CHAR searches for <char> through <string> starting at the index <from>. It returns the index of the first appearance of <char>, or NIL if none is found. <char> may be a character or a list of characters; in the latter case, the subscript of the first occurrence of one of the listed characters is returned. The default value for <from> is zero. For example,

```
(STRING-SEARCH-CHAR "B" "ABCDE") => 1
```

STRING–SEARCH–NOT–CHAR char string (from)

STRING–SEARCH–NOT–CHAR is the same as STRING–SEARCH–CHAR except that it searches for the occurrence of character which is “not” <char>, or, when <char> is a list, “not” a member of <char>. For example,

```
(STRING-SEARCH-NOT-CHAR "0" "007") => 2
```

STRING–SEARCH key string (from)

STRING–SEARCH searches for the string <key> in the string <string>. The search begins at subscript <from>, the default value of which is zero. The value returned is the subscript of the first character of the first instance of <key>, or NIL if none is found.

TRANSLATE string table

TRANSLATE converts characters in <string> using <table> as the conversion table. <table> must be a string of 256 characters. Its subscript-n character substitutes the character whose code is n. The argument <string> is physically altered. TRANSLATE returns the (modified) <string>.

LOWER–CASE {Variable}
UPPER–CASE {Variable}

Values of LOWER–CASE and UPPER–CASE are standard conversion tables for converting uppercase characters to lowercase ones and the reverse, respectively. These tables are also used by the Lisp reader and the printer (see the section *Input and Output* for details).

STRING–AMEND string1 string2 (from)

STRING–AMEND moves characters in <string2> into <string1>, physically altering characters in <string1>. All the characters in <string2> are moved to the portion of <string1>, beginning with the specified subscript value <from>. The default value of <from> is zero.

STRING–AMEND–AND string1 string2 (from)
STRING–AMEND–OR string1 string2 (from)
STRING–AMEND–XOR string1 string2 (from)

STRING–AMEND–AND, –OR and –XOR are the same as STRING– AMEND except that characters in <string2> are not simply moved into <string1>, rather, logical “and”, “or” or “xor” of characters in <string2> and corresponding characters in <string1> are moved to a portion of <string1> beginning with the specified subscript value <from>. The default value of <from> is zero.

## MANIPULATION OF CHARACTERS IN STRINGS

Characters of strings can be independently manipulated by the following functions. Note that the subscript origin of strings is zero.

GETCHAR string index

GETCHAR returns the <index>th character of <string> as an interned one-character symbol. For example,

```
(GETCH "ABC" 2) => C
```

SREF string index

SREF returns the <index>th character of <string> as a character, i.e., a “fixnum”.

SSET string index character

SSET sets the <index>th character of <string> to <character>, and returns <character>.

## CONVERTING STRINGS AND NUMBERS

Consecutive characters of a string may be considered as a binary representation of an integer number. Following functions are for conversions between such character sequences and “fixnums”.

CUTOUT string pos length

CUTOUT converts a character sequence beginning at the <pos>th character of <string> with length <length> into a “fixnum”. <length> should be positive and not greater than 3. If <length> is 1 or 2, upper bytes of the result will be padded with zero.

SPREAD value length

SPREAD converts a “fixnum” <value> into a string which contains the binary representation of <value>. The resulting string has the length <length>, which should be positive and not greater than 3. If <value> cannot be represented in <length> bytes, only lower bytes are converted and overflowed upper bytes are ignored.

## BIT STRING MANIPULATION

A string can also be regarded as a sequence of binary digits (bits). Thus, an array of logical values can be represented by a string, in which case, one character can hold eight distinct logical values. Using this representation, the memory space required for a large-scale bit table will be eight times smaller than when each character of a string is used to represent one logical value, or thirty-two times than when each vector element is used. To facilitate such a representation of bit tables, the following functions are provided by the system. Compact representation of bit tables using the following functions may save considerable memory space, however, computing speed will be somewhat slowed down. Note that functions such as STRING–AMEND–AND, –OR, and –XOR may also be useful for logical operation on bit tables.

BREF string index

BREF returns T if <index>th bit of <string> is set; otherwise, it returns NIL. <index> must be non-negative and smaller than eight times the length of <string>.

BSET string index value

If <value> is non-NIL, the <index>th bit of <string> is set; otherwise, it is reset. <index> must be non-negative and smaller than eight times the length of <string>. BSET returns <value> as its value.

# VECTORS

Vectors are Lisp objects that consist of elements, each of which is a Lisp object again. The individual elements are selected by numerical subscripts starting with zero. An error is generated if a subscript value specified is not appropriate, i.e., if it is negative or greater than or equal to the number of the elements.

Because elements of a vector can be accessed in constant time, it is advantageous to use vectors when a large amount of data is to be manipulated, instead of a list structure consisting of binary "cons" cells. The disadvantage of using vectors, compared with lists, is that the size must be known before used.

Vectors can be arbitrarily allocated and discarded like "cons" cells; they are independent objects in their own right, rather than being attributes of symbols as in some other Lisp systems. However, it is usually convenient to lambda-bind or assign a vector to a symbol and to use the symbol as its name, since vectors cannot be directly identified by the Lisp reader.

Multi-dimensional arrays can be represented by vectors of vectors; vectors the elements of which are vectors again.

## VECTOR MANIPULATION

VECTOR size (filler)

VECTOR allocates and returns a vector with its size being <size>; its subscript ranges from 0 to <size> minus 1. If the optional argument <filler> is not supplied, all the elements of the allocated vector are initiated to NIL. Otherwise, if <filler> is supplied, the allocated vector will be initiated using <filler> in the same way as the function FILL–VECTOR (see the description of FILL–VECTOR below).

VECTOR–LENGTH vector

VECTOR–LENGTH returns the number of elements of <vector>.

VREF vector subscript

VREF returns the <subscript>th element of <vector>.

VSET vector subscript value

VSET sets <value> into the <subscript>th element of <vector>. VSET returns <value> as its value.

FILL–VECTOR vector filler

FILL–VECTOR fills <vector> with specified data and returns (modified) <vector>.

When <filler> is an atom and not a vector, all the elements of <vector> become <filler>.

When <filler> is a list with one or more elements, <vector> is filled with the elements of that list. The subscript 0 element of <vector> is assigned the “car” of the list, subscript 1, the cadr, and so on. If the list is shorter than <vector>, remaining elements of <vector> are not affected. If the list is longer, remaining elements of the list are merely ignored.

When <filler> is a vector, <vector> is filled with corresponding elements of the filler vector. If the filler vector is shorter, remaining elements of <vector> are not affected. If the filler vector is longer, remaining elements of the filler vector are merely ignored.

The following examples illustrate the use of the FILL–VECTOR function. When the value of V is a vector with, for example, 10 elements,

```
(FILL-VECTOR V NIL)
```

fills the vector with NILs.

```
(FILL-VECTOR V '(0 1 2 3 4))
```

sets first 5 elements of the vector with 0, 1, 2, 3, and 4, respectively. The remaining 5 elements are not affected. If the value of W is another vector with the same size,

```
(FILL-VECTOR V W)
```

copies the contents of W into V.

## REFERENCES

It is often necessary to pass a vector and its subscript as a pair to functions. It would be more convenient if the pair could be treated just as a variable. UTILISP provides “reference” objects for this purpose.

A “reference” is a pointer to an element of a vector. The pointed element can be accessed by DEREF and updated by SETREF. DEREF and SETREF can also be applied to variables, i.e., symbols. It is recommended that DEREF and SETREF should be used in functions which utilize the “call-by-reference” parameter, instead of EVAL and SET.

REFERENCE vector subscript

REFERENCE makes and returns a “reference” pointing to the <subscript>th element of <vector>.

DEREF reference

DEREF returns the value of <reference>; if it is a symbol, the value of the symbol; if it is a “reference”, the element of the vector it points to.

SETREF reference value

SETREF sets <value> to <reference>; if it is a symbol, its value is set; if it is a reference pointer, the pointed element of a vector is set. SETREF returns <value> as its value.

REFERRED–VECTOR reference

> REFERRED–VECTOR returns the vector an element which is pointed to by <reference>.
>
> Note: Computation of this function requires time proportional to the subscript of the element pointed to by <reference>.

REFERRED–INDEX reference

> REFERRED–INDEX returns the subscript of the vector element pointed to by <reference>.
>
> Note: Computation of this function requires time proportional to the subscript of the element pointed to by <reference>.

See the description of MAPVECTOR and MAPV in the section *Mapping* to perform certain computation on all the elements of a vector.

# MACROS

## EVALUATION OF MACROS

When a “cons” cell with its “car” being a symbol is evaluated, the evaluator inspects the definition of that symbol. If the definition is a “cons” cell, and its “car” is the symbol MACRO, that definition is called a “macro”. The “cdr” of the definition is treated as a function of one argument. The evaluator applies that function to the “cdr” of the original form. The result of this application is evaluated again by the evaluator, and the value returned by this re-evaluation is finally returned as the result of the evaluation of the original form.

For example, suppose the definition of NCONS is

```
(MACRO LAMBDA (X) (LIST 'CONS (CAR X) NIL))
```

This is a macro; it is a “cons”, the “car” of which is the symbol MACRO. The evaluation process of a form (NCONS 'FOO) is as follows:

> The evaluator recognizes that the form to be evaluated is a “cons” cell, the “car” of which is a symbol, i.e., NCONS; the definition of the symbol NCONS is examined and the “car” of the definition is found to be the symbol MACRO. Then the evaluator takes the “cdr” of the definition, which is a LAMBDA-expression, and applies it to the “cdr” of the original form, i.e., the list ('FOO). X is bound to ('FOO) and the result of the application will be (CONS 'FOO NIL).
>
> The evaluator then evaluates this new form in place of the original one. (CONS 'FOO NIL) is evaluated to (FOO) and so the result of (NCONS 'FOO) is, finally, (FOO).

Macros can be expanded recursively; the expanded form of a macro form can be another macro form, in which case the expanded form is expanded again, until it becomes a non-macro form.

Macros can be used for a variety of purposes. For example, “custom-made” control structures can be easily implemented as macros.

For example, a WHILE−DO construct such as

```
(WHILE-DO condition . body)
```

can be defined as a macro using a MACRO special form such as

```
(MACRO WHILE-DO (X)
       (NCONC (LIST 'LOOP
              (LIST 'AND (CAR X) '(EXIT)))
       (CDR X)))
```

which expands the original form into

```
(LOOP (AND condition (EXIT)) . body)
```

Using macros may result in a considerable time and space overhead while the program is executed interpretively. However, once compiled, programs using macros can be executed as efficiently as those without macros, since the compiler expands macro calls prior to the compilation. Thus, using macros is considered to pay no penalty on run-time performance. Efficient execution can only be realized through compilation anyway.

As macros are expanded in compilation time, macros should not refer to global variables. The expansion should be the same in any context (on the assumption that, of course, CAR still means CAR, CDR means CDR, etc).

Macros cannot be applied to arguments in the same way as usual functions. Macros takes arguments which are not evaluated yet, while application is calling a function with already evaluated arguments. Thus, calling FUNCALL or APPLY with macros as the first argument will generate an error.

## DEFMACRO FACILITY

Complicated macros must have access to structural details of their argument lists. Such an access requires densely nested CAR and CDR functions, which may not only increase the difficulty of programming but also damage the readability of the resulting program. The DEFMACRO facility is provided to facilitate access to portions of the argument list by giving names to portions of the argument list.

DEFMACRO {Macro} name arg-pattern . body

A DEFMACRO form of the syntax

```
(DEFMACRO name arg-pattern . body)
```

is expanded into

```
(MACRO name (@) . expanded-body)
```

where <arg-pattern> may be an arbitrarily complicated tree structure of symbols, which serves as a template of the argument list. Its “car” represents the “car” of the argument list, its “cdr” represents the “cdr” of the list. <expanded-body> is almost the same as <body> except that all the accesses to the symbols in <arg-pattern> are converted to accesses to corresponding portions of the argument list.

For example, the WHILE–DO in the former example can be more elegantly defined using DEFMACRO as follows:

```
(DEFMACRO WHILE-DO (CONDITION . BODY)
      (NCONC (LIST 'LOOP
                   (LIST 'AND CONDITION '(EXIT)))
             BODY))
```

## BACKQUOTE FACILITY

It is still not easy to define a macro even with DEFMACRO. The difficulty lies in the fact that two different forms must be considered at one time: The expanded form which will finally be evaluated is one; the form which produces that form is the other, and this form is what the programmer has to write down. The backquote facility is provided to facilitate the construction of the latter.

The backquote character (`) is defined as a read macro (see the section *Input and Output* for details), which acts similarly to a normal single quote (') that makes a QUOTEd form of the S-expression following it. However, when a form included in the following S-expression is preceded by a comma (,), that form is not QUOTEd while all the other portions are effectively QUOTEd.

The following examples illustrate the use of the backquote facility.

`X is read in as (QUOTE X) which is the same as 'X.

`(A ,B C) is read in as (LIST 'A B 'C). As B is not quoted, it is evaluated when the whole form is evaluated.

The WHILE–DO macro can be still more elegantly defined as

```
(DEFMACRO WHILE-DO (CONDITION . BODY)
     `(LOOP (AND ,CONDITION (EXIT))
            . ,BODY))
```

Backquotes can be nested. When backquotes are nested twice, a double comma will cause a form to be evaluated in the first evaluation of the whole form; a form preceded by a single comma will be evaluated in its second evaluation.

# INPUT AND OUTPUT

## STREAMS

Streams are Lisp objects through which I/O operations are performed. Streams may be connected to an external file or to the user terminal. File streams are created by the function STREAM. They should be opened by the functions INOPEN or OUTOPEN before being used.

Any number of streams can be connected to a single external file. It is also possible to open two or more streams connected to one file in output mode. However, it is difficult to predict the result of output operations in such cases, since the files are modified through file buffers.

STREAM streamid

> STREAM makes a stream which is connected to the external file defined by <streamid>, which must be a string of one to eight characters, except in MTS where it is a FDUB pointer or a logical I/O unit name. These can be defined using Lisp functions ALLOC or CALLTSS; they can also be defined using the commands of the operating system (see the next section for details).

INOPEN stream

> INOPEN opens <stream> as an input stream. When opening is unsuccessful, an error is generated; otherwise, it returns <stream>. Record formats which the current system can deal with as input files are F (Fixed length), FB (Fixed length, Blocked), V (Variable length), and VB (Variable length, Blocked).

OUTOPEN stream (lrecl) (blksize)

> OUTOPEN opens <stream> as an output file. If <lrecl> is given, the file will have the logical record length of <lrecl>. If <blksize> is given, the file will have the block size of <blksize>. If either of these is not given, the information is extracted in some other way (from the file label, for example). If no information is available, the default values for <lrecl> and <blksize> are used, which are 255 and 2560, respectively. When opening is unsuccessful, an error is generated; otherwise, OUTOPEN returns <stream>.
>
> The system opens the file as a VB (Variable length, Blocked) format file.
>
> The <blksize> parameter is irrelevant in MTS and is ignored.

CLOSE stream

> CLOSE closes the file associated with <stream>. When closing is unsuccessful, an error is generated; otherwise, it returns <stream>.

OPENFILES {Variable}

> The value of OPENFILES is a list of streams which are currently open. The most recently opened stream comes first in the list. The list is automatically maintained by INOPEN, OUTOPEN, and CLOSE; the user may not update the value of OPENFILES

explicitly. All the files currently open can be closed by:

```
        (MAPC OPENFILES (FUNCTION CLOSE))
```

STREAM–MODE stream

STREAM–MODE returns the current state of <stream>; if it is open as an input file, it returns the symbol INOPEN; if it is open as an output file, it returns the symbol OUTOPEN; if it is not open, it returns NIL.

LINELENGTH (stream)

For input streams, LINELENGTH returns the length of the current line. For output streams, it returns the maximum line length allowed. The default value for <stream> is the value of STANDARD–OUTPUT.

Note: In operating systems other than MTS, when <stream> has the VB format, its LINELENGTH is 4 less than its logical record length since LINELENGTH does not include the size of the record descriptor field.

LINESIZE (size)

With no argument, LINESIZE returns the line length of the terminal as a “fixnum”. If the optional argument <size> is supplied, the line length of the terminal stream is set to <size>; <size> must be positive and less than 256. In the latter case, LINESIZE returns this new line length as its value.

CURSOR (stream)

CURSOR returns the current column position of <stream>, with column zero being the first column. The default value of <stream> is the value of STANDARD–OUTPUT.

COLLEFT (stream)

When <stream> is an input stream, COLLEFT returns how many more characters exist in the current line; if it is an output stream, it returns how many more characters can be printed on the current line. The default value of <stream> is the value of STANDARD–OUTPUT.

Note: CURSOR + COLLEFT is always equal to LINELENGTH.

STANDARD–INPUT {Variable}
STANDARD–OUTPUT {Variable}

The values of these variables are streams for which I/O operations are normally performed; the values of these variables are used as the default values of stream arguments in various I/O functions. Reading and printing can be elegantly directed to a desired stream by lambda-binding these variables to the stream. Using this style, these variables will recover their old values when they are unbound.

In MTS, the initial value of STANDARD–INPUT is a stream attached to SCARDS and the initial value of STANDARD–OUTPUT is a stream attached to SPRINT.

In other operating systems, the initial of STANDARD–INPUT and STANDARD–OUTPUT are the same as those of TERMINAL–INPUT and TERMINAL–OUTPUT, respectively, which are the streams connected with the user terminal (see below). For example,

```
(LET ((STANDARD-INPUT some-stream)) (READ))
```

is effectively the same as:

```
(READ some-stream)
```

TERMINAL–INPUT {Variable}
TERMINAL–OUTPUT {Variable}

The values of these variables are the streams which are connected to the user's terminal. In MTS, the streams are attached to GUSER and SERCOM.

For example, while the standard output stream is directed to some file stream, messages to the terminal can be explicitly directed to the terminal as in the following example:

```
(LET ((STANDARD-OUTPUT some-stream))
     (COND ((NULL L)
                (PRINT "L is null" TERMINAL-OUTPUT))
           (T (MAPC L 'PRINT))))
```

PROMPT {Variable}

The value of PROMPT is a string which is used for prompting input from the terminal. Initial value of PROMPT is ">". It is recommended that subsystems of the Lisp system should bind PROMPT to a certain string which identifies the subsystem to notify the terminal user what the prompting system is, or what kind of input is expected. For example,

```
(SETQ NAME
      (LET ((PROMPT "Who are you?  "))
           (READ)))
```

## ALLOCATING FILES

ALLOC filename

ALLOC allocates a file designated by a string <filename>. (ALLOC "filename") is the same as (CALLTSS "ALLOC DS(filename) SHR"), or giving a command such as "ALLOC DS(filename) SHR" to the TSS command interpreter. The case of characters in the string <filename> is ignored. If the allocation is successful, in MTS it returns a FDUB pointer and in other operating systems it returns a ddname for the file which is a string of 8 characters. If the allocation is unsuccessful, it returns the return code as a "fixnum".

CALLTSS tss-command

CALLTSS executes TSS commands given by a string <tss-command>. Commands which are executable are ALLOCATE, FREE, ATTRIBUTE, and command procedures including only these; other TSS commands can be executed using the function CALL (see the description of CALL in the section *Miscellaneous*). The case of characters in the string <tss-command> is ignored.

If the command is executed successfully and a certain data definition name (ddname) is available, that ddname (a string of 8 characters) is returned; otherwise, if successful and a ddname is not available but a certain data set name (dsname) is available, that dsname (a string of 44 characters) is returned; otherwise, the return code of the command, which is zero, is returned when the command is terminated successfully. For example,

```
(CALLTSS "ALLOC DS(FOOFILE)") => "SYS00013"
        (CALLTSS "ALLOC DD(SYS00013)")
     => "A0000.FOOFILE                                "
```

Note: The dsname always has 44 characters regardless of its real length. Blanks are used for padding if the file name is shorter than 44 characters.

The CALLTSS function is not implemented in MTS.

FILE–STREAM filename (member)

FILE–STREAM returns a stream object connected to the file specified by an FDname in MTS. <filename> is a string containing the full name of the file in uppercase letters including the user identifier (such as A3840), which cannot be omitted in other operating systems.

When the optional argument <member> is supplied, the file specified by <filename> must be a PO (Partition Organized) file, and the member of the PO file indicated by <member> will be the file specified; otherwise, the file is regarded to be a simple sequential file. For example,

```
(FILE-STREAM "A3840.LISPLIB.VDATA" "PRIND")
```

In MTS, the optional argument is not allowed.

## PRINTED REPRESENTATION

Lisp objects cannot be directly handled since they are stored inside the machine memory. In order to examine these Lisp objects, the system provides a representation of its objects in the form of printed text. This is called the printed representation.

Functions such as PRINT, PRIN1, and PRINC take a Lisp object as their argument and send the characters of its printed representation to a stream. These functions are known as the printer.

The function READ takes characters from a stream, interprets them as a printed representation of a Lisp object, constructs a corresponding object, and returns it. This function is known as the reader.

This section describes printed representation of various Lisp objects.

## The Printer

Printing is done either with or without “slashification”. The “non-slashified” representation looks simple and readable to human eyes, but they may not be properly read in again by the machine. The “slashified” version can be faithfully converted back into Lisp objects by READ except for some peculiar objects, namely streams, vectors, references, and code pieces.

The printed representation of an object depends upon its type:

For a “fixnum”:

> If the “fixnum” is negative, the printed representation is preceded by a minus sign (“−”) non-negative, no sign is printed. Then comes the decimal representation of the absolute value of the “fixnum”. Slashification does not affect the printing of “fixnums”.

For a “flonum”:

> The printed representation is preceded by a sign (“+” or “−”), then a digit zero (“0”), a decimal point (“.”), and the fraction part which is a sequence of decimal digits. The number of digits in the fraction part is specified by the value of the symbol DIGITS. Then comes the exponentiation part indicator (“↑”), the sign of the exponentiation part (“+” or “−”), and the value of the exponentiation part in two decimal digits. Thus, the number of characters of the printed representation of a “flonum” is, in total, DIGITS + 7. Slashification does not affect the printing of “flonums”.

For a symbol:

> If slashification is off, the printed representation is simply the successive characters of the print-name string of the symbol, except that when the value of USE−LOWER is non-NIL, uppercase characters are converted into corresponding lowercase characters using the value of the symbol LOWER−CASE as the conversion table.
>
> If slashification is on, some special characters are preceded by the escape character “/”. The decision as to whether or not escape is required is made using the current readtable, i.e., the current value of the symbol READTABLE. Objects printed with slashification are always read back faithfully, provided that the same readtable is used as when it is printed out.

For a string:

> If slashification is off, the printed representation is simply the successive characters of the string.
>
> If slashification is on, the string is printed between double quotes ("), and double quotes inside the string are duplicated.

For “cons” cells:

The printed representation for “cons” cells tends to favor lists, rather than dotted pairs. It starts with an open parenthesis. Then, the “car” of the “cons” is printed, and the “cdr” of the “cons” is examined. If it is NIL, a close parenthesis is printed. If it is anything but a “cons”, then a space, a dot, a space, and that object is printed followed by a close parenthesis. If it is a “cons”, a space is printed and the printing starts again all over from the point after the open parenthesis was printed, using this new “cons”. This procedure produces the usual printed representation such as those seen in this manual.

For a code piece:

The printed representation has the syntax C#<name>, where <name> is the name of the code piece, normally the name of the function to which the code piece is associated. Code pieces cannot be read back in.

For other objects:

The printed representation has the syntax <type>#<address>, where <type> is a character indicating the type of the object (“V” for vectors, “R” for references, “S” for streams), and <address> is the decimal representation of the current address of the object. The address is merely for convenience in discriminating between two objects; the objects may be relocated by the garbage collector. Vectors, references, and streams cannot be read back in.

DIGITS {Variable}

The value of DIGITS, which must be a positive “fixnum”, specifies how many digits are to be printed in the fraction part of the printed representation of “flonums”. The initial value of DIGITS is 7, thus the length of the printed representation of a “flonum” is, initially, 14.

USE−LOWER {Variable}

When the value of USE−LOWER is non-NIL, lowercase characters are used in the printed representation of symbols. It does not affect the printed representation of strings. Its initial value is NIL.

ATOMLENGTH x

ATOMLENGTH returns the length of the printed representation of an atom <x>. The printing is assumed to be “slashified”. If <x> is not an atom, an error is generated.

The following additional feature is provided for the printed representation of “cons” cells; as a list is printed, PRINT maintains the length of the list so far, and the depth of recursion of printing lists. If the length exceeds the value of the variable PRINTLENGTH, PRINT will terminate the printed representation of the list with “???” and a close parenthesis. If the depth of recursion exceeds the value of the variable PRINTLEVEL, the list will be printed as “?”. These features allow abbreviated printing which is concise and suppresses detail.

PRINTLEVEL {Variable}
PRINTLENGTH {Variable}

The values of these variables are used as described above. Their initial values are 4 and 10, respectively. Infinitely deep or long printed representation may be obtained by setting these variables to zero.

The special characters used in the printed representations can be changed by setting the value of the symbol SPECIAL–CHARACTERS.

SPECIAL–CHARACTERS {Variable}

The value of SPECIAL–CHARACTERS is a string, the elements of which are characters used for a special purpose in printed representations. The index values, the usage of the corresponding characters, and their initial values are as follows:

| | | |
|---|---|---|
| 0: | “ ” | blank space |
| 1: | “(” | open parenthesis |
| 2: | “)” | close parenthesis |
| 3: | “.” | dot for dotted notation |
| 4: | “/” | escape character |
| 5: | “”“” | string quote |
| 6: | “+” | plus sign |
| 7: | “–” | minus sign |
| 8: | “.” | decimal point |
| 9: | “↑” | exponentiation part indicator |
| 10: | “#” | separator used for codes, streams, etc. |
| 11: | “C” | code piece indicator |
| 12: | “S” | stream indicator |
| 13: | “V” | vector indicator |
| 14: | “R” | reference indicator |

## The Reader

The purpose of the reader is to accept characters, interpret them as the printed representation of a Lisp object, and return a corresponding Lisp object. The reader cannot accept all the printed representations; the printed representations of vectors, references, streams, and code pieces cannot be read in again. However, the reader has many features which are not seen in the printer.

The reader accepts slashified printed representation of numbers, symbols, strings, and conses. Some special characters can be defined as single character objects, which are read in as a one-character symbol of that character. Macro characters can be defined, which when read will cause a call to a function associated with that character. See following sections about the “readtable” and read macros.

Symbols with the same print-name are read as the same object. This is realized by keeping all the useful symbols in a table called the “obvector”. This table is organized as a hash table; the keys used are print-names of symbols. The registration process in the “obvector” is called “interning”.

OBVECTOR {Variable}

The value of OBVECTOR is the current obvector. An "interned" symbol <sy> is a top-level element of the element of the obvector, the index of which is given by:

```
(\ (HASH (PNAME sy))
   (VECTOR-LENGTH OBVECTOR))
```

DEFAULT–OBVECTOR {Variable}

The value of DEFAULT–OBVECTOR is the initial value of OBVECTOR. All the predefined symbols are initially registered in this table.

OBLIST (obvector)

OBLIST returns a list of symbols registered in <obvector>. The default value of <obvector> is the current value of the symbol OBVECTOR. The list is newly created each time this function is called.

INTERN {Variable}

The value of INTERN is the "interning" function used by the reader, which must be a function of one argument. When a character sequence which is to be interpreted as a symbol is encountered, the Lisp reader calls this function with one argument, the string consisting of the characters of that sequence. The result of reading the symbol will be the result of this function.

The initial value of INTERN is the function INTERN (see below). Any user-defined name-table-management discipline can be established by binding INTERN to a user-defined "interning" function.

INTERN string (obvector)

INTERN first converts all the lowercase characters in <string> to the corresponding uppercase characters (<string> is physically altered.) The value of the symbol UPPER-CASE is used for this conversion. Then <obvector> is searched for a symbol which has a print-name which is STRING–EQUAL to <string>. If it is found, INTERN returns that symbol; if not, a new symbol with its print-name being <string> is created, registered in <obvector>, and returned as the value of INTERN. The default value for <obvector> is the current value of the symbol OBVECTOR.

INTERN–SOFT string (obvector)

INTERN–SOFT works almost the same as INTERN except that it does not create a new symbol. <obvector> is searched for a symbol with a print-name which is STRING–EQUAL to <string>. If it is found, a list beginning with that symbol is returned; otherwise NIL is returned.

REMOB symbol (obvector)

REMOB searches <obvector> for a symbol which is EQ to <symbol>. If one is found, it is removed from the table, hiding it from the Lisp reader; if none is found, nothing is done. It returns NIL as its value. The default value of <obvector> is the current value of the symbol OBVECTOR.

## The Readtable

The reader is controlled by a vector called the “readtable”. A “readtable” is a vector consisting of 256 “fixnum” elements, the index of which corresponds to a character of EBCDIK code and indicates the nature of that character.

Currently, only the lower 16 bits of each element are used. Their meanings and the initial values in the default readtable are as follows:

| | |
|---|---|
| X'000001' | (LSB) indicates the character is an ordinary alphabetic character. All the usual characters have this bit on and others off. |
| X'000002' | indicates the character is an extended alphabetic character. This bit is currently not used. |
| X'000004' | indicates the character is a digit. Characters “0” through “9” have this bit on. |
| X'000008' | indicates the character is a sign. “+” and “–” have this bit on. |
| X'000010' | is the “alternate meaning” bit. This bit is used in several ways. For example, “–” has this bit on, while it is off for “+”. |
| X'000020' | indicates the escape character. “/” has this bit on. |
| X'000040' | indicates the character should be slashified in a symbol. Characters with special meaning have this bit on. |
| X'000080' | indicates the character should be slashified when appearing at the top of a symbol. Special characters, signs, and digits have this bit on. |
| X'000100' | indicates a string quote character. Double quote has this bit on. |
| X'000200' | indicates a macro character. The macro definition (a function with no argument) is in the corresponding position in the macrotable. |
| X'000400' | indicates a right parenthesis, “)”. |
| X'000800' | indicates a dotted pair dot, “.”. |
| X'001000' | indicates a left parenthesis, “(”. |
| X'002000' | indicates a blank or something similar, which is normally skipped between lexical elements. |
| X'004000' | indicates a a single character object. |
| X'008000' | indicates the character terminates a symbol or a number. All the special characters have this bit on. |

The “macrotable” is used to hold the definition of macro characters. The definition should be a function of no argument, the result of which is returned as the object read in.

READTABLE {Variable}
MACROTABLE {Variable}

> The values of READTABLE and MACROTABLE are the current readtable and macrotable, respectively. The initial value of these variables are the same as those of DEFAULT–READTABLE and DEFAULT–MACROTABLE, respectively (see below). User-defined readtables or macrotables can be used by binding these variables to certain values.

DEFAULT–READTABLE {Variable}
DEFAULT–MACROTABLE {Variable}

> The value of these variables are the standard readtable and the macrotable of the system.

## <u>Setting Readtable</u>

Characters can be defined as a macro character by the function READMACRO. When the reader encounters a macro character in the input text, a function associated with that character is called. The result of the function is returned as the return value of READ.

READMACRO char fn (readtable) (macrotable)

> <char> is defined as a macro character associated with <fn>. This definition is done in the <readtable> and the <macrotable> given as arguments. If they are absent, current values of READTABLE and MACROTABLE are assumed.
>
> For example, the macro character “'” could have been defined by:

```
(READMACRO (CHARACTER "'")
     (FUNCTION (LAMBDA NIL (LIST 'QUOTE (READ)))))
```

> Note that this works not only for (READ) but also for (READ some-stream); the latter binds the variable STANDARD–INPUT to <some-stream> making (READ) in the definition of the read macro get its input from that stream.
>
> If the backquote character “`” cannot be typed in from a certain terminal, an alternative character, say, “%”, can be set for the backquote macro by:

```
(READMACRO "%" (VREF MACROTABLE 121))
```

> 121 is the EBCDIC code for “`”.

Predefined read macros are quote “'”, backquote “`”, and comma “,” (see the section *Macros* for backquote and comma).

Characters may be defined as single-character objects. When the reader encounters one of them (except when reading characters in a string), it is read as an “interned” single-character symbol, regardless of preceding or following characters. Single-character objects can be defined by the function SINGLE–CHARACTER.

SINGLE–CHARACTER char (readtable)

> <char> is defined as a single-character object in <readtable>. If <readtable> is not supplied, the current value of READTABLE is assumed. For example,
>
> ```
> (SINGLE-CHARACTER "&")
> ```
>
> From then on,
>
> ```
> A&NIL&B
> ```
>
> will be read as 5 symbols: A, &, NIL, &, and B.

## INPUT FUNCTIONS

Functions described in this section bind the variable STANDARD–INPUT to the argument <stream> before reading in any character. Thus, input is always performed on the STANDARD–INPUT stream. The default value of <stream> is the current value of STANDARD–INPUT.

READ (stream)

> READ reads in one printed representation of a Lisp object from <stream> and returns it as its value.

READLINE (stream)

> READLINE reads the current line, from the current position to the line end, and returns a string consisting of the characters read in. The next character input from the stream will be the first character on the next line.

SKIPLINE (stream)

> SKIPLINE works the same as READLINE except that it returns NIL instead of a string.

CURRENT–LINE (stream)

> CURRENT–LINE returns the current line of <stream> as a string object. The returned string includes all the characters in the current input line, regardless of the current character position. The character position is not affected.

TYI (stream)

> TYI inputs one character from <stream> and returns its code as a “fixnum”.

TYIPEEK (stream)

TYIPEEK returns the next character of <stream>. It is different from TYI in that TYIPEEK does not advance the current character position of <stream>. Thus, consecutive calls to TYIPEEK will read the same character repeatedly.

READCH (stream)

READCH is the same as TYI except that instead of returning a character as a “fixnum”, it returns an “interned” symbol, the print-name of which is a one-character string of the character read in.

## OUTPUT FUNCTIONS

The functions in this section first bind the variable STANDARD–OUTPUT to the argument <stream> before any actual output. Thus, output operations are always performed on the STANDARD–OUTPUT stream. The default value for <stream> is the current value of the symbol STANDARD–OUTPUT.

PRIN1 x (stream)

PRIN1 outputs the printed representation of <x> to <stream> with slashification. The value of PRIN1 is <x>.

PRINT x (stream)

PRINT works the same as PRIN1 except that PRINT terminates the current line after printing out.

PRINC x (stream)

PRINC is the same as PRIN1 except that the printing is done without slashification.

TYO char (stream)

TYO outputs the character whose EBCDIK code is specified by <char> to <stream>. TYO returns <char> as its value.

TERPRI (stream)

TERPRI terminates the current line of <stream>. TERPRI returns NIL as its value.

TAB n (stream)

TAB will set the character position of <stream> at the column <n>. If the current character position is less than <n>, spaces are printed out until the column <n> is reached; if the current position exceeds the column <n>, the line is terminated and <n> spaces are put out on the next line. TAB returns NIL as its value.

## FORMATTED PRINTING

It is often required to print an S-expression in the midst of a certain message. For example, given a symbol <sy> and a number <num>, one might require such output as

```
    "The symbol <sy> appeared <num> times."
```

with <sy> and <num> varying from time to time. Of course, this can be achieved by

```
    (PROGN (PRINC "The symbol ")
           (PRIN1 sy)
           (PRINC " appeared ")
           (PRIN1 num)
           (PRINC " times.")
           (TERPRI))
```

but this looks ugly and is not readable.

This kind of output is required so often that the system provides a formatted printing facility.

FORMAT {Macro} pattern . args

FORMAT is a macro for formatted printing. The first argument <pattern> is a string specifying the output format and the rest of the arguments <args> is a list of forms which are evaluated and used according to <pattern>.

The string <pattern> is normally printed out as it is. However, when a slash character (/) is encountered, printing is controlled by the directive character immediately following it. If the directive character requires arguments, the values of <args> are used sequentially from left to right. The control-directive characters currently available and their meanings are as follows:

| | |
|---|---|
| S: | print one S-expression with slashification. |
| C: | print one S-expression without slashification. |
| B: | print one character the code of which is supplied as an argument. |
| G: | pretty-print one S-expression. |
| T: | tabulate to the column specified by the argument. |
| N: | terminate the current line. |
| /: | print "/", i.e., a slash should be doubled. |

The case of directive characters is ignored.

For example, the former example can be printed by

```
    (FORMAT "The symbol /S appeared /S times./N" sy num)
```

## INDENTED PRINTING

Printed representations of S-expressions are not easily examined by human eyes, especially when parentheses are densely nested. The indented printer PRIND will help produce more readable output by giving appropriate indention.

PRIND x (width) (asblock) (level) (length)

<x> is printed with certain indention. <width> is the maximum width for printing, the default value of which is the line length of the current output stream.

When <asblock> is non-NIL, the printout will be more compact than when it is NIL (the readability may be somewhat damaged). The default value of <asblock> is NIL.

When <level> and <length> arguments are supplied, they must be non-negative "fixnums", and when they are non-zero, the maximum level and length of printing lists will be <level> and <length>, respectively.

QUOTE forms such as (QUOTE A) are printed as 'A. Moreover, when the value of the variable USEBQ is non-NIL, backquotes and commas are used for printing CONS and LIST forms; (LIST A 'B C) is printed as `(,A B ,C); (CONS 'A B) as `(A . ,B).

PRIND returns NIL as its value, unlike PRINT which returns its first argument.

USEBQ {Variable}

When the value of USEBQ is non-NIL, backquotes and commas are used in the printout of PRIND. The initial value of USEBQ is NIL.

PP {Macro} funcname

The definition of the symbol <funcname> is printed so that the definition will be recovered when the printout is read in and evaluated. Usual functions are printed as

```
(DEFUN funcname argument-list . body)
```

Macros defined using DEFMACRO are printed as

```
(DEFMACRO funcname argument-pattern . body)
```

Other macros are printed as

```
(MACRO funcname argument-list . body)
```

# CODE PIECES

A code piece is a Lisp object which contains machine-language instructions and some Lisp objects which are accessed from the code. Though code pieces may be used as functions, it is usually more convenient to use their names, i.e., symbols, as functions. Code pieces are either predefined by the system or are obtained by compiling lambda forms.

A code piece has a name, which is normally a function symbol associated with that code piece.

FUNCNAME code

> FUNCNAME returns the name of <code>.

The number of arguments for a code piece may be restricted to reside in some range. The minimum and the maximum numbers of arguments are stored somehow in the code pieces for run-time checking, and can be examined by the following functions.

MINARG code
MAXARG code

> MINARG and MAXARG return the minimum and the maximum number of arguments for <code>, respectively. The values returned by these functions may not always be precise. However, it is guaranteed that an error will be generated when <code> is applied to fewer arguments than the result of MINARG or more than the result of MAXARG. If <code> allows an arbitrary number of arguments, MAXARG returns −1.

A code piece can be constructed by the following function.

LOAD−CODE x

> LOAD−CODE constructs and returns a code piece specified by the argument <x>, which has the syntax

```
(name maxarg machine-code quoted)
```

> where <name> is the name of the function, <maxarg> is the maximum number of arguments of the function, <machine-code> is a list of “fixnums” each of which represents one halfword (16 bits) of the machine code, and, finally, <quoted> is a list of Lisp objects accessed from the machine code.

# COMPILATION

The Lisp compiler is a program which translates interpretive functions in the form of lists into machine code which is directly executed by the hardware. The merit of compilation is that the execution speed will be considerably faster.

## COMPILING FUNCTIONS

COMPILE {Macro} . function-names

The compiler can be evoked by simply applying the macro COMPILE as in

```
(COMPILE . function-names)
```

where <function-names> is a list of symbolic atoms, the definitions of which are lambda forms. The definitions of these symbols will be replaced by the compiled code. COMPILE returns the list <function-names> as its value. For example, the interpretive functions F and G can be compiled by:

```
(COMPILE F G)
```

REVERT {Macro} . function-names

The interpretive definition of a function which is compiled using COMPILE is saved in the property list of the function symbol as its PREVIOUS–DEFINITION property. REVERT sets the definition of the symbols in <function-names> to their PREVIOUS–DEFINITION properties. REVERT returns <function-names> as its value.

The calling interface of compiled and interpretive functions are totally compatible. Thus, a compiled function may call interpretive functions and vice versa.

Macro calls in the definition of the function being compiled are expanded before the compilation. Thus, such macros must be defined before the compilation.

Usually, the compiler generates various run-time check codes. When the program has been completed and there is no possibility of errors, these check codes may be superfluous. The following variables are used to give direction to the compiler as to whether or not such check codes are required.

TYPECHECK {Variable}

When the value of TYPECHECK is non-NIL, the compiler generates type check codes; otherwise no run-time type check code is generated. The initial value of TYPECHECK is T.

UBVCHECK {Variable}

When the value of UBVCHECK is non-NIL, the compiler generates check codes for unbound variables; otherwise unbound variables are not checked in the object code. The initial value of UBVCHECK is T.

INDEXCHECK {Variable}

When the value of INDEXCHECK is non-NIL, the compiler generates check codes for array or string index range; otherwise no run-time check code for index range is generated. The initial value of INDEXCHECK is T.

UDFCHECK {Variable}

When the value of UDFCHECK is non-NIL, the compiler checks for undefined functions when making calls from compiled code; otherwise no undefined-function checking is performed. The initial value of UDFCHECK is T.

## DECLARATION

Various declarations may be required for exact compilation. The macro DECLARE and the function RESET–COMPILATION–FLAGS are provided for such declarations.

DECLARE {Macro} item-list indicator

DECLARE is used to declare that the elements of <item-list> have the attribute indicated by <indicator>. Currently, the indicators used are SPECIAL and REDEFINE.

RESET–COMPILATION–FLAGS

RESET–COMPILATION–FLAGS revokes all the declarations specified so far via the macro DECLARE.

The compiled object is designed so as to use “static” scope rule for local variables (the usual Lisp scope rule is the so-called “dynamic” scope rule). For exact compilation of functions which utilize global variables, all the non-locally referred variables (i.e., variables referred from functions other than those which bind the variable) should be declared to be SPECIAL.

The declaration of SPECIAL variables can be done by

```
(DECLARE var-list SPECIAL)
```

where <var-list> is a list of non-locally referred variables. For example, when variables X and Y are used non-locally, they should be declared SPECIAL before compiling functions which bind them by:

```
(DECLARE (X Y) SPECIAL)
```

If a non-local variable is not properly declared, the compiler treats the variable as a local variable. The value of a local variable is stored somewhere on the system stack and access to it is possible only from the function which binds the variable.

For calls to some of the predefined functions (such as ATOM, CAR, CDR, etc.), the compiler generates certain machine-code sequences which work effectively the same as these functions. Thus, if some of the predefined standard functions are to be redefined by the user program, they should be declared by

```
(DECLARE fn-list REDEFINE)
```

where <fn-list> is a list of the names of predefined functions which are to be redefined.

## STORING COMPILED OBJECTS

The compiler puts the compiled code in a relocatable form (in the form of a list of numbers and some Lisp objects) in the property list of the symbol which is the name of the compiled function as its COMPILED–CODE property. This can be printed to a file as a normal lisp object and can later be read back in. This relocatable form can be converted into a machine-code object (code piece) by the function LOAD–CODE. The result of LOAD–CODE may be put into the definition cell of the function name by the function PUTD (see the section *Code Pieces* for details).

For example, if the relocatable compiled code for the function F is stored in the file connected to an input stream which is the value of the variable OBJ, the definition of F can be loaded by:

```
(PUTD 'F (LOAD-CODE (READ OBJ)))
```

## DIFFERENCE FROM THE INTERPRETER

Because the compiled object is designed to attain efficient execution, some differences exist between the run-time behaviour of compiled codes and interpretive codes.

Non-local GO and RETURN in PROG forms as well as non-local EXIT in LOOP forms are not allowed in compiled functions. The only available non-local exit structure is that provided by CATCH and THROW.

Because of certain difficulty in implementation, the arguments of CATCH are evaluated interpretively. Thus, variables which are referred to in the arguments of CATCH should be declared to be SPECIAL.

## PROVIDING SPACE FOR COMPILED CODES

Compiled codes are stored in an area called FIXED–HEAP which is different from the usual HEAP for ordinary Lisp objects. When a large amount of code has to be compiled, the size of the FIXED–HEAP must be specified to be large enough. This is done by supplying an optional parameter "FIX" on the MTS command to invoke "LISP":

```
Run *UTILISP PAR=FIX(n)
```

where <n> is a number indicating how many pages (1024 words/page) should be provided for compiled objects. The default value of <n> is 8.

As the garbage collector does not collect garbage in the FIXED–HEAP area, the recompilation of functions leaves uncollectable garbage. See the section *Memory Management System* for details.

# ERRORS AND DEBUGGING

## THE ERROR SYSTEM

The system generates an error when the Lisp program attempts some invalid operation; for example, when the “car” of an atom is taken.

When an error is generated, the value of the symbol corresponding to the kind of error is examined. The value is interpreted as a function, which is called by the system with one argument; the kind of argument depends upon the kind of error. The initial value for these symbols are the symbols themselves. These symbols are defined as standard error handlers. They print an appropriate error diagnostic message, the information passed as the argument, and the name of the function in which, or while evaluating the arguments of which, the error took place.

Then the value of the symbol BREAK which must be a function of no argument is examined, and this function is then called in the environment where the error occurred (the variables have the same values as when the error took place). The result of this function call will be the result of the function call which generated the original error.

BREAK {Variable}

> The value of BREAK is a function which is called by the standard error handlers after printing error diagnostics. The initial value of BREAK is BREAK itself.

BREAK

> BREAK first binds STANDARD–INPUT and –OUTPUT to the streams connected to the terminal, READTABLE and MACROTABLE to the standard ones, PROMPT to the string “@”, and then enters a READ–EVAL–PRINT loop similar to the top-level loop. This loop can be terminated by the function UNBREAK (see below).

UNBREAK . args

> UNBREAK can be used to terminate a BREAK loop. The innermost BREAK loop is terminated and the value returned by that BREAK will be the last argument of UNBREAK. If no BREAK encloses an UNBREAK call, an error is generated.

The following are the variables whose values are used as error handlers and, at the same time, function names of the standard error handlers. The initial values of these variables are themselves. The optional argument <where> is interpreted as the function name where the error has occurred. The default value of <where> is the function from which the error handler is called. When an error handler is to be called explicitly (usually by FUNCALL), an appropriate function name should be given for this optional argument. For example, the function CADR could have been defined as:

```
(DEFUN CADR (X)
       (COND ((OR (ATOM X) (ATOM (CDR X)))
                  (FUNCALL ERR:ARGUMENT-TYPE X 'CADR))
             (T (CAR (CDR X)))))
```

The variables are:

ERR:ARGUMENT–TYPE {Variable}
ERR:ARGUMENT–TYPE x (where)

The type of <x> was not valid for the function applied to it.

ERR:BUFFER–OVERFLOW {Variable}
ERR:BUFFER–OVERFLOW dc (where)

A string or a symbol is read in which is longer than the string buffer. The size of the string buffer is, currently, 1000 characters. <dc> is always NIL.

ERR:CATCH {Variable}
ERR:CATCH tag (where)

THROW was called with its first argument being <tag>, but the corresponding CATCH with its first argument EQ to <tag> was not found.

ERR:END–OF–FILE {Variable}
ERR:END–OF–FILE stream (where)

The end of the file was reached while reading the file associated with <stream>. <stream> is automatically closed.

ERR:FLOATING–OVERFLOW {Variable}
ERR:FLOATING–OVERFLOW dc (where)

An arithmetical overflow occurred during a floating-point arithmetic operation. <dc> is always NIL.

ERR:FUNCTION {Variable}
ERR:FUNCTION x (where)

<x> was used as a function but is illegal as a function, i.e., a non-symbolic atom or a “cons” cell which is not a LAMBDA form.

ERR:GO {Variable}
ERR:GO tag (where)

A GO form was evaluated with <tag> but the corresponding PROG that has the label <tag> in its body was not found.

ERR:INDEX {Variable}
ERR:INDEX index (where)

<index> was used as an index for a vector or a string, but is out of index range or is not a “fixnum”.

ERR:IO {Variable}
ERR:IO stream (where)

<stream> was used for some I/O operation but has not been opened properly; an input stream was used for output, the reverse case, or <stream> was not open at all.

ERR:NUMBER–OF–ARGUMENTS {Variable}
ERR:NUMBER–OF–ARGUMENTS dc (where)

The number of arguments for a function was too many or too few. <dc> is always NIL.

ERR:OPEN–CLOSE {Variable}
ERR:OPEN–CLOSE stream (where)

Opening or closing of <stream> failed. Usually, some diagnostic message, besides that of the Lisp system, is printed out by the operating system.

ERR:READ {Variable}
ERR:READ dc (where)

The character sequence read in cannot be interpreted as an S-expression. This error is often caused by an improper usage of dots (".").

ERR:RETURN {Variable}
ERR:RETURN dc (where)

RETURN, EXIT, or UNBREAK was called but the corresponding PROG, LOOP, or BREAK was not found. <dc> is always NIL.

ERR:UNBOUND–VARIABLE {Variable}
ERR:UNBOUND–VARIABLE var (where)

<var> was evaluated but is unbound.

ERR:UNDEFINED-FUNCTION {Variable}
ERR:UNDEFINED-FUNCTION fn (where)

The symbol <fn> was used as a function but is undefined.

ERR:VARIABLE {Variable}
ERR:VARIABLE x (where)

<x> was used as a variable but is not a symbol.

ERR:ZERO–DIVISION {Variable}
ERR:ZERO–DIVISION dc (where)

Division by zero was attempted. This error may occur in both integer and floating arithmetic. <dc> is always NIL.

Two special errors are handled quite differently. They are the overflow of the system stack and the shortage of the available memory. When a stack overflow occurs, or when the garbage collector failed to collect enough memory for computation, a diagnostic message indicating the kind of error is printed, all the variables recover their top-level values, and the system resumes the top-level loop.

When a stack overflow occurs during a garbage collection, the system prints out a message and the Lisp session is terminated abnormally, since such a situation is fatal and recovery is impossible.

## ATTENTION HANDLING

When the execution of a Lisp program is interrupted from the terminal (by an attention interrupt), the attention-interrupt handler is called. If the system is doing a certain I/O operation, this call will be postponed until the termination of that operation.

ATTENTION–HANDLER {Variable}

> The value of ATTENTION–HANDLER is the attention-interrupt handler, which must be a function of no argument. The initial value of ATTENTION–HANDLER is BREAK.

## THE DEBUGGER

The debugger is a collection of functions which are useful in debugging Lisp programs. Because the debugger is designed for interpretive functions, it is recommended that you debug programs in interpretive form before compiling them into machine code (see the section *Compilation* for details).

TRACE {Macro} . funcnames

> TRACE takes an arbitrary number of arguments which are names of interpretive functions. The functions listed in <funcnames> become “traced”; the function name and arguments are printed on entry to these functions, the name and the result of the function are printed on exit, with the nesting level, in appropriate indention.
>
> “Tracing” is done by automatically rewriting the definition of the “traced” functions. This alteration can be undone by the function UNTRACE.

TRACE–WHEN {Macro} pred . funcnames

> TRACE–WHEN is the same as TRACE except that tracing is conditional. The form <pred> is evaluated each time a function listed in <funcnames> is called, and that function call will be “traced” if, and only if, the result of evaluating <pred> is non-NIL. As arguments to the function are already bound when <pred> is evaluated, <pred> may depend upon the arguments.

UNTRACE {Macro} . funcnames

> UNTRACE stops “tracing” of the functions listed in <funcnames>.

BACKTRACE (n)

> With no argument, BACKTRACE returns a list of the names of all the functions which are nested down to the current environment. When <n> is supplied, a list of the names of only the last <n> functions in the nesting is returned. Inside the BREAK loop of the standard error handler, this function can be used to examine the calling sequence up to the point where the error occurred.

OLDVALUE (n)

> With no argument, OLDVALUE returns a list of dotted pairs. The “car” of each pair is a variable which is bound by lambda-binding and the “cdr” is its previous value before the

binding. If the variable had been in unbound state before the binding, its previous value is indicated by the symbol *UBV*. The order in the list is such that recently bound variables come first. When <n> is supplied, only pairs concerning recent <n> bindings are included. This function can be used to get information on the binding history.

TOPLEVEL

TOPLEVEL first undoes all the variable bindings except top-level ones. Then the value of the variable TOPLEVEL is examined. The value must be a function of no argument, and this function is then called. As the initial value of the TOPLEVEL is the symbol UTILISP, TOPLEVEL can be used to resume the top-level loop.

TOPLEVEL {Variable}

The value of TOPLEVEL is a function of no argument which is used as the Lisp top-level. The initial value of TOPLEVEL is UTILISP.

## THE LOW-LEVEL DEBUGGER

The low-level debugger is a collection of functions for debugging the UTILISP system itself. As they are primarily prepared for maintainance of the system, some of them are not safe; misuse of them may cause a fatal error. They should be used with proper knowledge of the physical represenation of various Lisp objects.

ADDRESS x

ADDRESS returns the current memory address of <x> as a “fixnum”. If <x> is a “reference”, the address of the vector element pointed by <x> is returned. If <x> is a “fixnum”, <x> itself is returned.

Note: The Lisp objects may be relocated by the garbage collector, except for those which are allocated in the FIXED–HEAP area.

PEEK addr length

PEEK returns a string which contains a copy of the machine memory beginning at (ADDRESS addr) and is <length> long.

# MEMORY MANAGEMENT SYSTEM

The memory space used by the UTILISP system is divided into six areas:

| | |
|---|---|
| HEAP | for usual Lisp objects |
| FIXED–HEAP | for predefined objects and compiled codes |
| STREAM–HEAP | for streams |
| STACK | for control information and temporary storage |
| KERNEL | for the system kernel |
| SAVED | for I/O buffer and external programs (Not relevant in MTS) |

The sizes of the FIXED–HEAP, STACK, and SAVED areas can be specified by the parameters at the initiation of the Lisp system (see the section *Introduction* for details). The sizes of the STREAM–HEAP and KERNEL areas are system-defined constants. The size of the HEAP area can be set by the function HEAP–SIZE. Because the system uses the "copying garbage collection" scheme, the maximum size allowed for the HEAP area is half the memory size available. The size of the area available for the HEAP area is the maximum memory size allowed for a user job by the operating system minus the total size of all the other areas.

When an object is to be allocated, by CONS for example, and insufficient space is left in the HEAP area (or STREAM–HEAP for streams), the garbage collector is called.

The garbage collector gathers all the Lisp objects which cannot be accessed and the memory space occupied by them is reused. The execution of the original program is resumed.

The garbage collector of the UTILISP system can be used not only for collecting garbage but also for compacting and linearizing Lisp objects to keep the "working set" small. For this purpose, the garbage collector can be called before the available space has been completely exhausted; it is automatically called when the size of the HEAP area reaches the size set by the system. This size is twice as large as the area occupied by accessible Lisp objects after the previous collection, but must be between the minimum specified and the maximum allowed. Initially, this minimum size is set equal to the maximum. Thus, the garbage collector is invoked only when the available space has been completely exhausted.

The garbage collector can also be called explicitly by the function GC.

GC

GC evokes the garbage collector. It returns NIL as its value.

The following functions ask for the status of and set the parameters of the memory management system. The unit of memory space used in these functions is "word" (four bytes).

GCCOUNT

GCCOUNT returns a "fixnum" which indicates how many times the garbage collector has been called so far.

GCTIME

GCTIME returns a “fixnum” indicating CPU time required for GC so far. The unit used is the millisecond.

HEAP–SIZE (size)

With no argument, HEAP–SIZE returns a list of the form

```
(heap fixed)
```

where <heap> is the current size of the HEAP area and <fixed> is the size of the FIXED–HEAP area. When <size> is supplied, the size of the HEAP area is set to <size>. <size> must be a “fixnum” greater than the size of the HEAP area in use, and less than the allowed maximum.

MINIMUM–HEAP–SIZE (size)

With no argument, MINIMUM–HEAP–SIZE returns the minimum heap size used to decide the size of the HEAP area after garbage collections. When <size> is supplied, the minimum heap size is set to <size>.

MAXIMUM–HEAP–SIZE

MAXIMUM–HEAP–SIZE returns the maximum size allowed for the HEAP area.

HEAP–USED

HEAP–USED returns a list of the form

```
(current cumulative fixed)
```

<current> is the size of the HEAP area currently in use. As this includes garbages, it is precise only immediately after garbage collections. <cumulative> is the size of the HEAP area currently in use plus the total size of the HEAP area collected by the garbage collector so far. This size is, of course, equal to the total size of the Lisp objects allocated in the HEAP area so far. <fixed> is the size of the FIXED–HEAP area currently in use.

STACK–SIZE

STACK–SIZE returns the size of the STACK area.

STACK–USED

STACK–USED returns the size of the STACK area currently in use.

# STRUCTURE EDITOR — USE

USE (Utilisp Structure Editor) is a structure-oriented editor for inspecting and changing list structures, which may be Lisp programs or data.

One merit of using USE, compared with text-oriented editors, is that editing is done on Lisp data structures themselves, rather than on their printed representations; USE has the knowledge of the hierarchical structure of the edited data, and the editing commands of USE reflect this hierarchy. Another merit is that the editing is done in the Lisp environment; an arbitrary Lisp form can be evaluated during editing and currently edited structures can also be manipulated by Lisp functions.

USE always manipulates a copy of the original list structure. All the atoms in the copy are the same as the original ones, but all the "cons" cells are newly created for the copy. Thus, when a USE session is aborted by a K (Kill) command, the original program or data is not affected at all.

## INVOKING USE

The following macros are used to invoke USE and, on their normal termination, restore the edited result.

ED {Macro} fn

> ED invokes USE for editing interpretive functions. The definition of <fn> will be edited. ED puts the edited result in the definition cell of <fn> on normal termination.

EDV {Macro} var

> EDV invokes USE for editing values of variables. The value of <var> is edited. EDV sets the edited result in the definition cell of <var> on normal termination.

EDP {Macro} sym

> EDP invokes USE for editing properties of symbols. The property list of <sym> is edited, and the result will become the property list of <sym> on normal termination.

EDF {Macro} file-name

> EDF invokes USE for editing text files containing printed representations of Lisp objects. <file-name> is evaluated first. Its result must be a string representing the name of the file to be edited. It is often convenient to set a file name to a variable and use that variable as the argument of EDF, since a file name may be quite complicated. A string indicating the file name may also be used directly as an argument of EDF, since a string is evaluated to itself.
>
> EDF reads all the objects in the file, makes a list of them, and then edits that list.
>
> The order of the elements in the list is the same as in the file. The top-level elements of the result of editing will be printed back to the file on normal termination.

Note: Rewriting an external file does not affect the state of the Lisp objects, even if the file contains function definitions such as DEFUN or MACRO forms. The content of the file must be evaluated to effect redefinition.

EDL {Macro} loc

EDL invokes USE for editing some data which cannot be edited through ED, EDV, or EDP. <loc> should be a form to access a component of a certain structure, for example, (CAR cons) to access the "car" of a "cons" cell <cons>, (VREF vec n) to access the <n>th element of a vector <vec>. <loc> is evaluated first, and its value will be edited by USE. The result will be put back where it came from on normal termination.

USE x

USE is the USE system itself. It is normally called using the above-mentioned macros, but users may also call USE directly. USE returns one component list of the edited results when the USE session is normally terminated; it returns NIL when it is terminated abnormally (by K command).

## USE SESSION

USE prompts for terminal input with a "?" when it is expecting a command. If a symbol or a number is is typed in, it is interpreted as a command; otherwise, especially when you type in a list, the list is interpreted as a form which is evaluated and the result is printed. When the form is evaluated, the symbol ? is bound to the current "scope" (see the next section for details). The E command can be used to evaluate an atom.

E form

The form <form> is evaluated and the result is printed.

Arbitrarily many commands and their operands may be typed on a single line. They are executed sequentially, as long as no error is found. When an error is found, the rest of the current input line will be ignored.

The following commands terminate a USE session.

Q

Q (Quit) terminates the current session normally. If USE was called from one of the macros described in the previous section, the edited result is restored accordingly.

K

K (Kill) terminates the current session abnormally. If USE was called by one of the macros described in the previous section, the edited result is merely discarded, and the original definition, value, property list, etc. are not affected.

## SCOPE AND POSITION NUMBERS

The editor always has a current "scope", which is a portion of the whole S-expression being edited. This "scope" can be nested. The current scope can be an element of its parent scope, and this parent scope can have its parent, and so on. All insertion, deletion, and replacement are effected only inside the current scope.

When the current scope is a list, elements in the list can be specified by their positions. A positive "fixnum" n represents the n'th element. A negative number –n represents the n'th element counted from the last. 1 is the first element and –1 is the last.

For example, if the current scope is (A B C D E F G), then

 1 means A
 3 means C
–1 means G
–3 means E
 10 is invalid
–10 is invalid

## PATTERN MATCHING RULES

Sometimes it is desirable to search for a certain pattern of S-expressions rather than a particular S-expression. For example, one may want to search for a form SETQing to a symbol X without regard for the value assigned. This example can be expressed as (SETQ X ?).

The rules of pattern matching are quite simple:

(1) A pattern matches an S-expression EQUAL to it.
(2) ? matches any S-expression.
(3) ??? matches any portion of a list.

The following examples illustrate the above rules.

```
(CAR X) matches (CAR X) but not (CAR '(A B))
(CAR ?) matches both (CAR X) and (CAR '(A B))
(CONS ?  ?) matches both (CONS X X) and (CONS X Y)
(LIST ???) matches both (LIST) and (LIST X Y Z)
(A ??? Z) matches any of (A Z), (A B Z), or (A B C D E Z)
```

## PRINTING CURRENT SCOPE

P

The P (Print) command prints the current scope in the usual abbreviated way. See the section *Input and Output* for abbreviated printing.

PP

The PP (Pretty Print) command prints the current scope with appropriate indention. See the section *Input and Output* for pretty-printing.

LEVEL n
LENGTH n

The LEVEL and LENGTH commands are used to set the maximum printing level and length in abbreviated printing to <n>. <n> should be a "fixnum".

Note: The level and the length specified by these commands are only effective in one USE session. The values of PRINTLEVEL and PRINTLENGTH are not affected.

## CHANGING THE SCOPE

n
−n
0

Position numbers are commands that change the scope to the position specified. The command 0 changes the scope to the parent of the current scope, i.e., a list which contains the current scope as its element.

TOP

The TOP command changes the scope to the whole S-expression being edited.

N

The N (Next) command moves the scope to the element next to the current scope in the parent scope. When there is no parent scope or the current scope is the last element of the parent scope, it is an error.

B *******CHECK THIS**********

The L (Last) command moves the scope to the previous element before the current scope in the parent scope. When there is no parent scope or the current scope is the first element of the parent scope, it is an error.

W

The W (Where) command prints the location of the current scope beginning from the top level.

## SEARCHING

F pattern

The F (Find) command searches for an S-expression which "matches" <pattern> in textual order (searches "car" before "cdr"). Searching is done in the current scope only. If one is found, the scope is changed to the S-expression found. The intermediate scopes are saved and can be accessed using the command "0". If <pattern> is not found, a message is generated and the scope remains unchanged.

FF pattern

The FF (Find Forward) command is the same as the F command except that the search begins after the current scope.

FB pattern

The FB (Find Backward) command is the same as the F command except that the search is performed in reverse direction ("cdr" before "car") and the search begins before the current scope.

FN

The FN (Find Next) command is the same as the FF command except that the same <pattern> is used as for the previous F, FF, or FB command.

## INSERTING AND DELETING PARENTHESES

BI m n

The BI (Both In) command inserts an open parenthesis to the left of <m> and a close parenthesis to the right of <n>. <m> and <n> are position numbers. For example,

```
BI 2 4
(A B C D E) ==> (A (B C D) E)
```

BO n

The BO (Both Out) command deletes the parentheses enclosing <n> which should be a list. <n> is a position number. It is the inverse operation of BI. For example,

```
BO 2
(A (B C D) E) ==> (A B C D E)
```

LI n

The LI (Left In) command inserts an open parenthesis to the left of <n>. <n> is a position number. For example,

```
LI 2
(A B C D E) ==> (A (B C D E))
```

RI n

The RI (Right In) command inserts a close parenthesis to the right of <n>. <n> is a position number. For example,

```
RI 2
(A B C D E) ==> ((A B) C D E)
```

LO n

The LO (Left Out) command moves the open parenthesis of <n> to the beginning of the current scope. <n> must a position number which specifies a list. For example,

```
LO 2
(A (B C D) E) ==> ((A B C D) E)
```

RO n

The RO (Right Out) command moves the close parenthesis of <n> to the end of the current scope. <n> should be a position number specifying a list. For example,

```
RO 2
(A (B C D) E) ==> (A (B C D E))
```

## INSERTING AND DELETING S-EXPRESSIONS

I pos sexpr

The I (Insert) command inserts <sexpr> to the right of <pos>. If <pos> is a number, it is interpreted as a position number. Otherwise, it is interpreted as a pattern and the first S-expression found to match <pos> is assumed. To use a number as a pattern, quote the number like '3. In this case, the quote is not included in the pattern used for matching. For example,

```
(A B C D E)
   I 3 X
==> (A B C X D E)
   I B (FOO BAR)
==> (A B (FOO BAR) C X D E)
```

Note: Insertion at the top of a list can be achieved by specifying 0 for <pos>.

A sexpr

The A (Append) command replaces the tail of the current scope with <sexpr>. If the current scope is atomic, the whole scope is replaced with <sexpr>. For example,

```
NIL
  A (A B C)
==> (A B C)
  A D
==> (A B C . D)
```

IN sexpr

The IN (Insert Next) commands inserts <sexpr> to the right of the current scope in the parent scope.

D pos

The D (Delete) command deletes <pos> from the current scope. The meaning of <pos> is the same as in the I command. For example,

```
(A B C D E)
  D 3
==> (A B D E)
  D B
==> (A D E)
```

Y pos

The Y (Yank) command inserts an S-expression most recently saved by USE to the right of <pos>. What is saved is either the S-expression deleted using the D command, the S-expression replaced using R command, or the result of evaluating a form which is typed in instead of a command. The meaning of <pos> is the same as in the I command. This command can be used, with the D command, to move a portion of an S-expression inside the edited structure. For example,

```
(A B C D E)
  D 3
==> (A B D E)
  Y A
==> (A C B D E)
  (CONS 'A 'B) => (A . B)
  Y 3
==> (A C B (A . B) E)
```

## REPLACING S-EXPRESSIONS

R pos expr

The R (Replace) command replaces <pos> with <expr>. <pos> has the same meaning as in the I command. For example,

```
R 3 (FOO BAR)
(A B C D E) ==> (A B (FOO BAR) D E)
```

RA pattern expr

The RA (Replace All) command replaces all S-expressions in the current scope which match <pattern> with <expr>. The number of actual replacements is reported. For example,

```
RA X Y
(A X B X C) ==> (A Y B Y C)
2 OCCURRENCES ARE REPLACED
```

# CALLING EXTERNAL PROGRAMS

External programs, such as TSS command processors or assembly language routines, can be invoked from the Lisp system using the functions introduced in this section. Data can be passed as the parameters to external programs or as the values of "command symbols" (see the VOS3 manuals for details on "command symbols").

## CALLING TSS COMMANDS

CALL command (param)

The command indicated by <command> is called. A string <param>, if present, is passed to the command processor as its parameter string. It returns NIL if the execution of <command> is terminated normally; otherwise, it returns the return code of the <command> as a "fixnum".

Note: In MTS, the CALL function can be used to execute any MTS command except RUN, RERUN, LOAD, and DEBUG (in these cases, UTILISP would be unloaded).

Note: In TSS, command procedures and priviledged commands (including commands concerning file protection) cannot be executed using CALL.

## CALLING USER-DEFINED PROGRAMS

User-defined programs (which may be coded in the assembly language) can be loaded to the memory using the function PROGRAM–LOAD. Such a program can be invoked using PROGRAM–CALL and it can be removed from the memory space with PROGRAM–DELETE when it is no longer necessary.

The calling interface to the program is as follows:

| | |
|---|---|
| Register 1: | the address of the parameter list which consists of four words. The first word contains an address of a word which contains a pointer to a Lisp object specified as the parameter. The second word contains the address of the UPT, the third word the address of the PSCB, and the fourth word the address of the ECT. See the VOS3 manual number 8090-3-007 for information on UPT, PSCB, and ECT. |
| Register 13: | the address of the register save area. |
| Register 14: | the return address. |
| Register 15: | the entry address of the called program. |

When a program called by the function PROGRAM–CALL is terminated abnormally, the Lisp system itself also terminates abnormally. Thus, such a program must be coded with certain care.

If the program is to be called more than once, the REUS attribute must be specified when the program is linked (see the VOS3 manual 8080-3-301 for information on REUS attribute).

PROGRAM–LOAD name ddname

> The program specified by <name> is loaded from the file which is assigned the data definition name <ddname>. When the loading is successible, the symbol T is returned; otherwise, the returned code of the LOAD macro is returned.

PROGRAM–CALL entry (param)

> The program specified by <entry> is called with the interface described above. PROGRAM–CALL returns the return code of the called program.

PROGRAM–DELETE entry

> The program which has the entry name <entry> is deleted. PROGRAM–DELETE returns NIL as its value.

## MANIPULATING COMMAND SYMBOLS

Note: This section is not relevant to MTS. It applies only to Hitachi systems.

Command symbols can hold either an integer value or a character string value. The values of the command symbols can be defined and detected using the following functions.

The name of a command symbol must begin with an uppercase letter followed by an arbitrary sequence of uppercase letters and digits, and must not be longer than 32 characters.

DEFCS name value

> If the command symbol specified by <name> has not been defined, a new command symbol whose name being <name> is defined and given the value specified by the argument <value>. If it is already defined, <value> is set as the new value of the command symbol. <value> must be a “fixnum” or a string containing upto 256 characters. DEFCS returns <value> as its value.

DETCS name

> If the command symbol specified by <name> is defined, DETCS returns its value which is a “fixnum” or a string. If it is not defined, NIL is returned.

DELCS name

> If the command symbol specified by <name> is defined, it is deleted. If it is not defined, DELCS does nothing. DELCS returns NIL.

# MISCELLANEOUS

This section describes functions that do not seem to fit anywhere else.

TIME (form)

With no argument, TIME returns the CPU time used by the Lisp system so far. This includes the time required for garbage collection and for external programs which are called using PROGRAM–CALL, but excludes the time consumed by external programs called using CALL. If the optional argument <form> is supplied, <form> is evaluated, and CPU time required for this evaluation is returned. The time is returned as a “fixnum” object in milliseconds.

QUIT (retcode)

QUIT will return control to the caller of the Lisp system, usually to the operating system’s command level, with return code <retcode>. The default value of <retcode> is 0. All the files opened by the Lisp system will be automatically closed.

ABEND (retcode)

ABEND abnormally terminates the Lisp system with return code <retcode>. The default value of <retcode> is 4095.

VERSION {Variable}

The value of VERSION is a string which indicates the version of the system.

NEWS

NEWS will print out news from the implementer.

HELP {Macro} sym

HELP can be used to get information on the symbol <sym> from this manual. The portion of the manual beginning with the title <sym> is printed. This printing can be stopped by an attention interrupt (usually by the break key). HELP returns NIL as its value.

DATE–TIME

DATE-TIME returns a string containing the date and time. The string has the format

```
"YYMMDDHHMMSSCC"
```

where YY are the two least significant digits of the year, MM is the month, DD is the day, HH is the hour in the 24-hour system, MM are the minutes, SS are the seconds, and CC are the centiseconds. For example, at 5:30 in the evening of January the 20th, 1981,

```
(DATE-TIME) => "81012017300000"
```

USERID

USERID returns the user-ID of the current session as a string.

UTILISP

UTILISP is the top-level loop of the UTILISP system. An S-expression is read in, evaluated, and printed. This is repeated again and again. The prompting character of the top-level loop is “>”. The symbol UTILISP is the initial value of the symbol TOPLEVEL (see the section *Errors and Debugging* for details).

? {Variable}

Each time that a form read in is evaluated in a UTILISP loop or in a BREAK loop, the result is set to the variable ?. For example, in the top-level UTILISP loop,

```
(CONS 'FOO 'BAR) => (FOO . BAR)
? => (FOO . BAR)
```

# APPENDIX A
# SYMBOL INDEX

# PROLOG/KR

# INTRODUCTION

## THE LANGUAGE

Prolog/KR is an interactive programming system designed to support programs manipulating symbols and structures, especially AI (Artificial Intelligence) programs embedding knowledges (including control knowledges); KR stands for Knowledge Representation. Prolog/KR is equipped with some of the basic tools used in AI programs: backtracking, pattern matching, and database manipulations. Moreover, manipulating structures in Prolog is much simpler than in Lisp. A programmer can start at a point closer to the solution.

Prolog/KR accepts more than one assertion about one predicate. Each assertion is considered to express some knowledge (possibly including control knowledge) about the predicate. At a certain point of computation, when more than one assertion can be used, the system tries them one by one—if some inconsistency arises later, the system backtracks to the last choice point and tries the next assertion.

In representing knowledge, a procedural way and a declarative way are often distinguished. This is however a matter of degree. In Prolog/KR, both ways can be used. Prolog/KR can be used as a procedural language like Lisp by providing every control explicitly. In that case, no automatic backtracking occurs, and thus every computation is deterministic. If no controls are provided, on the other hand, automatic backtracking does its job, and the computation is non-deterministic. In usual cases, programming is done in a mixture of these two extremes.

## NOTATIONAL CONVENTIONS

Lowercase letters are usually converted into uppercase letters when read by the interpreter, i.e., there is no distinction between upper and lowercase letters. Nevertheless, they are distinguished in this manual. Symbols which consist only of lowercase letters denote meta symbols, while those consisting of uppercase letters denote the symbol themselves (they must be spelled as they appear in this manual). Mixed-case indicates that the word may be abbreviated to only the uppercase letters. For example, “PrettyPrint” may be spelled either “PP” or “PRETTYPRINT”.

Meanings and functions of system-defined predicates are explained in the following form:

```
<predicate name> <arguments>
    ........ <descriptions> ........
    ................................
```

<arguments> is shown as a sequence of Prolog/KR variables, possibly followed by “...” indicating repetition of the preceding argument. Arguments are referred to by their names enclosed in “<” and “>” in <descriptions>. Three distinct prefixes are used for explanation:

| | |
|---|---|
| * | This argument may be used both for input and output. |
| < | This argument must be an output variable. In other words, a variable which has no value must be given as an argument to receive a value from the predicate. |

> This variable must be an input variable. Giving an unbound variable is erroneous.

Note: These conventions on variable prefix are used only in this manual. The system does not distinguish those three kinds of variables.

The following examples illustrate the notational conventions.

FOO *e1 *e2

FOO takes two arguments. If FOO is called with a wrong number of arguments, an error is issued. Note that this rule is applicable only to the system-defined predicates. A user-defined predicate “fail”s when it is supplied with a wrong number of arguments.

BAR >e1 *e2

BAR also takes two arguments. <e1> must be either a variable which already has some value, or non-variable object. That is, <e1> is an input argument which expects to receive some value, rather than to return a value. <e2> may be anything, i.e., a variable or an object. For example,

```
(BAR (A B) *X)
```

and

```
(BAR (A B) (C D))
```

are legal, while

```
(BAR *X *Y)
```

is erroneous.

Note: Although the current version of the interpreter treats “>” and “*” equally, users are recommended to use “>” when the variable is supposed to be an input, to increase the readability.

TOng [*e1]

TONG takes zero or one argument. TONG may be abbreviated to TO.

POO [*e1] ...

POO takes any number of arguments (possibly zero).

ZOO (*e1 *e2)

ZOO’s argument must be a list of two elements.

# OBJECTS

The syntax of Prolog/KR objects is described in this section. Unlike other Prolog systems, Prolog/KR's objects (including programs) are expressed as lists or atoms. Thus, every object except variables follows Lisp(Utilisp)'s syntax.

Although Prolog/KR supports all the Utilisp data types as its objects, only the significant ones are described in this section.

## VARIABLE

Variables are atoms (cf. 2.2.) which begin either with "*", ">", or "<", e.g.,

```
*X >X <Y * **THIS_IS_ALSO_A_VARIABLE**
```

where *X and >X are different variables.

Three kinds of prefixes are provided solely for readability. The system does not distinguish those three kinds of variables.

Unlike most of other programming languages, variables are not space holders. Instead, variables are thought to be standing for some Prolog/KR objects. Variables are either undefined or standing for some other objects. When a variable stands for an object, it is said to be "instantiated" and it is just as though the object itself was there. Even if a variable is matched against another variable, they are both considered to be uninstantiated and they will be instantiated to the same object at the same time in the future.

VAR *variable

> Succeeds if <variable> has not been instantiated yet. If <variable> has already been instantiated or is not a variable at all, it fails.

PREFIX >string ...

> Changes the variable prefix to one of <string1>, <string2>, ... Only the first characters in the strings are effective. For example,
>
> ```
> (PREFIX "@@" "*")
> ```
>
> changes the variable prefix to one of "@" and "*". The initial state of Prolog/KR is
>
> ```
> (PREFIX "*" ">" "<")
> ```

There is another kind of variable called an anonymous variable, which is just like a variable except that it has no identity. An anonymous variable is written as

```
?
```

and each occurrence of an anonymous variable stands for a different object. For example,

```
(= (? ?) (A B))
```

succeeds, while

```
(= (* *) (A B))
```

fails.

The anonymous variables are to be used where there is no interest about the object. For example,

```
(ASSERT (CAR (*CARPART . ?) *CARPART))
```

CAR is a predicate to return the first element of a list. Since there is no interest about the rest of the list, "?" is used.

## ATOM

An atom is either a symbol, a number, or a string. Among these, only symbols can be used as predicate names.

A symbol consists of a sequence of characters except special characters. The following special characters may be used in a symbol provided that it is preceded by "/". Here is a summary of meanings of the special characters:

| | |
|---|---|
| | delimiter (blank) |
| ( | beginning of a list |
| ) | end of a list |
| [ | super left parenthesis |
| ] | super right parenthesis |
| ; | beginning of a comment (until the end of the line) |
| / | escape character |
| " " | beginning and end of a string |
| ' | quoter of a pattern |

Examples of symbols are:

```
THESE  ARE  SYMBOLS.  A.B  1+  /(A/)
```

Examples of numbers are:

```
12345 3.14159 1.0E-5 -0
```

Examples of strings are:

```
"123"  "Strings must be surrounded by ""."
```

ATOM is specified in the form

ATOM >x

ATOM succeeds if <x> is an atom; otherwise, it fails.

## LIST

Programs and structured data are expressed in lists. A list is a sequence of objects (including lists) enclosed in parentheses. Dotted notation is used to express the rest of a list. For example, "(A . *X)" matches any list which begins with "A". See the next section about the usage of the dotted notations in pattern matchings. Examples of lists are:

```
(ASSERT (HUMAN TURING))
(LISTS MAY CONTAIN *VAR 123 "string" ...)
(THIS IS A DOTTED . PAIR)
```

## PROGRAM

A Prolog/KR program consists of predicate calls and predicate definitions and both are expressed in lists. A predicate definition consists of one or more assertions. One chunk of knowledge is called an assertion and is added to the system using ASSERT:

```
(ASSERT (FALLIBLE *X) (HUMAN *X))
```

This assertion means:

```
*X is FALLIBLE if *X is HUMAN.
```

Procedurally, it is equivalent to:

```
To execute (FALLIBLE *X), execute (HUMAN *X).
```

The above an example of a predicate call (as well as an assertion) whose predicate name is ASSERT and whose arguments are (FALLIBLE *X) and (HUMAN *X).

When an assertion is added to the system, it is merged into a proper position in the corresponding definition. Details are given in the section *Defining and Modifying Predicates*.

# PATTERN MATCHING

Pattern matching (usually called “unification” in Prolog) is the basic mechanism of calling predicates in Prolog. Variables may occur anywhere in the caller and the callee, and there is no distinction between their roles. Variables may match any Prolog/KR objects including other variables or lists which have variables as their elements.

Since variables may be instantiated to an object which contains variables, it is possible to define a value partially and postpone defining the rest until further computation determines it. One of the most significant examples to demonstrate this feature is APPEND which appends two lists:

```
(ASSERT (APPEND () *ANY *ANY))
(ASSERT (APPEND (*CAR . *CDR) *ANY (*CAR . *REST))
        (APPEND *CDR *ANY *REST))
```

Under the above definition (see the section *Defining and Modifying Predicates* to understand what the above means), a predicate call,

```
(APPEND (A B) (C D) *X)
```

proceeds as follows:

```
(APPEND (A . (B)) (C D) (A . *REST_0001))
```

where *REST_0001 is determined by (APPEND (B) (C D) *REST_0001)

```
(APPEND (B . ()) (C D) (B . *REST_0002))
```

where *REST_0002 is determined by (APPEND () (C D) *REST_0002)

```
(APPEND () (C D) (C D))
```

The result of APPEND is first (A . *REST_0001); then (A B . *REST_0002); and finally (A B C D).

Note that in the above example, dotted notations play important roles. In the pattern “(*CAR . *CDR)”, it is used to decompose a list into its so-called car part and so-called cdr part. In the pattern “(*CAR . *REST)”, on the other hand, it is used to do the reverse, that is, to compose a list from *CAR and *REST. Note that *REST was not defined at the time of the composition. Note also that if APPEND is reversed to (APPEND *X (C D) (A B C D)), the roles of the patterns also are reversed.

MATCH *x *y
= *x *y

Tests if <x> and <y> can be matched. If the matching succeeds, variables in <x> and <y> are instantiated accordingly. Otherwise, MATCH (or =) fails.

CAUTION: If you execute (MATCH *X (F *X)), it succeeds and results in instantiating *X to (F (F (F (F .....

EQ >x >y

Tests if <x> and <y> are the same objects (in the sense of Lisp's EQ). It does not suffice that they are the same patterns. For example:

```
(EQ A A)                      succeeds
(EQ (A B C) (A B C))          fails
(AND (= *X A) (EQ *X A))      succeeds
(EQ *X A)                     fails
```

## EXTENDED FEATURES

### Quoting Patterns

Sometimes it is desired that a certain pattern not match any variables.[2] The pattern should be preceded by a "'" for that purpose. The quoted pattern will only match patterns which are equivalent. For example,

```
(= 'A *X)                   fails
(= 'A A)                    succeeds
(=  A *X)                   succeeds
(= '(A B C) '(A B C))       succeeds
(= '(A B C) (A B *X))       fails
(= '*X *X)                  succeeds
(= '*X *ELSE)               fails
(MEMBER A  (*X B C))        succeeds
(MEMBER 'A (*X B C))        fails
```

Quote can be used as a escape character of an atom which incidentally begins with one of the variable prefixes.

Note: predicates *, >, >=, <, and <= should be used without the quote:

```
(* 512 256 *X)
(> *X *Y)
```

------------------

[2]For example, a negative information is expressed with FAIL:

```
(ASSERT (CAN PENGUIN WALK))
(ASSERT (CAN PENGUIN FLY) (FAIL))
(ASSERT (CAN HAWK FLY))
```

so that (CAN PENGUIN FLY) fails. But if you leave these assertions as they are, even

```
(CAN *X FLY)
```

which is supposed to be a query to search for a flying object, fails before reaching HAWK. To avoid this phenomenon, the previous assertion must be changed to:

```
(ASSERT (CAN 'PENGUIN 'FLY) (FALSE))
```

so that (CAN *X FLY) does not match (CAN 'PENGUIN 'FLY) but (CAN HAWK FLY).

## Executing Predicates within a Pattern

A list which begins with "!" is executed during the pattern matching unless the pattern is inside another pattern which matched a variable. A variable with no names, usually "*", in the rest of the pattern is first instantiated to the target of the pattern matching, and then the pattern is executed. For example, a pattern

```
(! MEMBER * (FOO BAR))
```

matches either FOO or BAR, for

```
(MEMBER * (FOO BAR))
```

is executed with * instantiated to the target of the matching.

Using this feature, a pattern which matches only a pattern whose first element is a variable may be expressed as:

```
((! VAR *) . *ANY)
```

Only the first prefix defined by PREFIX (the default is "*") is considered to be the variable to get the value, i.e., the target of the pattern matching. Thus after executing (PREFIX "@" "#"), "@" is the variable to be instantiated.

Note that no backtracking occurs after finishing the execution. Therefore,

```
(AND (= (! MEMBER * (FOO BAR)) *X)
     (= *X BAR))
```

fails, for * (and thus *X) is bound to FOO once and for all.

Note: Variables instantiated during the execution of the pattern may be unbound on exit. Particularly, the value of "*" is, due to the obvious reason, local to the pattern, and hence invisible from outside. For the value of "*" to be exported, it must be copied to another variable, e.g.,

```
(! AND (VAR *) (= * *THE-VARIABLE))
```

Executable patterns can also be used to simulate "functions". For example, a function FACT (factorial) is defined as:

```
(ASSERT (FACT *N
    (! COND ((= *N 0) (= * 1))
         ((TRUE)
          (TIMES *N (! FACT (! SUB1 *N *) *) *]
```

Note the nesting of executable patterns. To show the correspondence between "!"s and "*"s, a suffixed version (although it is not allowed) of the same example follows:

```
(ASSERT (FACT *N
    (!a COND ((= *N 0) (= *a 1))
         ((TRUE)
          (TIMES *N (!b FACT (!c SUB1 *N *c) *b) *a]
```

Suppose FACT is called by

```
(FACT 5 *F)
```

the following process takes place:

(1) *N = 3
(2) *a = *F
(3) (TIMES 3 (!b FACT (!c SUB1 *N *c) *b) *F) is executed
(4) (FACT (!c SUB1 3 *c) *b) is executed
(5) (SUB1 3 *c) is executed
(6) *c = 2
(7) ....
(8) *b = factorial(2) = 2
(9) *F = 6

For more convincing usage of the executable pattern, see the description of SELECT.

Notes on "'" and "!": ' and ! are something like Lisp's QUOTE and EVAL. ' prevents ! from being executed.

```
(ASSERT (P (! ADD1 1 *)))
```

asserts

```
(P 2)
```

To prevent ! from being executed at the time of the assertion, it must be quoted as:

```
(ASSERT (P '(! ADD1 1 *)))
```

Similarly,

```
(ASSERT (P 'A))
```

is asserted as:

```
(P A)
```

rather than:

```
(P 'A)
```

To assert the latter, ' must be doubled:

```
(ASSERT (P ''A))
```

# INTERPRETER

## TOP LEVEL LOOP

At the top level of the Prolog/KR system, the interpreter is waiting for the user's input. If a list is typed in, it is interpreted (executed) and, usually, the list is echoed back with variables instantiated according to the result of the execution. For example, suppose

```
(PLUS 1 2 *)
```

is typed in, then

```
(PLUS 1 2 3)
```

is printed back by the interpreter. If the execution has failed, "NIL" is printed instead. For example,

```
(PLUS 1 2 5)
```

results in

```
NIL
```

Since all inputs are executed, even the definitions of predicates must be provided in the form of predicate calls (see the section *Defining and Modifying Predicates*). For example, FACT may be defined as:

```
(ASSERT (FACT 0 1))
(ASSERT (FACT *N *F)
        (SUB1 *N *N1) (FACT *N1 *F1)
        (TIMES *N *F1 *F))
```

In this case, the interpreter acknowledges simply with "ASSERTED" instead of printing the whole assertion back.

For typing conventions, extra right parentheses are ignored. Super parentheses "[" and "]" are also available:

```
(PRINT (A (B]
(PRINT [A (B])
```

The predicates are as follows:

LAST−INPUT *FORM

Returns the last input. LAST−INPUT is defined in the same form as user-defined predicates so that it can be edited and re-executed. Note that the execution must be done in the editor, since once you get out of the editor, the last input becomes "(EDIT LAST−INPUT)".

LAST–RESULT *RESULT

Returns the last result typed out by the interpreter. To see the detail of the result which had been abbreviated to “?”, type:

```
        (GRIND (! LAST-RESULT *))
```

Of course, this does not change LAST–RESULT.

TOP

Transfers the control to the top-level loop.

PROLOG

Initiates the top-level loop. Unlike TOP, which transfers the control to the top level of the current loop, PROLOG initiates a new loop, nested inside the old one.

EPILOG

Exits from a loop initiated by PROLOG and returns to the previous status.

## Executing the OS Command

If a line begins with a symbol, not a list, the line is regarded as an OS command. Any OS command, except command procedures and safe commands, can be called. For example,

```
    :LIST FOO.VDATA(BAR)
```

lists the contents of FOO.VDATA(BAR). “:” in the above example is a prompt of the top level.

Pressing the break key stops the execution of the OS command; the control returns to the top level of Prolog/KR.

## ERROR HANDLING

In the case of errors, the interpreter automatically enters the stepper, where the status of the execution can be examined. Q command or “(TOP)” transfers the control to the top level.

The standard behavior on an error, described above, can be changed by defining a predicate ERROR, which is called on error.

ERROR >message >object

STANDARD–ERROR–HANDLER >message >object

Prints <message> followed by <object>, where <message> is a string and <object> is a Prolog/KR object.

The behavior of the error handler can be defined for each specific error. Since the first argument of ERROR is a message (string), the type of the error can be detected by the argument. For example,

```
(ASSERT (ERROR *MES *OBJ)
    (SELECT *MES
         ("UNDEFINED PREDICATE" (FALSE))
         ( ..... )
         (? (STANDARD-ERROR-HANDLER *MES *OBJ)))
```

Here is a list of messages of errors:

```
"INVALID ARGUMENT TO predicate-name"
"INVARID ARGUMENT LENGTH TO predicate-name"
"ILLEGAL FORM"
"UNDEFINED PREDICATE"
"UNDEFINED VARIABLE"
"ILLEGAL PATTERN"
"NO MORE DATA AREA"
"NO CATCHING STRUCTURE"
```

## ATTENTION HANDLING

To abort the execution of a program, press the break key on the terminal; the interpreter then enters the stepper, where some commands are accepted. To resume execution, issue a command “F”, meaning “finish the execution”. However, to continue exactly from the place of the break is almost hopeless. The very predicate which was about to be called at the time of the break may not be completed properly. If the break key is pressed inside the editor, on the other hand, the control is transferred to the top level (command interpreter) of the editor.

The standard behavior described above can be changed by defining a predicate ATTENTION, which is called when the break key is pressed. The execution suspended by the break is resumed after executing the body of ATTENTION. Again, the very predicate which was about to be called at the time of the break may not be completed properly.

ATTENTION
STANDARD–ATTENTION–HANDLER

Enters the stepper.

## DEFINING AND MODIFYING PREDICATES

The following predicates are available:

ASsert (>predicate-name . >args) [>predicate-call] ...

Adds the assertion to the corresponding definition. Since there may be more than one assertion for one predicate, the word "definition" is used to denote the entire collection of assertions about the predicate. For example,

```
(ASSERT (FALLIBLE *X) (HUMAN *X))
```

ASSERT merges the assertion into a proper (but system-defined) position in the definition. The position is decided so that special cases appear earlier than more general ones. If the automatic ordering is not desired, use ASSERTA, ASSERTZ, or EDIT.

Note: When you make assertions on system-defined predicates, they hide the system's definitions until the assertions are RETRACTed.

AssertA (>predicate-name . >args) [>predicate-call] ...

Just like ASSERT except that it adds the assertion at the top of the corresponding definition.

AssertZ (>predicate-name . >args) [>predicate-call] ...

Just like ASSERT except that it adds the assertion at the end of the corresponding definition.

RETRACT >pat

If <pat> is a symbol, all the assertions about <pat> are retracted.

If <pat> is a list, the first assertion whose heading matches the pattern is retracted. Note that (FOO BAR) matches (FOO *X). Therefore,

```
(RETRACT (FOO BAR))
```

may retract an assertion which begins with (FOO *X). To avoid that case, quote the pattern as:

```
(RETRACT (FOO 'BAR))
```

or

```
(RETRACT '(FOO BAR))
```

Usually, the definitions of predicates are added using ASSERTs. But sometimes it is more convenient to override the previous definitions. In particular, when definitions are loaded from a file, it is rarely intended that loading from the same file twice should amount to double the definitions. The latest loading should override the previous ones.

DEFINE >name [>definition-list] ...

Changes the definition of <name> to <definition-list>s.

The previous definition of <name> is overridden by the new one.

<definition-list> is to be given in the following form:

```
(<arguments> . <body-list>)
```

For example,

```
(DEFINE MEMBER ((*X (*X . *L)))
               ((*X (*Y . *L)) (MEMBER *X *L)))
```

is equivalent to:

```
(RETRACT MEMBER)
(ASSERT (MEMBER *X (*X . *L)))
(ASSERT (MEMBER *X (*Y . *L)) (MEMBER *X *L))
```

Note: (DEFINE name) is equivalent to (RETRACT name).

Calling a predicate which lacks definition is an error. Hence, it is sometimes useful to define a predicate which always fails. As a special usage of DEFINE,

```
(DEFINE name NIL)
```

is allowed to define such a predicate. Of course, the same effect is produced by:

```
(DEFINE name (() (FALSE)))
```

SET >name >value ...

Sets the definition of <name> so that a predicate call (name *X) returns <value>. For example,

```
(SET X 1)
```

is equivalent to

```
(DEFINE X ((1)))
```

or

```
(RETRACT X)
(ASSERT (X 1))
```

DEFINITION >name *definition

Retrieves the definition of <name>. For example,

```
(DEFINE FALLIBLE ((*X) (HUMAN *X))
                 ((*X) (GOD *X)))
(DEFINITION FALLIBLE *DEF)
```

sets *DEF to (((*X) (HUMAN *X)) ((*X) (GOD *X))).

For example, ASSERTA can be defined as:

```
(ASSERT (ASSERTA (*PRED . *ARGS) . *BODY)
        (DEFINITION *PRED *DEFS)
        (DEFINE *PRED
            ((*ARGS . *BODY) . *DEFS)))
```

LISting >name ...

Prints the definition of <name> as (ASSERT ... )... Notes on the definition form: Definition of a predicate appears in different forms in ASSERT(A/Z), DEFINE, DEFINITION, and WITH. Internally, it is stored in the form of DEFINE:

```
((<header1> . <body-list1>)
 (<header2> . <body-list2>) ... )
```

For example, 'append' is stored as:

```
((((() *X *X))
 (((*A . *X) *Y (*A . *Z))
  (APPEND *X *Y *Z)))
```

If we call it “definition-body”, each form is expressed as:

```
(DEFINE <name> . <definition-body>)
(DEFINITION <name> <definition-body>)
(WITH ((<name> . <definition-body>) ... ) ... )
```

# CONTROLS

Although Prolog/KR's default control strategy is the depth first trial with backtracking, users can control the execution using control predicates provided by the system. Control predicates with no backtracking are called deterministic control predicates. If all the controls are provided using deterministic control predicates, the system runs without any backtrackings—like Lisp.

## SUCCESS AND FAILURE

Prolog's control depends on "success" and "failure". A predicate call succeeds if and only if the following conditions are all satisfied:

(1) There exists a definition (either system-defined or user-defined) for the predicate. Otherwise, it is an error.
(2) The pattern of the caller matches the pattern of the callee. If they do not match, the behavior is determined by whether the callee is system-defined or not: if the callee is system-defined, it is an error; otherwise, it is a failure.
(3) If the callee is the user-defined predicate, all the predicate calls in the body of the callee succeed. If it is a system-defined predicate, the behavior is described in this manual.

On failure, the system backtracks and tries alternatives if any (deterministic control predicates leave no alternatives). If there are no alternatives, the entire execution fails.

FALSE

Always fails and triggers backtracking. For example,

```
(AND (MEMBER *X (A B C))
     (PRINT *X)
     (FALSE))
```

produces a result:

```
A
B
C
NIL
```

where "NIL" indicates the failure of AND.

FAIL

Causes a failure of the caller of the predicate which contains FAIL in its body. FAIL is to be used to express a negative information. In other words, if FAIL is encountered while executing a body of a predicate P, then P must be false. For example, NOT may be defined as:

```
(ASSERT (NOT *PRED) *PRED (FAIL))
(ASSERT (NOT ?))
```

NOT >predicate

Inverts the success and failure of <predicate>. Therefore, NOT succeeds when <predicate> fails and fails when <predicate> succeeds. Note that NOT does not instantiate any variable because at least one failure occurs whether in <predicate> or at NOT itself.

NOT is to be used only to test rather to return some result.

```
(NOT (P *X))
```

means

```
For all, *X (P *X) is false.
```

rather than

```
Exists *X such that (P *X) is false.
```

Therefore, for example,

```
(NOT (= *X A))
```

fails while

```
(AND (= *X B) (NOT (= *X A)))
```

succeeds.

TRUE

Immediately succeeds, but only once—TRUE does not succeed in the case of backtracking. Therefore, TRUE and FALSE cannot be used to construct a loop:

```
(AND (TRUE) (PRINT DOING) (FALSE))
```

is executed only once.

ONBT >predicate-call

Does nothing during the forward execution. When backtracking propagates up to ONBT, on the other hand, <predicate-call> is executed; backtracking continues regardless of the result.

Note that (ONBT (TRUE)) has no effect at all because it does nothing even in the case of backtracking.

CUT

Throws backtracking alternatives away up to a certain point (up to the caller of the predicate which contains CUT in its body).

Note: It is suggested by the implementor that this predicate not be used. CUT is provided only for compatibility with other Prolog systems.

## NON-DETERMINISTIC CONTROLS

Since non-deterministic (this word is used as a synonym of “with backtracking” in this manual) control is the default strategy used in Prolog, all the control predicates described in this section may be eliminated by defining extra predicates. Nevertheless, they are provided for the sake of simplicity and efficiency.

AND [>predicate-call] ...

Executes <predicate-call>s from left to right with backtracking in case of failures until all of them (and the rest of the execution) succeed. This is also the default mode of executing a body of a predicate. Thus the following two assertions have exactly the same meaning.

```
(ASSERT (INTERSECT *X *Y *E)
        (MEMBER *E *X) (MEMBER *E *Y))
(ASSERT (INTERSECT *X *Y *E)
        (AND (MEMBER *E *X) (MEMBER *E *Y)))
```

In the above example, INTERSECT returns elements of the intersection of X and Y one by one until INTERSECT finally fails. For example,

```
(AND (INTERSECT (A B C) (B C D) *E)
     (PRINT *E)
     (FALSE))
```

results in

```
B
C
```

Note that “(AND)” immediately succeeds.

OR [>predicate-call] ...

Executes <predicate-call>s from left to right until one of them succeeds. In case of later backtracking, the alternatives of the <predicate-call> which succeeded most recently are tried. If they all fail, the next <predicate-call>, if any, is tried. If no <predicate-call> remains, OR fails.

OR can be eliminated by providing an extra predicate. For example,

```
(ASSERT (P *X) (OR (Q *X) (R *X)))
```

is equivalent to:

```
(ASSERT (P *X) (Q-OR-R *X))
(ASSERT (Q-OR-R *X) (Q *X))
(ASSERT (Q-OR-R *X) (R *X))
```

Note that “(OR)” fails.

## DETERMINISTIC CONTROLS

Control predicates described in this section somewhat limit the scope of backtracking.

IF >predicate-call >then-part [>else-part]

If <predicate-call> succeeds, <then-part> is executed. Otherwise, <else-part> is executed; if <else-part> is omitted, (TRUE) is assumed and IF succeeds. Once <predicate> succeeds, it will not be backtracked later, and <else-part> will not be executed. For example,

```
(ASSERT (MAX *X *Y *M)
        (IF (> *X *Y) (= *M *X) (= *M *Y)))
```

is equivalent to:

```
(ASSERT (MAX *X *Y *X) (> *X *Y))
(ASSERT (MAX *X *Y *Y) (<= *X *Y))
```

Note that the above example is not equivalent to:

```
(ASSERT (MAX *X *Y *X) (> *X *Y))
(ASSERT (MAX *X *Y *Y))
```

DO [>predicate-call] ...

Executes <predicate-call>s regardless of their success or failure. In other words, arguments of DO are executed one by one and failing to execute one of them does not cause backtracking; the execution proceeds to the next one. For example, the result of

```
(DO (MEMBER *X (1 2 3))
    (PRINT *X)
    (FALSE)
    (PRINT NEXT))
```

is

```
1
NEXT
(DO (MEMBER *X ?) (PRINT ?) ? . ?)
```

DAND [>predicate-call] ...

Executes <predicate-call>s one by one until all of them succeed. If one of them fails, DAND fails without backtracking. For example,

```
(DAND (OR (PRINT A) (PRINT B)) (FALSE))
```

fails after printing only A.

```
(DAND p q r)
```

is equivalent to:

```
(IF p (IF q (IF r (TRUE) (FALSE)) (FALSE)) (FALSE))
```

DOR [>predicate-call] ...

Executes <predicate-call>s one by one until one of them succeeds. No alternatives are tried on later backtracking; DOR fails then. For example,

```
(DOR a b c)
```

is equivalent to:

```
(IF a (TRUE)
    (IF b (TRUE)
        (IF c (TRUE) (FALSE))))
```

COND [(>predicate . >body)] ...

Tries <predicate>s from left to right until one of them succeeds, and then the corresponding <body> is executed. Although backtracking may occur during the execution of <body>, it does not propagate to <predicate> or other alternatives; if <body> fails, COND fails. Once <body> succeeds COND succeeds and no more alternatives are tried on later backtracking. For example,

```
(COND ((AND (PRINT A) (FALSE)) (PRINT FIRST-CHOICE))
      ((PRINT B) (PRINT SECOND-CHOICE) (FALSE))
      ((PRINT THIS-WONT-BE-EXECUTED)))
```

results in

```
A
B
SECOND-CHOICE
NIL
```

RETURN [>predicate-call] ...

Fixes the choice of the alternatives of predicate to <predicate-call>s. This predicate is to be used to simulate a kind of macro. For example:

```
(ASSERT (>= *X *Y)
        (RETURN (OR (> *X *Y) (= *X *Y))))
```

is used to achieve equivalence between

```
(AND ... (>= *X *Y) ...)
```

and

```
(AND ... (OR (> *X *Y) (= *X *Y)) ...)
```

regarding backtracking. Even though there may be another assertion about >=, no alternatives on >= are tried in case of backtracking.

A more convincing but more complex example is defining DOR using RETURN (the semantics of DOR has been given earlier in this section):

```
(ASSERT (DOR *P . *REST)
        *P
```

```
                (RETURN))
        (ASSERT (DOR *P . *REST) (RETURN (DOR . *REST)))
        (ASSERT (DOR) (RETURN (FALSE)))
```

The second and the third RETURN are redundant in this example. A pattern which matches the second assertion never matches the third one. RETURN executed in the last alternative has no effect.

Note: Do not use RETURN or FAIL in RETURN as:

```
        (RETURN (PRINT RETURNING-TWICE) (RETURN ...))
```

It may return or fail too many levels.

SELECT *pattern (*pattern1 . >body1) ...

Tries to match <pattern> to <pattern1>, <pattern2>, etc., until one of them can be matched. Like COND, no alternative patterns are searched on backtracking. A pattern corresponding to “(TRUE)” in COND is, of course, “?”. For example, a deterministic version of APPEND can be defined as:

```
        (ASSERT (APPEND . *A)
                (SELECT *A
                    ((() *X *X))
                    (((*B . *X) *Y (*B . *Z))
                    (APPEND *X *Y *Z))))
```

In another example, branching on the input can be expressed as:

```
        (AND (READ *INPUT)
             (SELECT *INPUT
                 (YES (DO-YES))
                 (NO  (DO-NO))
                 (?   (HESITATE))))
```

Note that “?” matches anything.

If some of the branches are the same, an executable pattern is convenient to use:

```
        (SELECT *INPUT
            ((! MEMBER * (YES Y)) (DOING YES))
            ((! MEMBER * (NO N))  (DOING NO))
            (? (HESITATE)))
```

rather than:

```
        (SELECT *INPUT
            (YES (DOING YES))
            (Y   (DOING YES))
            (NO  (DOING NO))
            (N   (DOING NO))
            (?   (HESITATE)))
```

## REPETITION

FOR–ALL >predicate-call >body ...

Tries <body>s for each alternative of <predicate-call>. FOR–ALL succeeds if, and only if,

```
<predicate-call> logically implies <body>s.
```

In other words, FOR–ALL tests whether all solutions of <predicate-call> also satisfy <body>. No variables are instantiated inside FOR–ALL. For example:

```
(AND (FOR-ALL (MEMBER *X (A B C)) (PRINT *X))
     (VAR *X))
```

succeeds after printing:

```
A
B
C
```

Note that (VAR *X) succeeds, because *X is left uninstantiated.

When <body>s fail, FOR–ALL immediately fails without trying the next alternative of <predicate>. For example,

```
(FOR-ALL (MEMBER *X (A B C)) (PRINT *X) (FALSE))
```

results in

```
A
NIL
```

CANDIDATES *variable >predicate-call ...

Instantiates <variable> to a list of all the values of <variable> which satisfy <predicate-call>s. <variable> must be an uninstantiated variable on entrance. For example, provided that EVEN–NUMBER succeeds only when its argument is an even number,

```
(CANDIDATES *X (MEMBER *X 1 2 3 4 5)
    (EVEN-NUMBER *X))
```

sets *X to (2 4). In another example, INTERSECTION and UNION may be defined as follows:

```
(ASSERT (INTERSECTION *X *Y *RESULT)
    (CANDIDATES *RESULT
        (MEMBER *RESULT *X)
        (MEMBER *RESULT *Y)))
(ASSERT (UNION *X *Y *RESULT)
    (CANDIDATES *RESULT
        (OR (MEMBER *RESULT *X)
            (MEMBER *RESULT *Y))))
```

Note that (INTERSECTION (A B) (B C) (B)) fails under the above definition, because, for example, (MEMBER (B) (A B)) fails. CANDIDATES must be used to produce a result, not to test the result.

ACCUMULATE *structure >predicate *variable

Instantiates <variable> to a list of <structure>, the value of which is determined by each alternative of <predicate>. ACCUMULATE is a generalized version of CANDIDATES. In fact, CANDIDATES can be defined as:

```
(ASSERT (CANDIDATES *X *P)
    (ACCUMULATE *X *P *X))
```

For example, here is a new definition of INTERSECTION according to which (INTERSECTION (A B) (B C) (B)) succeeds.

```
(ASSERT (INTERSECTION *X *Y *RESULT)
    (ACCUMULATE *TEMP
         (AND (MEMBER *TEMP *X)
              (MEMBER *TEMP *Y))
        *RESULT))
```

Note: On the current implementation, some predicates cannot be used inside the predicates described so far in this section. They are NOT, DOR, COND, POR, RECORD, and ERASE. This rule is not applicable to the following predicates.

LOOP [predicate-call] ...

Repeats <predicate-call>s until EXIT is executed. “while-do” and “repeat-until” are considered to be special cases of LOOP. For example:

```
(ASSERT (WHILE-DO *P *BODY)
    (LOOP (IF *P (TRUE) (EXIT)) *BODY))
(ASSERT (REPEAT-UNTIL *BODY *P)
    (LOOP *BODY (IF *P (EXIT))))
```

EXIT

Exits from the innermost LOOP.

## PSEUDO-PARALLELISM

A limited version of pseudo-parallel computation is supported in current implementation.

POR >predicate-call ...

Executes <predicate-call>s in a pseudo parallel mode. Usually, each <predicate-call> is executed one step (one predicate invocation) and then another one is executed. As the name suggests, there is no interaction between <predicate-call>s. They are executed independently even if they share the same variables; those variables are to be thought of as different ones.

When one of the <predicate-call>s succeeds, POR succeeds once and for all. No alternatives are tried on later backtracking.

If a control predicate (except AND, OR, and NOT), RECORDED, or ERASE is called within POR, the call is completed before other <predicate-call>s are executed. For example,

```
(AND (POR (AND (PRINT A) (PRINT B))
     (OR (PRINT OR-1) (PRINT OR-2))
         (PRINT C))
     (FALSE))
```

results in

```
A
OR-1
NIL
```

Note that the same rule is applied when POR is called within POR.

## NON-LOCAL EXIT

CATCH >predicate-call

Executes <predicate-call>. If THROW is executed during the execution of <predicate-call>, the control may return to CATCH if the following conditions are satisfied:

(1) <predicate-call> matches the argument of the THROW.
(2) The CATCH is the innermost one which satisfies 1).

In this case, variables in <predicate-call> are instantiated through the pattern matching.

If <predicate-call> terminates normally, either with success or failure, CATCH terminates with the result.

THROW >pattern

Transfers the control up to the innermost CATCH whose argument matches <pattern>. For example, here is a definition of DOR using CATCH and THROW.

```
(ASSERT (DOR . *X)
    (CATCH (FOR-ALL (MEMBER *PRED *X)
        (IF *PRED (THROW (FOR-ALL . ?))))))
```

## CO-ROUTINE

In Prolog/KR, co-routines are divided into a producer and a consumer. A producer is a normal Prolog/KR predicate which is supposed to succeed with more than one result. For example,

```
(MEMBER *X (A B C))
```

can be used as a producer which returns *X=A, *X=B, and *X=C. A consumer, on the other hand, does special things: initiates the producer and then gets results from it.

INITIATE >predicate-call *name

Creates a producer with <predicate-call> and names the producer uniquely. The unique name is returned to <name>. For example:

```
INITIATE (MEMBER *X (A B C)) *M)
```

NEXT >producer . >pattern

Gets one result from <producer>. To be more precise, the producer is forced to backtrack to another result, the pattern which was given as the first argument of INITIATE is instantiated with the result, and finally the pattern is matched with <pattern>.

NEXT does not produce another result on backtracking; it fails instead. The fact that the producer is called is not undone on backtracking either. Therefore, if the same NEXT is entered more than once (after failing once or more), it produces the next result. For example, the producer which was initiated by:

```
(INITIATE (MEMBER *X (A B C)) *M)
```

can be called by:

```
(NEXT *M *Y (A B C))
```

or

```
(NEXT *M *Y ?)
```

In both cases, *Y is instantiated to A on the first call; to B on the second call, and so on.

As you can see by the example, the last argument of NEXT, “(A B C)”, is not necessary for normal uses. Hence, for the convenience of the programmer, <pattern> may be shorter than the actual pattern. In that case, the rest of the pattern of the producer is ignored. The previous producer can thus be called also by:

```
(NEXT *M *Y)
```

# DATABASE FACILITIES

In Prolog, predicate definitions can be regarded as a collection of assertions to a database. Prolog/KR supports facilities to construct a kind of a hierarchical database.

## INTERNAL DATABASE

Prolog/KR provides an indexed internal database which is completely separated from predicate definitions.

Since the internal database is indexed by its contents (only by symbols and numbers), it guarantees quick retrieval of the contents. This is the main difference from the top-level database which consists of predicate definitions; the top-level database is indexed only by predicate names.

RECORD >pattern

Adds <pattern> into the internal database. <pattern> must be a list whose first element is a symbol (other elements may be any objects including variables).

IMPORTANT: The effect of RECORD is undone in case of backtracking.

RECORDED >pattern

Retrieves one element in the internal database which matches <pattern>. In case of backtracking, the next one is retrieved. The order of the retrieval is undefined.

RECORDING >pattern >daemon

Creates a daemon which is activated when something which matches <pattern> is RECORDed. The daemon vanishes after one activation. For example,

```
(RECORDING (HUMAN TURING)
    (PRINC "Turing is a human.")
    (TERPRI)
    (RECORD (FALLIBLE TURING)))
(RECORDING (FALLIBLE TURING)
    (PRINC "Turing is fallible."))
(RECORD (HUMAN TURING))
```

results in

```
Turing is a human.
Turing is fallible.
```

If there already exists a datum which matches <pattern>, the daemon is immediately activated.

ERASE >pattern

Erases one datum which matches <pattern>. Another one will be erased in case of backtracking.

## WORLDS

Prolog/KR consists of at least two partitions of definition spaces; one of them consists of system-defined predicates and the other is the default space for the users (called STANDARD–WORLD). A user can construct more worlds explicitly or implicitly. Predicates defined within other worlds, except parent worlds, cannot be called directly. They must be called as:

```
(WITH name-of-the-world (predicate .... ))
```

The following picture gives a rough idea of relations between worlds.

```
                -----------            -----------
                | WORLD-A  |           | WORLD-A  |
              -----------------      ---------------
              | STANDARD-WORLD |     |   WORLD-B    |
          --------------------------------------------
          |    WORLD-OF-SYSTEM-DEFINED-PREDICATES      |
      ------------------------------------------------------
```

A world can be created explicitly (CREATE–WORLD) or implicitly (WITH), referred to (WITH), saved (DUMP), loaded from a file (LOAD–WORLD) and destroyed (ERASE–WORLD). Suppose, for example, worlds A and B contain the following definitions:

A contains definitions of PA1, PA2
B contains definitions of PB1, PB2

And suppose that the current world is A, then PA1 and PA2 can be called directly, but PB1 and PB2 must be called as follows:

```
(WITH B (PB1 ... ))
(WITH B (PB2 ... ) (PA1 ...))
```

Note that PA1 is visible even when the world B is opened. In other words, an inner world inherits predicate definitions from outer worlds. The inheritance is dynamically determined unlike other programming languages or knowledge representation systems.

WITH >world [>body] ...

If <world> is a name of a world (i.e., a symbol), WITH executes <body>s within the world. If the name of the world is a new one, a new world is created.

If <world> is a list of predicate definitions, those definitions become visible only while <body>s are being executed. Once the system exits from the world, those definitions are not only invisible but also destroyed.

Predicate definitions given outside of WITH are effective only if they are not overridden by <world>.

Here is an example in which WITH is used to simulate global variables:

```
(WITH ((X) (Y ((1))))
     ...
     (SET X A)      ; sets the value of X to A
     (X *X)         ; *X becomes A
     (Y *Y)         ; *Y becomes 1
     ...)
```

CREATE–WORLD >name [>predicate-definition] ...

Creates a new world called <name>, which initially contains predicate definitions given by<predicate-definition>s. For example,

```
(CREATE-WORLD PP
    (PRIN1 ((*OBJ) (PRINC *OBJ)))
    (PRINT ((*OBJ) (GRIND *OBJ))))
```

creates a new world called PP, in which PRINT is equivalent to GRIND except that escape character is not supplied. Therefore,

```
(WITH PP (PRINT /-/ A/ -))
```

prints

```
- A -
```

while both

```
(PRINT /-/ A/ -)
```

and

```
(GRIND /-/ A/ -)
```

print

```
/-/ A/ -
```

ERASE–WORLD >name

Destroys the world <name> so that it no longer contains definitions.

LOAD–WORLD >file-name >name

Loads definitions from <file-name> into a world <name>. Since there is no notion of a world once definitions are dumped into a file (cf. DUMP), the name of the world must be given explicitly when it is loaded from a file.

WORLD–NAME *name

Returns the name of the current world.

# DEBUGGING AIDS

Prolog/KR has two modes: debugging mode and non-debugging mode. In debugging mode, some extra information is saved. For example, backtracing is effective only when the interpreter has been running in the debugging mode.

DEBUG *mode

Switches debugging mode according to <mode>: if <mode> is ON, the interpreter runs in debugging mode thereafter; if <mode> is OFF, the interpreter runs in non-debugging mode; if <mode> is a variable, the current mode (ON or OFF) is returned.

BACKTRACE *list

Returns the history of the execution (including backtracking) to <list>.

## TRACER

TRACE [>name] ...

Enables tracing of <name>s. The output of the tracer is in the following form:

```
level (predicate-call)
```

or

```
level= (predicate-call)
```

The former is printed when a predicate call is tried; the latter is printed when it succeeds.

TRACE–ALL

Enables tracing of all predicates.

UNTRACE [>name] ...

Disables tracing of <name>s.

UNTRACE–ALL

Disables tracing of all predicates.

## STEPPER

The stepper is called via STEP.

STEP >predicate-call [>name] ...

Execute <predicate-call> in a step-by-step manner. On each step, the stepper requests for command(s) to determine the next action.

If <name>s are supplied, the stepper stops only at the entrances of <name>s.

The stepper commands are:

C

Continue step-by-step execution.

F

Finish step-by-step mode and resume normal execution.

G

Execute the body in normal mode and return to step-by-step mode afterwards.

Q

Quit stepping and return to the top level.

BackTrace

Backtrace the history of the execution. BT prints the outlines, while BACKTRACE prints with details. Debug mode must be ON to use this command.

PP

Prettyprint the current goal.

list

Execute the list.

## EDITOR

The structure-oriented editor Amuse is available in Prolog/KR. To edit a predicate definition or a file, do

```
(EDIT predicate-name)
```

or

```
(EDIT "file-name")
```

respectively.

You may also execute some editor commands on entrance by:

```
(EDIT name commands ...)
```

You can use this feature to define some function to manipulate function definitions.

EDIT >name [([>editor-command] [>argument] ...)]

> Edits <name>. If <name> is a symbol, the definition of <name> is edited. If <name> is a string, on the other hand, a file with that name is edited. In that case, the top-level scope is a list of the contents of the file. All the commands are the same in both cases.

## Scope

The editor moves around the structure by shifting its attention. The range of the current attention is called "scope". You can narrow the scope by moving down in the structure, or widen it by moving up.

You can print the current scope by P command. If you use V command on the other hand, a list which includes the current scope as its top-level element is shown. Suppose you are at the first pair of COND list:

```
(COND ((ATOM X) Y)
      (T (CONS (CAR X) (APPEND (CDR X) Y))))
```

P prints

```
((ATOM X) Y)
```

while V prints

```
(COND $$$ (T (CONS ? ?)))
```

Note: If "$$$" appears in a list used as an example in this section, the list is supposed to be showing one upper level of the current scope.

## The Stack

The editor has one stack to save various elements. If you kill or copy an element, it is pushed into the stack. The contents of the stack may be shown by the STACK command. If you do not need the topmost element of the stack, it may be removed by POP command. Elements of the stack can be used as arguments to a command, such as "F" and "R". An integer at the argument position designates the nth argument from the top of the stack. For example:

```
I 1
```

inserts the topmost element on the stack just after the current scope.

Caution: If you want to give an integer itself as an argument, precede it with a "'", e.g., '3.

## Pattern Matching and Variables

Patterns may appear as arguments to F(find) or RA(replace all), etc. Unlike the top level, an atom which begins with a character "&" is regarded as a variable in the pattern. It can match any objects. For example, (&A &A . &B) matches any of

```
(A A)
(A A X Y Z)
```

```
(XXX XXX XXX)
```

but not:

```
(A *X)
(A B C)
(X)
(&1 &2)
```

Note that Prolog/KR variables are not treated as variables inside the editor. Note also that the pattern matching of the editor is one way.

If variables appear in the arguments of RA command, the values are held until each replacement completes. You can thus, for example, change the order of arguments of all the function calls of "F" by:

```
RA (F &1 &2) (F &2 &1)
```

The above command changes

```
(AND (F *X *Y) (F (F A B) (G 1 2)))
```

to

```
(AND (F *Y *X) (F (G 1 2) (F A B)))
```

Note that (F A B) has not been changed, because RA is applied to the outermost pattern.

The first argument of "G" can be quoted by:

```
RA (G &1 . &2) (G '&1 . &2)
```

## How to Give Commands

Basically, there are two ways to give commands to the editor.

One, usually the editor, reads commands from the terminal with a prompt "E:". The user may input any number of commands (each of them may followed by some arguments) and they are executed at once.

Two, they are given as the second argument to EDIT as:

```
(EDIT FOO (PP 1 I (SOMETHING)))
```

which first prints the top level and inserts "(SOMETHING)" after the first element before accepting further commands from the terminal.

## Basic Commands

Each command for the editor consists of one or more character(s), possibly with argument(s). Basic commands are defined as Prolog/KR predicates. The name of the predicate begins with EC: (standing for Editor Command) followed by the command name. For example, an insert command is

```
(EC:IN '(A B C))
```

Note that arguments of a command are evaluated in Utilisp.

Basic commands also have an abbreviated form. They can be given from a terminal without parentheses or “EC:”. For example, the above insert command may also be invoked by:

```
IN (A B C)
```

Arguments are not evaluated this time.

Commands are described in the following:

B
(EC:B)

Moves to the previous element (the opposite of N). For example, if the current scope is

```
(A B $$$ D)
```

“B” changes the current scope to B:

```
(A $$$ C D)
```

BI number1 number2
(EC:BI number1 number2)

Encloses elements (from <number1>th to <number2>th) in parentheses. For example:

```
BI 2 3
```

changes

```
(A B C D E F G)
```

to

```
(A (B C) D E F G)
```

BO
(EC:BO)

Removes the parentheses enclosing the current scope (the opposite of BI).

C
(EC:C)

Copies the current scope to the top of the stack.

D

Deletes the current scope. The new scope is the next element which followed the deleted one. If none follow, i.e., the deleted element was the last element of the list, the scope goes up one level.

E name
(EC:E name)

Abandons the result of the current editing and shifts the object of editing to <name>.

F pattern
(EC:F pattern)

Moves to an element which matches <pattern>. If no proper element is found, EC:F returns NIL in Utilisp and fails in Prolog/KR. If F had been used, on the other hand, an message indicating the failure is printed and any commands following the same line are ignored.

FN
(EC:FN)

Moves to the next element which matches the pattern most recently given to "F" command. If no proper element is found, EC:FN returns NIL in Utilisp and fails in Prolog/KR. If FN had been used, on the other hand, an message indicating the failure is printed and any commands following the same line are ignored.

I element
IN element
(EC:IN element)

Inserts <element> just after the current element. The scope is shifted to the inserted element.

IB element
(EC:IB element)

Inserts <element> just before the current element. The scope is not changed.

K
(EC:K)

Kills the current element and pushes it on top of the stack. The new scope is the next element which followed the killed one. If none follow, i.e., the killed element was the last element of the list, the scope goes up one level.

L
(EC:L)

Moves to the last element of the current scope.

LD
(EC:LD)

Executes all the elements of the topmost scope. This command is intended to be used while editing a file to reload changed definitions.

LI
(EC:LI)

Inserts a left parenthesis just before the current element. The scope remains the same. For example, if the current scope is

```
(A $$$ C)
```

LI changes it to:

```
(A ($$$ C))
```

LEVEL number
(EC:LEVEL number)

Resets the printing depth to <number>. The default depth is 7.

N
(EC:N)

Moves to the next element. If the current element is the last one, an error is issued.

P
(EC:P)

Prints the current element without details. Details are printed by "?". The level of the printing is controlled by LEVEL.

PP
(EC:PP)

Prints the current element with details.

POP
O
(EC:POP)

Pops the stack.

Q
(EC:Q)

Restores the result of the editing and exits from the editor.

R element
(EC:R element)

Replaces the current scope by <element>.

RA pattern1 pattern2
(EC:RA pattern1 pattern2 [maximum-number-of-replacement])

Replaces all the elements of the current scope (including the scope itself) which match <pattern1> by <pattern2>. <pattern1> is matched with the element on each replacement. Hence, variables appearing in <pattern2> are replaced properly. The scope is not changed by the command.

If <maximum-number-of-replacement> is supplied, only the first <maximum-number-of-replacement> patterns are replaced. For example, suppose that the current scope is

```
(P (T A) (T B) (T2 C))
```

and you issue “RA (T &X) (TT &X)”, the result is

```
(P (TT A) (TT B) (T2 C))
```

R1, R2, R3

Are all the same as “RA” except that they change only first one, two, or three elements, respectively.

RI
(EC:RI)

Inserts a right parenthesis just after the current scope. For example, if the current scope is

```
(A $$$ C)
```

RI changes it to:

```
((A $$$) C)
```

S
(EC:S)

Restores the current result without exiting from the editor.

SC name
(EC:SC name)

Stores the current result of editing to <name>. When a file is being edited, <name> must be a name of an existing file, expressed as a string.

STACK
(EC:STACK)
ST
(EC:ST)

Prints the contents of the stack.

TOP
(EC:TOP)

Moves to the top level of the definition, which is normally a list of clauses.

U symbol
(EC:U symbol)

Goes up in the structure to a structure which begins with <symbol>. For example,

```
    U DEFINE
```

goes up to a structure (DEFINE ...).

V
(EC:V)

Shows the current position by printing the upper level. The current scope is printed as "$$$".

VAR character
(EC:VAR character)

Changes the variable prefix to <character>. The original variable prefix is "&".

X predicate-call
(EC:X predicate-call)

Executes <predicate-call> and prints the result. Like other commands, elements in the stack can be given as <predicate-call> by specifying a number. The number 0 is treated specially by X; it designates the current scope. This feature is useful when a file is being edited. Since changing the contents of a file does not affect the loaded definitions, the changed definitions must be reloaded. The most simple way is to shift the scope to (DEFINE ...) and execute the command "X 0".

Z
(EC:Z)

Abandons editing and exits. The original definition or contents of the file are not changed.

?
(EC:?)

Prints the name of the predicate/file which is being edited.

number
(E:MOVE number)

Moves to the <number>th element of the current scope. If the number is 0, the scope is shifted to the upper level. And if the number is negative, the elements are counted from the end of the list. For example, here is an example list with commands to move to its elements listed below:

```
(FOO BAR POO ZOO TONG)
  |   |    |   |    |
  1   2    3   4    5
 -5  -4   -3  -2   -1
                    L
```

If the corresponding element is lacking (because the list is too short), or (E:MOVE 0) is attempted at the top level, E:MOVE returns NIL in Utilisp and fails in Prolog/KR.

(number . commands)

Repeats <commands> for <number> times. If <number> is not positive, nothing is executed.

## Editor Macros

Macro commands can be defined using the full power of Prolog/KR. Following is an example definition of a predicate to print every expression matching the pattern (given as the argument).

```
(ASSERT (PRINT-ALL *PAT)
    (EC:F *PAT) (EC:PP)
    (LOOP (IF (EC:FN) (EC:PP) (EXIT))))
```

In the above example, (EC:PP) is repeated until (EC:FN) finally fails.

The command defined above must be called as

```
(PRINT-ALL)
```

To allow the user to call the command without parentheses,

```
(E:DEFCOM PRINT-ALL (PRINT-ALL))
```

may be done.

E:DEFCOM >command-name [>calling-sequence] ...

Defines <command-name> to be the same as actually calling predicates in the form <calling-sequence>.

Some extra predicates are defined to support macros.

E:SCOPE <scope

Returns the current scope.

E:STACK <stack

Returns a list of all the contents of the stack.

E:READ >prompt <object

Gets one argument with prompt <prompt>. If a number is read, the corresponding element in the stack is returned.

E:GETFILE >filename <contents

Reads the contents of the file and returns a list of them.

E:PUTFILE >filename >contents

Dumps <contents> to a file named <filename>. <contents> must be a list of objects to be dumped.

# INPUTS AND OUTPUTS

## FUNDAMENTAL I/O PREDICATES

I/O is done through “streams” whose default values are assigned to the terminal. READ and PRINT interacts with the terminal unless otherwise specified. To access a file, a stream corresponding to the file must be created using INOPEN or OUTOPEN according to whether the stream is input or output, respectively.

### Inputs

READ *result [>stream]

Reads one object from <stream>. If <stream> is omitted, STANDARD–INPUT is assumed (see the description about STANDARD–INPUT below).

If <result> is not a variable, the read object is compared with <result>; if they do not match, READ fails. The read object will not be put back to the input stream on backtracking. Nor are any alternatives read; READ simply fails on backtracking. This feature applies all the I/O predicates.

RIND [>prompt] *result

Reads one object from the terminal prompting with proper indentations. This is also the way the top-level loop reads. Super parentheses, “[” and “]”, are also available in this mode.

STANDARD–INPUT *stream

If <stream> is a variable, the current STANDARD–INPUT is returned. If <stream> is a input stream, the STANDARD–INPUT is reset to <stream>. A special symbol TERMINAL–INPUT is recognized by this predicate: After doing (STANDARD–INPUT TERMINAL–INPUT), the STANDARD–INPUT is reset to the terminal.

You may also ASSERT or DEFINE STANDARD–INPUT:

```
(STANDARD-INPUT <stream>)
```

and

```
(ASSERT (STANDARD-INPUT <stream>))
```

have almost the same effect except that the latter may restore the previous STANDARD–INPUT after (RETRACT STANDARD–INPUT) but the former does not. Asserting STANDARD–INPUT is recommended to be used with WITH as:

```
(WITH ((STANDARD-INPUT ((<stream>))))
      (do some inputs))
```

which does not leave any change on STANDARD–INPUT.

TERMINAL–INPUT <stream

Returns an input stream which is connected to your terminal. Note that this predicate is output only, unlike STANDARD–INPUT.

PROMPT <character

Returns the default prompt character. To change the default prompt character, you must redefine PROMPT (by either ASSERT, DEFINE, or WITH).

## Outputs

PRINT >object [>stream] [>printlevel]

Prints <object> to <stream> with <printlevel>. The default values of <stream> and <printlevel> are STANDARD–OUTPUT and 3, respectively. Details of <object> whose level exceeds <printlevel> are abbreviated to "?". If <printlevel> is equal to or less than zero, no details are abbreviated.

PRIN1 >object [>stream] [>printlevel]

Just like PRINT except that PRIN1 does not feed line after printing <object>. Note that nothing will be printed until TERPRI is executed.

PRINC >object [>stream] [>printlevel]

Just like PRIN1 except that PRINC does not supply """ or "/". Note that nothing will be printed until TERPRI is executed.

TERPRI [>stream]

Begins a new line.

TAB >indentation [>stream]

Indents the output line to <indentation>. If the current position is over the indentation, a new line with <indentation> is begun.

GRIND >object

"(GRIND object)" is equivalent to "(PRINT object 0)". Note that GRIND always outputs to STANDARD–OUTPUT.

PRINT-LEVEL >level

Sets the default printlevel to <level>.

CASE *case

If <case> is UPPER, the output is printed using uppercase characters (this is the default). If <case> is LOWER, the output characters are lowercase ones. These changes do not effect printing strings (in strings, lower and uppercases are preserved). If <case> is a variable, either UPPER or LOWER is returned according to the status. Note that CASE is used commonly by all output streams (i.e., CASE is not the property of the stream).

LINESIZE *size

If <size> is an integer, the line size of the current output stream (STANDARD–OUTPUT) is set to <size>. If <size> is a variable, the current line size is returned. LINESIZE is a property of each stream.

STANDARD–OUTPUT *stream

If <stream> is a variable, the current STANDARD–OUTPUT is returned. If <stream> is an input stream, the STANDARD–OUTPUT is reset to <stream>. A special symbol TERMINAL–OUTPUT is recognized by this predicate: After doing (STANDARD–OUTPUT TERMINAL–OUTPUT) the STANDARD–OUTPUT is reset to the terminal.

You may also ASSERT or DEFINE STANDARD–OUTPUT:

```
(STANDARD-OUTPUT <stream>)
```

and

```
(ASSERT (STANDARD-OUTPUT <stream>))
```

have almost the same effect except that the latter may restore the previous STANDARD–OUTPUT after (RETRACT STANDARD–OUTPUT) but the former does not. Asserting STANDARD–OUTPUT is recommended to be used with WITH (see the description about STANDARD–INPUT above).

TERMINAL–OUTPUT <stream

Returns an input stream which is connected to your terminal. Note that this predicate is output only, unlike STANDARD–OUTPUT.

## FILE MANIPULATIONS

INOPEN >file-name <stream

Creates an input stream by opening a file <file-name>. <file-name> may be a closed stream (a stream does not vanish even if it is closed).

OUTOPEN >file-name <stream

Creates an output stream by opening a file <file-name>. <file-name> may be a closed stream (a stream does not vanish even if it is closed).

CLOSE >stream

Closes <stream> which has either been in-opened or out-opened.

NEW–FILE >file

Creates a new file. <file> must be given as a string.

LOAD >file

Executes the contents of <file> and remembers the names of predicates defined in it. The list which contains the contents of <file> is called “loaded-list”. “loaded-list” may be updated using ADD or DEL.

LOADED >file *list

Sets <list> to a list of predicate names loaded from <file> (called “loaded-list”). Note that this does not return the original contents if “loaded-list” is modified by ADD or DEL.

DUMP >file {>definitions}

Dumps <definitions> into <file>. <definitions> must be either a list of predicate names or the name of a world. Particularly, to dump all the definitions into a file,

```
(DUMP <file-name> STANDARD-WORD)
```

is useful.

Something which is not a definition of a predicate — such as a predicate call of printing something — can also be dumped by just adding it, instead of a predicate name which is a symbol, into the list of predicate names. For example, after

```
(DUMP "TEMP" ((PRINT BEGIN) P Q (PRINT END)))
```

the contents of the file TEMP will be

```
(PRINT BEGIN)
(DEFINE P ... )
(DEFINE Q ... )
(PRINT END)
```

STORE >file

Restores definitions of predicates loaded from <file>. Addition or deletion of those definitions to/from <file> is handled by the following two predicates.

Note: Use STORE to store definitions in an existing file. DUMP is designed to create a new file and store definitions in it.

ADD >file [>predicate-name] ...

Adds <predicate-name>s to the sets of predicates loaded from <file> (“loaded-list”). Note that this does not change the actual contents of <file> until STORE is executed.

DEL >file [>predicate-name] ...

Deletes <predicate-name>s from the sets of predicates loaded from <file> (“loaded-list”). Note that this does not change the actual contents of <file> until STORE is executed.

## CALLING LISP FUNCTIONS

All the UTILISP functions can be called from Prolog/KR by either:

(1) using a predicate X
(2) giving one extra argument to receive the value
(3) without the extra argument (those functions are listed later in this section)

Usually, Lisp functions are called directly; the last argument is supposed to be the value of the function. For example a Lisp function REVERSE is called as:

```
    (REVERSE (A B C) *X)
```

There are some Lisp functions for which the values are not important (i.e., only the side effect is important). There are also some Lisp functions which are used as predicates, e.g., >, <, STRING–LESSP; in this case, the value is either T(rue) or NIL(false). Since the result is either not important or necessary only to control the behavior of the program, those functions are called without extra arguments. For those functions, the call succeeds or fails according to whether the result is non-nil or nil, respectively.

Here is a list of Lisp functions to be called without extra arguments:

> >x ...
< >x ...
>= >x ...
<= >x ...
0> >x
0< >x
0= >x
CALL >command [>arguments]
DEFCS >command-symbol >value
GREATERP [>x] ...
LISTP >x
LESSP [>x] ...
MEMQ >element >list
NUMBERP >x
STRING–LESSP >string1 >string2
STRINGP >x
TYO >char [>stream]
VECTORP >object
ZEROP >x

Note: Arguments of the functions are not evaluated. For example,

```
    (CAR (CONS A B) *X)
```

instantiates *X to CONS, rather than A.

There is also a system-defined predicate to call Lisp functions.

X >function-name [>argument] ...

Calls <function-name> with <argument>s. <Argument>s are evaluated.

# THE SECOND-ORDER FEATURES

The second-order features of Prolog/KR have been introduced implicitly in the previous sections. Those are:

(1) Arguments to a predicate themselves may be predicates.
(2) Predicate themselves may not have the fixed interpretations or definitions; they are context dependent.

## VARIABLES AS PREDICATES OR PREDICATE CALLS

User-defined predicates can also have those second-order features mentioned above. This is done through variables; variables may appear as predicates (the first element of the predicate call) or as predicate calls. For example, NOT may be defined as follows.

```
(ASSERT (NOT *PRED)
        (IF *PRED (FAIL)))
```

In the above example, the second appearance of "*PRED" is at the place where a predicate call must be. For example, here is an example in which a variable appears as a predicate name.

```
(ASSERT (MAP *FN (*A . *L) (*R . *REST))
        (*FN *A *R)
        (MAP *FN *L *REST))
```

## MANIPULATING PROGRAMS AS DATA

In writing a compiler of Prolog in Prolog itself, for example, a program must be treated as a datum. In this case, variables contained in the programs to be compiled should not be treated as variables in pattern matching. For example,

```
(ASSERT (COMPILE (*FUNCTION NIL)) ... )
```

should not be called when compiling, say,

```
(FACTORIAL *N)
```

To avoid the matching between *N and NIL, either (a) the variable prefix should be changed so that *N is no longer a variable, or (b) NIL should be quoted as 'NIL (see the section *Extended Features*) to avoid matching to variables.

Provided that the variable prefix is, for example, "=" in compiler, all the usual variables such as "*X" or "*Y" are treated as constants. However, even if the variable prefix has been changed to avoid the problem, the compiler cannot compile itself. The latter solution (quoting) should be used in the compiler.

# MISCELLANEOUS PREDICATES

The following predicates are available:

QUIT
END
EPILOG

Exits from Prolog/KR and returns to the OS level.

LISP

Enters UTILISP top level. (PGO) transfers the control back to Prolog/KR.

MEMBER *element >list

Checks if <element> is a member of the <list>. If <element> is an uninstantiated variable, elements of <list> are returned one by one on each backtracking.

REWRITE >old-file >new-file

Rewrites the contents of <old-file> which is written in the old syntax (Prolog/KR I versions) to <new-file> in the new syntax. The syntax of I versions is very close to that of Marseille Prolog.

TECO

Enters the text editor TECO. If the saved area for OS is too small, calling TECO fails and Prolog/KR is exited. Just press the return key to come back to Prolog/KR.

TIME >predicate-call *time

Sets <time> to the CPU time (in milliseconds) elapsed in executing <predicate-call>.

VERSION

Prints the current version name.

HELP >predicate-name

Prints the information in this manual concerning <predicate-name>. The first 20 lines are printed and “—more—” is printed waiting for your input. Simply press the return key to continue the output (for 20 more lines). Any character followed by the return key or a single break (without any preceding characters) stops the output.

Note: any characters after (HELP ...) on the same line are ignored.

MANUAL [>output]

Prints this manual. If <output> is omitted, the output is printed at your terminal. If <output> is supplied, the output is changed according to <output>:

number

The output is printed at your terminal. But the printing pauses every <output> lines. Hit the return key to continue. Hit break to quit. This is convenient for display terminals.

file-name

The output is printed in a file. The file must exist (to create a file, use NEW–FILE).

LP

The output is printed on a line printer. Use a line printer which has lowercase letters for printing.

# APPENDIX A
# HOW TO ENTER PROLOG/KR

There are basically two ways to invoke Prolog/KR:

(1) Call the function PROLOG in Utilisp.

```
>> UTILISP FIX(40)
> (PROLOG)
```

(2) Use the command procedure PROLOG.

```
>> PROLOG
```

There are several optional parameters to PROLOG.

SAVE(pages)

Area size in pages (one page is 4KB) to be saved for the operating system. This area is not required unless you call the OS commands from inside Prolog/KR. The default value is 32.

STACK(pages)

Area size in pages for the system stack. The default value is 16.

INITIAL(file-name)

A file name, the contents of which are to be executed before reading from the terminal.

HELP

If this option is given, a brief survey of the options is printed instead of invoking Prolog/KR.

For example,

```
>> PROLOG SAVE(0) INITIAL(ANTI.PKR)
```

may invoke your own system ANTI written in Prolog/KR.

If you assign a command symbol PROLOGLIB to an existing file,

```
>> SETCS PROLOGLIB file.name
```

the contents of the file are executed first of all (even before executing the file given as INITIAL parameter). PROLOGLIB is supposed to be linked to a file which contains your own definitions of utility predicates.

# APPENDIX B
# EXAMPLES

Definitions of control predicates are given in Prolog/KR itself to help users understand the language. Since it is impossible to give the definitions of some control predicates, only those which can be defined in terms of other predicates are listed. Note that these definitions have nothing to do with the actual implementation.

```
(ASSERT (AND >PRED . >REST) >PRED (AND >REST))
(ASSERT (AND))
(ASSERT (COND (>PRED . >BODY) . >REST)
    (IF >PRED >BODY (COND . >REST)))
(ASSERT (DOR >PRED . >REST)
    (IF >PRED (TRUE) (DOR . >REST)))
(ASSERT (FALSE) (FAIL 0))
(ASSERT (IF >PRED >THEN . >ELSE)
    >PRED (RETURN >THEN))
(ASSERT (IF >PRED >THEN . >ELSE)
    (RETURN . >ELSE))
(ASSERT (NOT >PRED) (IF >PRED (FAIL)))
(ASSERT (OR >PRED . >REST) >PRED)
(ASSERT (OR >PRED . >REST) (OR . >REST))
```

The following is a definition of MAP which maps a predicate over lists of arguments. For example,

```
(MAP APPEND ((A B) (C D))
     ((0 1) (2 3 4))
     ((A B 0 1) (C D 2 3 4)))
```

holds

```
(ASSERT (NILS NIL))
(ASSERT (NILS NIL . *NILS) (NILS *NILS))

(ASSERT (DIVIDE NIL NIL NIL)
(ASSERT (DIVIDE ((*CAR . *CDR) . *REST)
        (*CAR . *CARS) (*CDR . *CDRS))
        (DIVIDE *REST *CARS *CDRS))

(ASSERT (MAP *FN . *ARGS)
        (DIVIDE *ARGS *CARS *CDRS) (*FN . *CARS)
        (RETURN (MAP *FN . *CDRS)))
(ASSERT (MAP *FN . *ARGS) (NILS *ARGS))

(ASSERT (CARS NIL NIL))
(ASSERT (CARS ((*CAR . *CDR) . *REST) (*CAR . *RESULT))
        (CARS *REST *RESULT)))

(ASSERT (CDRS NIL NIL))
(ASSERT (CDRS ((*CAR) . *REST) *Y) (FAIL))
(ASSERT (CDRS ((*CAR . *CDR) . *REST) (*CDR . *RESULT))
                (CDRS *REST *RESULT))
```

There follows a definition of FOR which iterates the body, just like other conventional languages, giving a variable consecutive values from <FROM> to <TO>.

```
        (ASSERT (FOR *I *FROM *TO . *BODY)
                (WITH ((:FOR-INDEX))
                      (DO (*I *FROM (! ADD1 (! :FOR-INDEX *) *))
                      (<= *I *TO)
                      (SET :FOR-INDEX *I)
                      (AND . *BODY]
```

# APPENDIX C
# SYMBOL INDEX

**Reader's Comment Form**

**UTILISP in MTS**

**Volume 22**

**May 1988**

Errors noted in publication:

Suggestions for improvement:

Your comments will be much appreciated. The completed form may be sent to the Computing Center by Campus Mail or U.S. Mail, or dropped in the Suggestion Box at the Computing Center, UNYN, or NUBS.

Date: ______________________

Name:
______________________________________________________

Address: ______________________________________________________

______________________________________________________

______________________________________________________

Publications
Computing Center
University of Michigan
Ann Arbor, Michigan 48109
USA