utilisp

Short Contents

Table of Contents

1 Introduction

1.1 General Information

The UtiLisp32 is a new implementation of UtiLisp for Unix system. The original UtiLisp (University of Tokyo Interactive LISt Processor) system was designed for highly interactive programming and debugging of sophisticated programs on mainframes.

The new UtiLisp system is a transportation of the original UtiLisp. UtiLisp32 is for Unix 4.2 bsd machines whose CPU's have 32 bit address bus. It is now available on MC68010, MC68020 and Vaxen. This new UtiLisp is called "Unix UtiLisp" or simply "UtiLisp".

The transportation was done carefully so that the new system is compatible to old one. However some of operating system interface functions were not implemented or have different formats.

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 acquire a complete knowledge of the system from this manual.

1.2 How to Run and Stop the System on Unix

UtiLisp32 on Unix is supported as Unix shell command. It is invoked from shell as follows:

```
% utilisp
```

Options are:

`-h` *size*
: This specifies that the heap area is to be *size* kilo bytes. The default heap size is 512 kilo bytes.

`-ls` *size*
: This specifies that the parameter stack is to be size kilo bytes. The defalut stack size is 32 kilo bytes.

`-cs` *size*
: This specifies that the code stack is to be size kilo bytes. The defalust stack size is 16 kilo bytes.

`-bs` *size*
: This specifies that the binding stack is to be size kilo bytes. The default stack size is 64 kilo bytes.

`-es` *size*
: This specifies that the environment stack is to be size kilo bytes. The default stack size is 16 kilo bytes.

`-m` *size*
: The area used by malloc is to be size kilo bytes. The default area size it 16 kilo bytes.

`-d` *filename*
: The system is booted up from the file designated by *filename*. The sized of stacks and malloc area are automatically set to the corresponding ones when the dumpfile was executed.

`-gctype` *type*
: This specifies the Garbage Collection (GC) algorithm. 0 specifies the Copying GC; 1 specifies the Mark ans Sweep GC. Thought the Mark and Sweep needs 3 times as much GC time as the Copy GC, since it requires as a half heap memory, this algorithm might be superior for the programs that use the huge memory spaces.

`-n`
: This specifies that UtiLisp32 should not read and evaluate the file named `.utilisprc` in your home directory on starting up.

`-F` *filename*
: This specifies that UtiLisp32 should read and evaluate *filename* file on starting up.

`-p` *size*
: This specifies the `extendheap-ratio`(0-100). If the size of live cells exceeds `extendheap-ratio %` of heap size after an GC, `extendheap` was called and heap size becomes twice as before.

`-E` *expression*
: *expression* is evaled as an Lisp expression on starting up.

If you have a run command file named `" .utilisprc "` in your home directory, the UtiLisp32 system will read and evaluate it first. This evaluation is identical with that of the standard toplevel Lisp loop, except that the results are not displayed. The `-n` option supresses this initial evaluation.

After the evaluation of the run command file (if any), UtiLisp32 enters the toplevel loop. Each S-expression read in is evaluated and the result is displayed. Note that the toplevel evaluator is `eval`, not `evalquote`.

The session is terminated by evaluating the function `quit`. If one wishes to terminate the session abnormally, evaluate function `abend`.

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 UtiLisp32 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 interrupt) will stop the current computation and the system enters the `break` loop. For details, see Chapter ~see Chapter 14 [ErrDebug], page 81 `"Errors and Debugging"`.

1.3 Notational Conventions and Notes on Syntax

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

In this manual, Lisp symbols are printed in `typewriter type style` . *Italic words* appearing in S-expressions represent certain Lisp objects the details of which are irrelevant or explained elsewhere.

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 description of its arguments. And if it is not an ordinary function, its category will be shown in curly brackets, "" and "". Specifically, the categories are "Function”, "Special Form", "Macro", and "Variable". The following examples illustrate the manner in which the arguments are described:

quote *arg* — Special Form

`quote` is a " `special form` " and takes one argument.

cons *x y* — Function

`cons` is an ordinary function and requires exactly two arguments, *x* and *y*, and their absence generates an error.

gensym (*prefix*) (*begin*) — Function

`gensym` may take zero to two arguments; *prefix* and *begin* are optional.

plus *. args* — Function

`plus` may take arbitrarily many (possibly zero) arguments.

- *arg . args* — Function

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

As in the examples, argument names appear in italics in the description of the function.

The symbol "=>" is 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('), backquote(` ), comma( `,` ),semicolon( `;` ), and slash( `/`).

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 may 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.

Slashes are used for escaping 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 Chapter ~see Chapter 10 [Macros], page 57 "Macros".

1.4 Data Types

There are ten data types in UtiLisp32, i.e., `symbol`, `cons`, `fixnum`, `bignum`, `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 `symbol` 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 functions `set` and `setq`. The `definition` is functional attribute of the symbol; access is effected by `getd` and updating by `putd` or `defun`. The `property list` contains an even number(possibly zero) of elements; direct access and updating are 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 Chapter ~see Chapter 11 [Inand-Out], page 61, "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 may 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 three kinds of numerical objects in this system, upon which arithmetical operations may be performed; one is `fixnum` which possesses 28-bit signed integer value. `Bignum` is an `integer` of arbitrary length. Both `fixnum` and `bignum` are categorized as integers. Most of arithmetic functions convert the type between the two automatically. The other is `flonum` which possesses 64-bit floating point value. The accuracy is about 15 decimal digits in MC68000 series and 17 decimal digits in Vax.

A `string` is a finite(possibly zero) sequence of character. Each character has an 8-bit code value which is usually interpreted in terms of the ASCII code. Independent access to and updating of these characters are effected by means of the functions `sref` and `sset`, respectively. The length of a string may be known by applying the function `string-length`.

A `vector` is a finite(possibly zero) ordered set of Lisp objects. Vectors are created by means of the function `vector`. Access to the vector element is effected by means of the function `vref` and updating by the function `vset`. The length of a vector may be known by applying 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 is 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 compiled functions. Code pieces have names, normally a symbol, access to which is effected by means of the function `funcname`.

1.5 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 argument (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 becomes the final result of the application. The formal parameters are then unbound and the state is restored to that of preapplication. If the number of actual arguments is not equal to the length of *lambda-list*, an error is generated.

In UtiLisp32, an element of *lambda-list* may 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 list 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 given, then 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.

Examples of lambda-lists:

`(a b c)` actual parameters for `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))`
: `a` 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`.

2 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` otherwise.

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`.

2.1 Predicates on Data Types

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

symbolp *arg* Function

`symbolp` returns `t` if *arg* is a `symbol` ; otherwise `nil`.

consp *arg* Function

`consp` returns `t` if *arg* is a `cons` ; otherwise `nil`.

listp *arg* Function

`listp` is equivalent to `consp`. This is incorporated mainly for compatibility with other Lisp systems.

atom *arg* Function

`atom` returns `t` if *arg* is not a `cons` ; otherwise `nil`.

fixp *arg* Function

`fixp` returns `t` if *arg* is a `fixnum` object, i.e., a small integer number; otherwise `nil`.

bigp *arg* Function

`bigp` returns `t` if *arg* is a `bignum` object, i.e., a big integer number; otherwise `nil`.

integerp *arg* Function

`integerp` returns `t` if *arg* is a `fixnum` or a `bignum`, i.e., an integer number; otherwise `nil`.

floatp *arg* Function

`floatp` returns `t` if *arg* is a `flonum` object, i.e., a floating-point number; otherwise `nil`.

numberp *arg* Function

`numberp` returns `t` if *arg* is a `fixnum`, a `bignum` or a `flonum`, i.e., a numerical object; otherwice `nil`.

stringp *arg* Function

`stringp` returns `t` if *arg* is a `string` ; otherwise `nil`.

vectorp *arg* Function

`vectorp` returns `t` if *arg* is a `vector` ; otherwise `nil`.

referencep *arg* Function

`referencep` returns `t` if *arg* is a `reference` pointer; otherwise `nil`.

streamp *arg* Function

`streamp` returns `t` if *arg* is a `stream` ; otherwise `nil`.

codep *arg* Function

`codep` returns `t` if *arg* is a `code piece` ; otherwise `nil`.

2.2 General Purpose Predicates

The following functions are some other general purpose predicates.

eq *x y* Function

`eq` returns `t` if *x* and *y* denote the same Lisp object; otherwise `nil`. Lisp objects which have the same printed representations are not necessarily identical. However, the `interning` process ensures that two symbols with the same print name are identical (see Chapter ~see Chapter 11 [InandOut], page 61 "Input and Output", for details). Unlike some other Lisp systems, equality of values of small integer numbers (`fixnums`) may 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* Function

(`neq` *x y*) is equivalent to (`not` (`eq` *x y*))

equal *x y* Function

`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 `bignums` are `equal` if they have the same integer value, two `flonums` are `equal` if they have the same floating-point value, two `vectors` `equal` if their size is same and all their contents are `eq`, and two `cons` cells are `equal` if their respective `cars` and `cdrs` are `equal` inductively. 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* Function

null *x* Function

`not` returns the symbol `t` if *x* is `eq` to `nil`; the symbol `nil` otherwise. `null` is equivalent to `not`; both functions are incorporated for the sake of readability. It is recommended that `null` is used for checking whether a given value is `nil`, and that `not` be used for inverting a logical value.

UtiLisp32 also includes various predicates in addition to those introduced in this chapter. These will be described in the chapters on the various data types accepted by these predicates; for example, the predicate `zerop` is described in Chapter ~see Chapter 7 [Numbers], page 39, "Numbers".

3 Evaluation

3.1 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`, a `bignum`, a `flonum`, a `string`, a `code piece`, a `vector`, a `reference` or a `stream`, then the result of its evaluation is simply the form itself.

If the form is a `symbol`, then 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`, then 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 Chapter ~see Chapter 10 [Macros], page 57, "Macros".

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

3.2 Various Functions Concerned with Evaluation

eval *x* Function

`eval` evaluates *x*, and returns the result. Ordinarily, `eval` is not often 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* Function

`apply` applies the function *fn* to the set of arguments given by *arglist*, and returns the resulting value.

funcall *fn . args* Function

`funcall` 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.

Example: s

```
(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 *arg* Special Form

`quote` simply returns the argument *arg*. Its usefulness largely consists in the fact that its argument is not evaluated by the evaluator.

Example: s

```
(quote x) => x
(setq x (quote (cons 1 2))) => (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 `quote`d forms. For example,

```
(setq x '(cons 1 2))
```

is converted into

```
(setq x (quote (cons 1 2)))
```

function *fn* Special Form

The form (`(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 . *args* Special Form

`comment` ignores its arguments and always returns `nil`; it is useful for inserting explanatory remarks.

progn . *args* Special Form

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 *. args* Special Form

`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 effect occur.

Example:

```
(setq x (prog1 y (setq y x)))
```

This form interchanges the values of the variables `x` and `y`.

prog2 *. args* Special Form

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 *bindings . body* Macro

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 the corresponding *var*'s. For example, *vform2* cannot depend upon *var1*, that is, if *var1* appears in *vform2*, then a variable named *var1* must have been bound somewhere outside this `let` form.

lets *bindings . body* Macro

`lets` is similar to `let` except that `lets` binds its variables sequentially, one by one, while `let`, as mentioned above, binds them at once. (`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 *vforms-i* constitutes the default value list for the corresponding *var-i*, and therefore can depend upon the preceding *var*'s (see Section 1.5, "Lambda Lists", for details).

Note: The interpretation of `lets` is faster than that of `let`. However, once compiled, their speeds become identical.

4 Flow of Control

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

Functional 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 may be used for sequential execution of programs. The forms in a `progn` structure are evaluated sequentially from left to right.

In this chapter, 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.

4.1 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 . *args* Special Form

`and` evaluates the arguments sequentially, from left to right. If the value of some argument is `nil`, then `nil` is returned and the remaining arguments are not evaluated. If the value of all the arguments are non-`nil`, then 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` .

Example: s

```
(and x y)
(and (setq temp (assq x y))
     (rplacd temp z))
(and error-exists (princ "There is an error!"))
```

Note: `(and) => t`

or . *args* Special Form

`or` evaluates the arguments sequentially, from left to right. If the value of some argument is `nil`, the next argument is evaluated. If there are no remaining arguments, then `nil` is returned. However, if the value of some argument is non-`nil`, then 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 . *clauses* Special Form

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, then the value of the *antecedent* is returned. If the clauses are exhausted, that is, the value of every *antecedent* is `nil`, then the value of the `cond` form is `nil`.

selectq *key-form . clauses* Special Form

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 (only once). The resulting value is called the `key`. The *key-form* is followed by a number of `cluases`, 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, then `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` The key matches if it is `eq` to the atom.
2. A list 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.

Example: The preceding example is expressed with `selectq` 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.

match *key-form . clauses* Special Form

`match` is a special form for pattern matching. A `match` form has the following syntax:

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

The first argument *key-form* is evaluated first (only 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 `match` returns the value of the last *consequent*. If ther are no "matches", or if there is no *consequent* in the selected clause, then `nil` is returned. Note that the *patterns* are not evaluated.

The Objects which may be used as the *pattern* and their "matching" conditions are as follows:

1. `nil` The *key* matches if it is `nil`.
2. Any symbol except the symbol `nil` The *key* matches any symbol, and the symbol is `lambda`-bound to the *key*. The binding is unbound when the pattern matching fails. When the matching succeeds, the binding is kept during the evaluation of the clause and is unbound immediately before the evaluation ot the `match` form.
3. non symbolic atom The *key* matches if it is `eq` the atom.
4. (`quote` S-expression) The *key* matches if it is `eq` the S-expression.
5. `cons` other than 4 The *key* matches if its `car` "matches" the `car` of the *pattern* and its `cdr` "matches" the `cdr` of the pattern.

Example: s To return the S-expression if what is read in is of the form (`quote` S-expression); the first element if it is a list; `nil` for all other cases:

```
(match (read)
  (('quote sexpr) sexpr)
  ((top . rest) top))
```

The same program would become with `cond` form as follows:

```
(lets ((x (read)))
  (cond ((atom x) nil)
        ((and (eq (car x) 'quote)
              (consp (cdr x))
              (null (cddr x)))
        (cadr x))
        (t (car x))))
```

Function copy is realized with `match` as follows:

```
(match x
  ((head . tail) (cons (copy head) (copy tail)))
  (x x))
```

Note: When the some variables appear more than twice in *pattern*, consistency of the parts correspond to the same pattern would not be checked. The variable is `lambda`-bound by the last corresponding part.

4.2 Iteration

prog *locals . body* Special Form

`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 saved, 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*, then 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, then the variable is initialized to `nil`.

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 process reaches the end of the *body*, then `nil` is returned. However, two special devices (described below) may be used to alter the flow of control in the *body* of a `prog`.

When

`(return $x_1$ ... $x_n$)`

is evaluated, then processing of the *body* is terminated and the value of the last argument x_n is returned as the value of the `prog` form. If n=0, i.e., if no arguments are present, then the value returned will be `nil`. Only those `return` statements which are explicitly included in the body of a `prog` form should 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.)

When

`(go` *tag*`)`

is evaluated, then 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, then 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*), then an error is generated. Any statement of the form (go *tag*) must explicitly be included in the `prog` form containing the destination indicated by *tag*.

go *tag* Special Form

See the explanation under the entry for `prog` above.

Note: *tag* may be an atom of any type including `symbols` or `fixnums` . Since the process of searching is effected using the equality criterion `eq`, `bignum`, `flonums`, `strings`, `vectors`, etc. are generally not appropriate as labels.

return . *args* Function

See the explanation under the entry for `prog` above.

loop . *body* Special Form

`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 inner-most `loop` containing it is terminated and the value of the last argument of this `exit` is returned as the value of that `loop` form.

Example: The top-level loop of UtiLisp32, although actually defined in terms of machine language, could have been defined as follows:

```
(loop (print (eval (read))))
```

exit . *args* Function

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 *index-part exit-part* . *body* Macro

`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 second argument *exit-part* has the syntax as

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

After initially binding the index variables, and after each round of *next* value updating, 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` may 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.

Example: s Printing all the elements of a list `x` separated by a space may 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 may 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*.

do* *index-part exit-part . body* Macro

4.3 Non-local Exits

catch *tag . forms* Special Form

`catch` is a function primarily utilized for non-local exits non-local exit. (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 only 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`.

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* Function

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.

Example: The following program returns `a` rather than `b`.

```
(catch 'a (catch 'b (throw 'a 'a)) 'b)
```

4.4 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 shows the relations between the six map functions on list structures.

```
+---------------------------------------------------+
|                        |   applies function to    |
|                        |--------------------------+
|                        |  successive | successive |
|                        |   sublists  |  elements  |
+------------------------+-------------+------------+
|         |    its own   |             |            |
|         |     first    |    map      |  mapc      |
|         |   argument   |             |            |
|         +--------------+-------------+------------+
|         |  list of the |             |            |
| returns |    function  |   maplist   |  mapcar    |
|         |    results   |             |            |
|         +--------------+-------------+------------+
|         | nconc of the |             |            |
|         |   function   |   mapcon    |   mapcan   |
|         |   results    |             |            |
+---------+--------------+-------------+------------+
```

map *list fn* Function

The function *fn* is applied to the successive sublists of *list*, i.e., first *list* itself, then its `cdr`, then `cddr`, and so on. The value returned is its original argument *list* (possibly modified by *fn*).

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* Function

The function *fn* is applied to the successive elements of *list*, i.e., first the `car` of *list*, then its `cadr`, then `caddr`, and so on. The value returned is its original argument *list* (possibly modified by *fn*).

Example:

```
(mapc '(a b c) (function prin1))
```

This program prints out

```
abc
```

and returns (a b c)

maplist *list fn* Function

The function *fn* is applied to the successive sublists of *list*, i.e., first *list* itself, then its `cdr`, then `cddr`, and so on. The value returned is a newly created list of the results of these applications.

Example:

```
(maplist '(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* Function

The function *fn* is applied to the successive elements of *list*, i.e., first `car` of *list*, then its `cadr`, then `caddr`, and so on . The value returned is a newly created list of the results of these applications.

Example:

```
(mapcar '(a b c) (function prin1))
```

This program prints out

```
abc
```

and returns (a b c), which appears the same as the original arguments, but, actually, has newly been created.

mapcon *list fn* Function

The function *fn* is applied to the successive sublists of *list*, i.e., first *list* itself, then its `cdr`, then `cddr`, and so on. The value returned is the results of these applications concatenated together.

Example:

```
(mapcon '(a b c) (function ncons))
```

This program returns ((a b c) (b c) (c))

mapcan *list fn* Function

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 results of these applications concatenated together.

Example:

```
(mapcan '(a b c) (function ncons))
```

This program returns (a b c), which appears the same as the original argument, but, actually, has newly been created.

mapv *vector fn* Function

`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 Chapter 9, "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*).

Example:

```
(mapv (vector 5)
  (function (lambda (r)
              (setref r (read)))))
```

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

mapvector *vector fn* Function

`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.

5 Manipulating List Structure

5.1 Cons manipulation

car *x* Function

`car` returns the `car` of *x*. If *x* is an atom, an error is generated.

cdr *x* Function

`cdr` returns the `cdr` of *x*. If *x* is an atom, an error is generated.

c...r *x* Function

All the compositions of `car` and `cdr`, upto 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 end with `r`.

Example:

```
(cddar x)
```

is effectively the same as

```
(cdr (cdr (car x)))
```

cr *x* Function

`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* Function

`cons` is a primitive function which creates a new `cons` cell, the `car` and `cdr` of which are *x* and *y*, respectively.

Example: s

```
(cons 'a 'b)  =>  (a . b)
(cons 'a '(b c d))  =>  (a b c d)
```

ncons *x* Function

(ncons *x*) is effectively the same as (cons *x* nil)

xcons *y x* Function

`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.

Example:

```
                    (xcons 'a 'b)  =>  (b . a)
```

copy *x* Function

`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.

5.2 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`. Most functions treat the dotted list as if the last non-`nil` atom being `nil` .

last *list* Function

`last` returns the last top-level `cons` of *list*. If *list* is an atom, an error is generated; if the toplevel structure of *list* is cyclic, then an endless computation occurs.

Example:

```
                    (last '(a (b c) d e))  =>  (e)
          (last '(a b c . d)) => (c . d)
```

length *list* Function

`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.

Example: s

```
                    (length '(a (b c) d e))  =>  4
                    (length nil)  =>  0
          (length '(a b c . d)) => 3
```

first *x* Function

(first *x*) is equivalent to `(car x)`

second *x* Function

(second *x*) is equivalent to `(cadr x)`

third *x* Function

(third *x*) is equivalent to `(caddr x)`

fourth *x* Function

(fourth *x*) is equivalent to `(cadddr x)`

fifth *x* Function

(fifth *x*) is equivalent to `(car (cddddr x))`

sixth *x* Function

(sixth *x*) is equivalent to `(cadr (cddddr x))`

seventh *x* Function

`(seventh x)` is equivalent to `(caddr (cddddr x))`

nth *n list* Function

`nth` returns the *n*th top-level element of *list*, where (car *list*) is counted as the zeroth 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)`.

Example:

```
(nth 2 '(a b c d e))  =>  c
```

nthcdr *n list* Function

`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.

Example:

```
(nthcdr 2 '(a b c d e))  =>  (c d e)
```

list . *args* Function

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

Example: s

```
(list 1 2 (car '(3 5)) (+ 2 2))  =>  (1 2 3 4)
(list)  =>  nil
```

append . *lists* Function

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.

Example: s

```
(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 `append`ed and the order of the resulting list is not essential, the longest argument should be placed last since it is not copied; this reduces both computing time and required memory space.

reverse *list* Function

`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.

Example:

```
(reverse '(a (b c) d))  =>  (d (b c) a)
```

nconc . *lists* Function

`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).

Example: s

```
(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`. Note: when x is nil,x is not altered.So,you use not side effect but return value.

nreverse *list* Function

`nreverse` reverses its argument *list*, which is altered in the `rplacd` manner throughout the list (see also the description of `reverse`).

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 *item var* Special Form

(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 *var* Special Form

(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.

5.3 Alteration of List Structure

The functions `rplaca` and `rplacd` serve to alter existing list structure; that is, to change the `car` and `cdr` 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 effect side effect, 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* Function

`rplaca` replaces the `car` of *x* by *y* and returns (modified) *x*. *x* must be a `cons`, while *y* may be any Lisp object.

Example:

```
(setq x '(a b c))
(rplaca x 'n)  =>  (n b c)
x  =>  (n b c)
```

rplacd *x y* Function

`rplacd` replaces the `cdr` of *x* by *y* and returns (modified) *x*. *x* must be a `cons`, while *y* may be any Lisp object.

Example:

```
(setq x '(a b c))
(rplacd x 'c)  =>  (a . c)
x  =>  (a . c)
```

subst *x y z* Function

(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*.

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.

5.4 Tables

UtiLisp32 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 the element of which 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 Chapter ~see Section 5.6 [Hashing], page 33 for details).

memq *item list* Function

(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` may be used as a predicate.

Example:

```
(setq x '(a b c d e))
(memq 'c x)  =>  (c d e)
(memq 'foo x)  =>  nil
```

member *item list* Function

`member` functions in the same manner as `memq`, except that `equal`, rather than `eq`, is used for comparison.

mem *predicate item list* Function

`mem` functions in the same manner as `memq`, except that it takes an additional argument *predicate*, which 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)
```

Example:

```
(mem (function (lambda (x y) (0= (+ x y))))
     13
     '(1 3 -4 -13 7 -6))
   =>  (-13 7 -6)
```

delq *item list* (*n*) Function

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 exercised, except that any initial segment of *list* all the elements of which are `eq` to *item* is not altered in this manner (see 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.

Example:

```
(setq x '(a b a b))
(delq 'b x)  =>  (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 \code(b b) in both cases. However, `y` is given the value \code(b b) in the former case and `(a b b)` in the latter.

remq *item list* (*n*) Function

`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* Function

`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, then `every` returns `t`. If any of these applications yields `nil`, then `every` returns `nil` immediately, and no further applications are executed.

some *list predicate* Function

`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, then `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`, then `some` returns `nil`.

assq *item alist* Function

`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, or otherwise, `nil` is returned. The association list may be updated by applying `rplacd` to the result of `assq`, if the latter is not `nil`.

Example:

```
(assq 'c '((a b) (c d) (e f))) =>  (c d)
```

assoc *item alist* Function

`assoc` functions in the same manner as `assq`, except that `equal` instead of `eq` is used for comparison.

ass *predicate item alist* Function

`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`.

5.5 Sorting

sort *table predicate* Function

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.

Example:

```
(sort '(3 1 4 5 2) 'greaterp)
      => (5 4 3 2 1)
```

5.6 Hashing

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

UtiLisp32 provides a standard hashing function for Lisp objects to facilitate the maintainance of hashed data bases.

hash *x* Function

`hash` computes hash value for *x* and returns it as an integer number `fixnum` . The result may be positive, negative, or zero. Its properties guaranteed are:

1. Objects which are `equal` are hashed to the equal value.
2. A `fixnum` is hashed to itself.
3. A `bignum` is hashed to non negative value.
4. A string is hashed to non-negative value.
5. A symbol is hashed to the same value as its print-name.

6 Symbols

Symbolic atoms such as `x` or `cons` are called `symbols` in UtiLisp32. A `symbol` is associated with four Lisp objects; the `binding` is the value of the symbol when it is used as a variable; the `definition` is the functional definition of the `symbol` when it is used as the name of a function or a macro; the `property list` is used to retain various Lisp objects associated with the `symbol` ; the `print name` is used for input and output operations.

6.1 The Value

A `symbol` may 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 may 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` always must be 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* Function

`Assignment` to *variable* is 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 . *args* Special Form

(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*, . . .

Example: Values of two variables `x` and `y` are exchanged by

```
(setq x y y x)
```

boundp *variable* Function

`boundp` returns `t` if *variable* is bound to some value; otherwise, i.e., if it is unbound, `nil` is returned.

make-unbound *variable* Function

`make-unbound` makes *variable* unbound. The current value of *variable*, if any, is discarded. `make-unbound` returns the symbol *variable* as its value.

6.2 The Definition

A symbol may 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 nor a macro, the symbol is said to be `undefined` ; an error is generated when an undefined symbol is used as a function.

defun *name lambda-list . body* Macro

`defun` is 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 *name lambda-list . body* Macro

`macro` is 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 are more elegantly defined using `defmacro`. See Chapter ~see Chapter 10 [Macros], page 57 "Macros", for detail.

getd *sym* Function

`getd` returns the definition of a symbol *sym*. If *sym* is undefined, an error is generated.

putd *sym def* Function

`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* Function

`definedp` returns `t` if *sym* is defined as a function or a macro; otherwise, i.e., 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 nor a macro. Use the function `specialp` (see below) to discriminate special form indicators.

specialp *sym* Function

`specialp` returns `t` if *sym* is a special form indicator (such as `cond` or `prog`); otherwise, it returns `nil`.

make-undefined *sym* Function

`make-undefined` makes the symbol *sym* undefined. Current definition of *sym*, if any, is discarded. It returns *sym* as its value.

6.3 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` .

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 set initially to `nil`.

Note: `Printnames`, `bindings` and `functional definitions` are often implemented as `properties` of `symbols` in various Lisp systems; however, they are not implemented as usual `properties` in UtiLisp32.

get *sym name* Function

`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 that `property` exists or not, only from the result of `get`.

putprop *sym value name* Function

If the symbol *sym* has no `property` with its name being *name*, then `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 *sym value name* Macro

(defprop *x y z*) is effectively the same as (putprop 'x 'y 'z)

defnprop *sym value name* Macro

(defnprop *x y z*) is effectively the same as (putprop 'x (function *y*) 'z)

remprop *sym name* Function

`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.

plist *sym* Function

`plist` returns the `property list` of *sym*.

setplist *sym property-list* Function

`setplist` sets the `property list` of *sym* to *property-list*. It returns *property-list* as its value.

6.4 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 Chapter ~see Chapter 8 [Strings], page 47 "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 Chapter ~see Chapter 11 [InandOut], page 61 "Input and Output", for details).

pname *sym* (*new-name*) Function

`pname` returns the `print name` of the symbol *sym*. If the second parameter *new-name* is specified, the `print name` of the symbol *sym* is changed to *new-name*.

6.5 Creation of Symbols

symbol *pname* Function

`symbol` creates and returns a new uninterned `symbol` with its `print name` being *pname*.

symbol-copy *sym* Function

`symbol-copy` creates and returns a new uninterned `symbol` with its `print name` is same as *sym*.

gensym (*prefix*) (*begin*) Function

`gensym` generates a new print name, and creates a new "uninterned" symbol with that print name (see Chapter~see Chapter 11 [InandOut], page 61 "Input and Output", for "interning").

The generated `print name` is prefixed by a string, which is initially `g` but may be changed by giving `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*.

Example: s

```
(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,as far as they are not interned.

See also Chapter~see Chapter 11 [InandOut], page 61 "Input and Output", for `intern` which may create a symbol with given `print name` .

7 Numbers

There are three types of numbers in UtiLisp32, namely `fixnums`, `bignums` and `flonums` .

The `fixnums` of UtiLisp32 have a signed integer value of 28 bits. No overflow checking is made on arithmetical operations for `fixnums` only. All the results are treated modulo 2^28.

The `bignums` are arbitrary long integer. The integer which exceeds the length of `fixnum` will be `bignum` . The `bignums` are made when the reader reads a big integer and when results of computation exceed the limit of `fixnum` . But some functions don't do this conversion. The `fixnum` and the `bignum` are categorized to `integers` together.

The `flonums` have a 64-bit floating point value with the accuracy of about 15 decimal digits in MC68000 microprocessors. In case of Vax, the accuracy is about 17 decimal digits. Neither overflow nor underflow are checked on arithmetical operations on `flonums` .

`Integers` are denoted using conventional decimal notation (e.g., `15`) and `flonums` using decimal notation with a decimal point (e.g., `15.0`); `flonums` may also have "exponent part" indicated by the character "'^'" (e.g., '`1.5^1`').

Functions described in this chapter expect numbers of appropriate types for their arguments; if an argument of an illegal type is given, an error is generated.

Functions on numbers are grouped into three categories by the type of the numbers they accept: Functions the name of which includes alphabetic characters (e.g., `plus`) are applied to all of the types of the numbers. Conversions between `fixnum` and `bignum` are automatically made in these functions. 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. Notice that no overflow check or conversion to `bignum` are made for `fixnum` functions. These rules apply to all the functions described in this chapter except explicitly stated otherwise.

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

7.1 Numeric Predicates

zerop *x* Function

0= *x* Function

0=$ *x* Function

`zerop`, `0=` and `0=$` return `t` if *x* is zero (of proper type); otherwise, they return `nil`.

plusp *x* Function

0< *x* Function

0<$ *x* Function

plusp, 0< and 0<$ return t if *x* is a positive number (of proper type); otherwise, they return nil.

minusp *x* Function

0> *x* Function

0>$ *x* Function

minusp, 0> and 0>$ return t if *x* is a negative number (of proper type); otherwise, they return nil.

oddp *x* Function

oddp returns t if *x* is odd integer; otherwise, it returns nil. *x* must be a fixnum or a bignum .

= *x y* Function

=$ *x y* Function

= and =$ return t if *x* and *y* are equal numbers (of proper type); otherwise, they return nil.

Note: Equality of numbers is also tested using the function equal; equality of fixnums is also tested using eq.

*x y* Function

<> *x y* Function

#$ *x y* Function

<>$ *x y* Function

#, <>, #$ and <>$ return t if *x* and *y* are not equal numbers; otherwise, they return nil.

lessp *args* Function

< *args* Function

<$ *args* Function

`lessp`, `<` and `<$` return `t` if the number of arguments is less than 2, or each argument (except the last) is strictly smaller than its successor; otherwise, they return `nil`.

greaterp *args* Function

> *args* Function

>$ *args* Function

`lessp`, `<` and `<$` return `t` if the number of arguments is less than 2, or each argument (except the last) is strictly larger than its successor; otherwise, they return `nil`.

<= *args* Function

<=$ *args* Function

`lessp`, `<` and `<$` return `t` if the number of arguments is less than 2, or each argument (except the last) is strictly smaller than or equal to its successor; otherwise, they return `nil`.

>= *args* Function

>=$ *args* Function

`lessp`, `<` and `<$` return `t` if the number of arguments is less than 2, or each argument (except the last) is strictly larger than or equal to its successor; otherwise, they return `nil`.

7.2 Conversion Functions

fix *x* Function

`fix` converts a `flonum` *x* into an integer (a `fixnum` or a `bignum`) and returns that integer; rounding is used for the conversion. \beginfunctionfloatx `float` converts a `fixnum` or a `bignum` *x* into a `flonum` and returns that `flonum` .

7.3 Arithmetics

plus . *args* Function

+ . *args* Function

+$. *args* Function

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

minus *x* Function

`minus` returns the nagative of *x*.

difference *arg* . *args* Function

`difference` returns its first argument minus all the rest of its arguments.

- *arg* . *args* Function

-$ *arg* . *args* Function

With only one argument, `-` and `-$` behave the same as `minus`; they return the negative of the argument. With more than one arguments, `-` and `-$` are effectively the same as `difference`; they return their first argument minus all of the rest of the arguments.

times . *args* Function

***** . *args* Function

***$** . *args* Function

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

quotient *arg* . *args* Function

// *arg* . *args* Function

//$ *arg . args* Function

`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 escape character in UtiLisp32 syntax and must be doubled.

add1 *x* Function

`(add1` *x*`)` is equivalent to `(plus` *x* `1)`

1+ *x* Function

`(1+` *x*`)` is equivalent to `(+` *x* `1)`

1+$ *x* Function

`(1+$` *x*`)` is eqivalent to `(+$` *x* `1.0)`

sub1 *x* Function

`(sub1` *x*`)` is equivalent to `(difference` *x* `1)`

1- *x* Function

`(1-` *x*`)` is equivalent to `(-` *x* `1)`

1-$ *x* Function

`(1-$` *x*`)` is equivalent to `(-$` *x* `1.0)`

incr *var amount* Macro

(incr *var amount*) is equivalent to (setq *var* (+ *var amount*))

decr *var amount* Macro

(decr *var amount*) is equivalent to (setq *var* (- *var amount*))

remainder *x y* Function

‘’ *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* Function

`max` returns the largest of its arguments.

min *arg . args* Function

`min` returns the smallest of its arguments.

abs *x* Function

`abs` returns $`|`$*x*$`|`$, the absolute value of the number *x*.

expt *x y* Function

**** *x y* Function

\\$ *x y* Function

`expt`, `\` and `\$` return the *y*th power of *x*. *y* must be a `fixnum` . When *x* is a `fixnum` and *y* is non negative, then the result will be a `fixnum` or `bignum` ; in all the other cases, the result will be a `flonum` .

sin *x* Function

`sin` computes and returns sin *x*. *x* may be any type of numbers, and the result is a `flonum` .

cos *x* Function

`cos` computes and returns cos *x*. *x* may be any type of numbers, and the result is a `flonum` .

tan *x* Function

`tan` computes and returns tan *x*. *x* may be any type of numbers, and the result is a `flonum` .

arcsin *x* Function

`arcsin` computes and returns arcsin *x*. *x* may be any type of numbers, and the result is a `flonum` .

arccos *x* Function

`arccos` computes and returns arccos *x*. *x* may be any type of numbers, and the result is a `flonum` .

arctan *x* Function

`arctan` computes and returns arctan *x*. *x* may be any type of numbers, and the result is a `flonum` .

sqrt *x* Function

`sqrt` computes and returns the square root of *x*. *x* may be any type of nonnegative numbers, and the result is a `flonum` .

log *x* Function

`log` computes and returns the natural logarithm of *x*. *x* may be any type of positive numbers, and the result is a `flonum` .

log10 *x* Function

`log10` computes and returns the ordinary logarithm of *x*. *x* may be any type of positive numbers, and the result is a `flonum` .

exp *x* Function

`exp` computes and returns a `flonum` the natural logarithm of which is *x*.

7.4 Logical Operations on Numbers

Following functions treat `fixnums` as bit sequences of 28-bit long. If a non `fixnum` argument is supplied, an error is generated.

logor . *args* Function

`logor` returns bitwise logical `"or"` of the arguments. When no arguments are supplied, `0` is returned.

logand . *args* Function

`logand` returns bitwise logical `"and"` of the arguments. When no arguments are supplied, `-1` is returned.

logxor . *args* Function

`logxor` returns bitwise logical `"xor"` of the arguments. When no arguments are supplied, `0` is returned.

logshift *x y* Function

`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 28.

8 Strings

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

Characters of a `string` may be independently referenced and updated using `sref` and `sset`, respectively. The subscript origin for strings is zero. If a 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 generated.

`Characters` are `fixnums` which resides between 0 and 255, i.e., representable in one byte (8 bits). They are usually treated as ASCII character codes in input and output operations.

`Strings` may also be seen as `vectors` of small non negative integer `fixnums` ranging 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.

8.1 Characters

`Characters` are a `fixnum` which resides between 0 and 255. They are treated as ASCII codes in input and output operations.

character *x* Function

Some of the functions manipulating `strings` require their arguments to be a `character` . Though most of the functions introduced in this chapter 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.

8.2 String Manipulation

Note that the subscript origin for strings is zero.

string *x* Function

Functions manipulating `strings` require `string` arguments. Though most of the functions introduced in this chapter 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 `print name` is returned. If *x* is a `character`, a one-character `string` containing *x* is returned. Otherwise, an error is generated.

make-string *length* (*char*) Function

`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 given, 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* Function

`string-length` returns the number of characters in *string*, which is one more than the largest subscript value for *string*.

string-equal *string1 string2* Function

`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`.

Although comparison of equality of two `strings` is also effected by the function `equal`, `string-equal` is more specific and, therefore, more efficient.

string-lessp *string1 string2* Function

`string-lessp` compares two `strings` in dictionary order. The result is `t` if *string1* is the lesser, and `nil` if they are equal or *string2* is the lesser.

Example: s

```
(string-lessp "abc" "abd")  =>  t
(string-lessp "abc" "ab")  =>  nil
```

substring *string* (*start*) (*end*) Function

`substring` extracts a substring of *string*, starting at the character indexed by *start* up to but not including the character indexed 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 0.

Example: s

```
(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 that copy.

string-append . *strings* Function

Any number of `strings` are copied and concatenated into a single `string` . If no arguments are given, `string-append` returns a null string `""`.

string-reverse *string* Function

`string-reverse` returns a copy of *string* with the order of characters reversed. The original `string` is not physically altered (see also the description of `string-nreverse` below).

string-nreverse *string* Function

`string-nreverse` 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 also the description of `string-reverse` above).

string-search-char *char string* (*from*) Function

`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.

Example:

```
(string-search-char "b" "abcde")  =>  1
```

string-search-not-char *char string* (*from*) Function

`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*.

Example:

```
(string-search-not-char "0" "007") =>  2
```

string-search *key string* (*from*) Function

`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.

Example:

```
(string-search "word" "Where is the word?") => 13
```

translate *string table* Function

`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 upper-case characters to lower-case ones and the reverse, respectively. These tables are also used by the Lisp reader and the printer (see Chapter ~see Chapter 11 [InandOut], page 61 "Input and Output", for details).

string-amend *string1 string2* (*from*) Function

`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 0.

string-amend-and *string1 string2* (*from*) Function

string-amend-or *string1 string2* (*from*) Function

string-amend-xor *string1 string2* (*from*) Function

`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 0.

string-trim *string* (*char*) Function

`string-trim` trims consecutive *char*s from both left and right ends of *string* and returns it. The default value of *char* is a blank space.

string-left-trim *string* (*char*) Function

`string-trim` trims consecutive *char*s from left end of *string* and returns it. The default value of *char* is a blank space.

string-right-trim *string* (*char*) Function

`string-trim` trims consecutive *char*s from right end of *string* and returns it. The default value of *char* is a blank space.

string-match *pattern string* Function

`string-match` matches *string* against *pattern* and returns `t` or `nil` according to the result. `pattern` is a (limited) regular-expression, with special characters " `?` " and " `*` ". " `?` " matches any single character, and " `*` " matches any sequence of characters (possibly empty). There is no way to escape these special characters.

Example: s

```
(string-match "?b?" "abc")  =>  t
(string-match "*b*" "b")  =>  t
```

8.3 Manipulation of Characters in Strings

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

getchar *string index* Function

`getchar` returns the *index*-th character of *string* as an interned one-character symbol.

Example:

```
(getchar "abc" 2)  =>  c
```

sref *string index* Function

`sref` returns the *index*-th character of *string* as a character, i.e., a `fixnum` .

sset *string index character* Function

`sset` sets the *index*-th character of *string* to *character*, and returns *character*.

8.4 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* Function

`cutout` converts a character sequence beginning at the *pos*-th character of *string* with length *length* into a `fixnum` . *length* should be positive. Upper bytes of the result will be padded with zero.

spread *value length* Function

`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. If *value* cannot be represented in *length* bytes, only lower bytes are converted and overflowed upper bytes are ignored.

8.5 Bit String Manipulation

A string may also be regarded as a sequence of binary digits (bits). Thus, an array of logical values may be represented by a string, in which case, one character holds 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, following functions are provided by UtiLisp32. Compact representation of bit tables using 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* Function

`bref` returns `t` if *index*-th bit of *string* is set; otherwise, it returns `nil`. *index* should be non negative and smaller than eight times the length of *string*.

Note: Bits are indexed from left to right, the most significant bit of the 0th character of `string` being 0.

bset *string index value* Function

If *value* is non `nil`, the *index*-th bit of *string* is set; otherwise, it is reset. *index* should be non negative and smaller than eight times the length of *string*. `bset` returns *value* as its value.

9 Vectors

A `vector` is a Lisp object that consists of a number (possibly zero) of elements, each of which is a Lisp object again. The individual elements are selected by numerical subscripts origined zero. An error is generated if an subscript value specified is not appropriate, i.e., if it is negative or not less than the number of the elements.

As elements of a `vector` are accessed in constant time, it is advantageous compared with list structure consisting of binary `cons` cells when a large amount of data is to be manipulated. Disadvantage of using `vectors`, compared with `lists`, is that the size should be known before used.

`Vectors` are arbitrarily allocated and discarded like `cons` cells; they are independent objects on 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`, to use the `symbol` as its name, since `vectors` are not directly identified by the Lisp reader.

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

9.1 Vector Manipulation

vector *size* (*filler*) Function

`vector` allocates and returns a `vector` with its size being *size*; its subscript ranges from 0 to *size* - 1. If the optional argument *filler* is not given, all the elements of the allocated `vector` are initiated to `nil`. Otherwise, if *filler* is given, 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* Function

`vector-length` returns the number of elements of *vector*.

vref *vector subscript* Function

`vref` returns the *subscript*-th element of *vector*.

vset *vector subscript value* Function

vset sets *value* into the *subscript*-th element of *vector*. `vset` returns *value* as its value.

fill-vector *vector filler* Function

`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.

Example: s When the value of `v` is a `vector` with, for example, 10 elements,

```
(fill-vector v nil)
```

fills the vector with `nil`'s.

```
(fill-vector v '(0 1 2 3 4))
```

sets first 5 elements of the `vector` with `0`, `1`, `2`, `3`, and `4`, respectively. 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`.

9.2 References

It is often required 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. UtiLisp32 provides `reference` objects for this purpose.

A `reference` is a pointer to an element of a `vector` . The pointed element is accessed by `deref` and updated by `setref`. `deref` and `setref` are also applied to variables, i.e., symbols. It is recommended that `deref` and `setref` should be used in functions which utilize `call-by-reference` parameter, instead of `eval` and `set`.

reference *vector subscript* Function

`reference` makes and returns a `reference` pointing to the *subscript*-th element of *vector*.

deref *reference* Function

`deref` returns the value of *reference*; if it is a symbol, the value of the symbol; if it is a `reference`, the element of a `vector` it is pointing.

setref *reference value* Function

`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* Function

`referred-vector` returns the `vector` an element of which is pointed by *reference*.

Note: Computation of this function requires time proportional to the subscript of the element pointed by *reference*.

referred-index *reference* Function

`referred-index` returns the subscript of the `vector` element pointed by *reference*.

Note: Computation of this function requires time proportional to the subscript of the element pointed by *reference*.

See also `mapvector` and `mapv` (Section 4.4, "Mapping") which perform certain computation on all the elements of a `vector` .

10 Macros

10.1 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`, then that definition is called a `macro`. The `cdr` of the definition is treated as a function that has 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.

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 deffn Macro symbol `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 may be expanded recursively; 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 are used for a variety of purposes. For example, `custom-made` control structures are easily implemented with macros.

Example: `while-do` construct such as

(while-do *condition* . *body*)

is defined as a macro using `macro` special form 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 are 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 may 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.

macro-expand . *form* Macro

`macro-expand` only expands a macro in *form* and returns it without second evaluation.

Example:

```
(macro-expand incr x 3) => (setq x (plus x 3))
```

10.2 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 damages the readability of the resulting program. `defmacro` facility is provided to facilitate access to portions of the argument list by giving names to portions of the argument list.

defmacro *name arg-pattern . body* Macro

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`, 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.

Example: The `while-do` in the former example may be more elegantly defined using `defmacro` as follows:

```
(defmacro while-do (condition . body)
   (nconc (list 'loop
                (list 'and condition '(exit)))
          body))
```

10.3 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 a time: The expanded form which will be finally evaluated is one; the form which produces that form is the other, and this form is what the programmer have 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 Chapter ~see Chapter 11 [InandOut], page 61, "Input and Output" for detail), which acts similarly to normal single quote (') that makes a *quote*d 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 `quote`d while all the other portions are effectively `quote`d.

Example: s ‘`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.

`while-do` macro may be still more elegantly defined as

```
(defmacro while-do (condition . body)
    `(loop (and,condition (exit))
           .,body))
```

Backquotes may be nested. When backquotes are nested twice, double comma will make 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.

11 Input and Output

11.1 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 may 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 *filename* Function

`stream` makes a stream which is connected to the external file defined by *filename*.

When making a stream from a Unix file descriptor which has already been opened, the descriptor must be given as a fixnum to the *filename* parameter.

string-stream *string* Function

String *string* is used as a stream.

inopen *stream* Function

`inopen` opens *stream* as an input stream. When opening is unsuccessful, an error is generated; otherwise, it returns *stream*.

outopen *stream* Function

`outopen` opens *stream* as an output file. When opening is unsuccessful, an error is generated; otherwise, `outopen` returns *stream*.

appendopen *stream* Function

`appendopen` opens *stream* as an open file and makes it append mode. When opening is unsuccessful, an error is generated; otherwise, `appendopen` returns *stream*.

close *stream* Function

`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`, `appendopen` and `close`; the user may not update the value of `openfiles` explicitly.

Example: All the files currently open are closed by

```
(mapc openfiles (function close))
```

stream-mode *stream* Function

`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*) Function

`linelength` returns the maximum line length allowed, for output streams. The default value for *stream* is the value of `standard-output`.

cursor (*stream*) Function

`cursor` returns current column position of *stream* for output streams, where column zero being the first column. The default value of *stream* is the value of `standard-output`.

colleft (*stream*) Function

When *stream* is an output streams, 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`) for outputstreams.

charleft (*stream*) Function

colleft for the variable length line.

string-stream-index (*string-stream*) Function

The fixnum number of bytes already read or written from the beginning of the string-stream.

string-stream-limit (*string-stream*) Function

The fixnum number of bytes readable or writable for the string-stream.

standard-input Variable

standard-output Variable

Values of these variables are streams for which I/O operations are normally performed; values of these variables are used as the default values of stream arguments in various I/O functions. Reading and printing are 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. The initial values 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).

Example:

```
(let ((standard-input some-stream)) (read))
```

is effectively the same as

```
(read some-stream)
```

terminal-input Variable

terminal-output Variable

Values of these variables are the streams which are connected to the user terminal.

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

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 certain string which identifies the subsystem to notify the terminal user what the prompting system is, or, what kind of input is expected.

Example:

```
(setq name
      (let ((prompt "Who are you?  "))
           (read)))
```

11.2 Allocating Files

alloc *filename* Function

`alloc` returns *filename* itself. This function is only for compatibility with UtiLisp on mainframes.

11.3 Printed Representation

Lisp objects are not directly handled since they are stored inside the machine memory. In order to examine these Lisp objects, UtiLisp32 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.

11.3.1 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 is 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 on its type.

For an `integer` : If the `integer` is negative, the printed representation is preceded by a minus sign (-); if non negative, no sign is printed. Then comes the decimal representation of the absolute value of the `integer` . Slashification does not affect the printing of `integers` .

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. Number of digits in the fraction part is specified by the value of the symbol `digits`. Then comes the exponent part indicator (‘`^`’), sign of the exponent part (`+` or -), and the value of the exponent 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` . If `slashification` is on, some special characters are preceded by the escape character `/` . The decision whether escape is required is made using the current readtable, i.e., the current value of the `symbol readtable`. Objects printed with `slashification` on 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 to 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 is printed, using this new `cons` followed by a close parenthesis. 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` are not read back in properly.

For a `stream` : The printed representation has the syntax `S#` *name*, where *name* is the name of the external file which the `stream` is connected to. When the `stream` is connected to terminal, the *name* is `"terminal-input"` or `"terminal-output"` . `Streams` are not read back in properly.

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`), and *address* is the decimal representation of the current address of the object. The address is merely for convenience in discriminating two objects; the objects may be relocated by the garbage collector. `Vectors` and `references` are not read back in properly.

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, and, thus, the length of the printed representation of a `flonum` is, initially, 14.

atomlength *x* Function

`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

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 zero to these variables.

11.3.2 The Reader

The purpose of the reader is to accept characters, interpret them as the printed representation of a Lisp object, and return the corresponding Lisp object. The reader does not accept all the printed representations; the printed representations of vectors, references, streams, and code pieces are not 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 may be defined as single character object, which are read in as a one character `symbol` of that character. Macro characters may be defined, reading which 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 name` of `symbols` . The registration process to 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 list which is the element of the obvector, the index of which is given by

```
(‘’ (hash (pname sy))
   (vector-length obvector))
```

default-obvector Variable

Value of `default-obvector` is the initial value of `obvector`. All the predefined symbols are initially registered in this table.

oblist (*obvector*) Function

`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 when 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 principle may be established by binding `intern` to a user-defined `interning` function.

intern *string* (*obvector*) Function

`intern` searches *obvector* for a `symbol` which has the `print name` `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*) Function

`intern-soft` works almost the same as `intern` except that it does not create a new `symbol` . *obvector* is searched for a `symbol` with the `print name` `string-equal` to *string*. If it is found, a list beginning with that `symbol` is returned; if not found, `nil` is returned.

remob *symbol* (*obvector*) Function

`remob` searches *obvector* for a `symbol` which is `eq` to *symbol*. If found, it is removed from the table making it hidden from the Lisp reader; if not, nothing is done. It returns `nil` as its value. The default value of *obvector* is the current value of the symbol `obvector`.

11.3.3 The Readtable

The reader is controlled by a vector called the `readtable`. A `readtable` is a vector consisting of 256 `fixnum` elements, the index *n* element of which corresponds to the character of ASCII code *n*, and indicates the nature of the character.

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

`0x0000001`
: (LSB) means that this character is an ordinary alphabetic character. All the usual characters have this bit on and others off.

`0x0000002`
: means that this character is an extended alphabetic character. This bit is currently not used.

`0x0000004`
: means that this character is a digit. Characters `"0"` through `"9"` has this bit on.

`0x0000008`
: means that this character is a sign. `"+"` and `"-"` has this bit on.

`0x0000010`
: is `alternate meaning` bit. This bit is used in several ways. For example, `"-"` has this bit on, while it is off for `"+"`.

`0x0000020`
: means the escape character. `" / "` has this bit on.

`0x0000040`
: means that this character should be slashified in a `symbol` . Characters with special meaning have this bit on.

`0x0000080`
: means that this character should be slashified when appeared at the top of a `symbol` . Special characters, signs, and digits have this bit on.

`0x0000100`
: means string quote character. Double quote has this bit on.

`0x0000200`
: means macro character. The macro definition (a function with no argument) is in the corresponding position of the macrotable.

`0x0000400`
: means right parenthesis, `" ) "`.

`0x0000800`
: means dotted pair dot, `"."`.

`0x0001000`
: means left parenthesis, `" ( "`.

`0x0002000`
: means blank and alike, which is normally skipped between lexical elements.

`0x0004000`
: means a single character object.

`0x0008000`
: means that this 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

Values of `readtable` and `macrotable` are the current readtable and the macrotable, respectively. The initial value of these variables are the same as those of `default-readtable` and `default-macrotable`, respectively (see below). User defined readtable or macrotable may be used by binding these variable to certain values.

default-readtable Variable

default-macrotable Variable

Value of these variables are the standard readtable and the standard `macrotable` of the system.

11.3.4 Setting Readtable

Characters may 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*) Function

char is defined as a macro character associated with *fn*. This definition is done in *readtable* and *macrotable* given as arguments. If they are absent, current values of `readtable` and `macrotable` are assumed.

Example: s The macro character "'" could have been defined by

```
(readmacro (character "'")
  (function (lambda nil (list (quote 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 input from that stream.

If the backquote character " ‘ " never be typed in from certain terminal, an alternative character, say, " % ", may be settled for backquote macro by

```
(readmacro "%" (vref macrotable 96))
```

96 is the ASCII code for " ‘ ".

Predefined read macros are quote "'", backquote " ‘ ", and comma " , ". See Chapter 10, "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), then it is read as an `interned` single character `symbol`, regardless of preceding or following characters. Single character objects may be defined by the function `single-character`.

single-character *char* (*readtable*) Function

char is defined as a single character object in *readtable*. If *readtable* is not supplied, current value of `readtable` is assumed.

Example:

```
(single-character "&")
```

From then on,

```
a&nil&b
```

will be read as 5 symbols, `a`, `&`, `nil`, `&`,and `b`.

11.4 Input Functions

Functions described in this section bind the variable `standard-input` to the argument *stream*, before reading any character in. Thus, input is always performed on `standard-input` stream. The default value of *stream* is the current value of `standard-input`.

read (*stream*) Function

`read` reads in one printed representation of a Lisp object from *stream*, and returns it as its value.

readline (*stream*) Function

`readline` reads the current line, from current position to the line end, and return 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*) Function

`skipline` works the same as `readline` except that it returns `nil`, instead of a string.

current-line (*stream*) Function

`current-line` returns the current line of *stream* as a string object. Returned string includes all the characters in the current input line, regardless of the current character position. The character position is not affected. Notice that this function only reads the current buffer contents of *stream* and never updates the *stream*. The line which is spread over two buffers is not read by `current-line`.

tyi (*stream*) Function

`tyi` inputs one character from *stream* and returns its code as a `fixnum` .

tyipeek (*stream*) Function

`tyipeek` returns the next character of *stream*. The difference with `tyi` is that `tyipeek` does not advance the current character position of *stream*. Thus, consecutive calls of `tyipeek` will result the same.

readch (*stream*) Function

`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.

11.5 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*) Function

`prin1` outputs the printed representation of *x* to *stream*, with `slashification` . The value of `prin1` is *x*.

print *x* (*stream*) Function

`print` works the same as `prin1`, except that `print` terminates the current line after printing out.

princ *x* (*stream*) Function

`princ` is the same as `prin1` except that the printing is done without `slashification` .

tyo *char* (*stream*) Function

`tyo` outputs the character whose ASCII code is specified by *char* to *stream*. `tyo` returns *char* as its value.

terpri (*stream*) Function

`terpri` terminates the current line of *stream*. `terpri` returns `nil` as its value.

flush (*stream*) Function

`flush` flushes out the contents of buffer of *stream* and returns `nil`. In general, output of newline character flushes out `stream` .

tab *n* (*stream*) Function

`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.

11.6 Formatted Printing

It is often required to print Lisp objects in the midst of a certain message. For example, given a symbol *sy* and a number *num*, one might require such an output as

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

with *sy* and *num* varying 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 not readable.

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

format *pattern . args* Macro

`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 slant character (`/`) is encountered, printing is controlled by the directive character immediately following it. If the directive character requires arguments, values of *args* are used sequentially from left to right. Control directive characters currently available and their meanings are as follows:

`s` prints one Lisp object with `slashification` .

`c` prints one Lisp object without `slashification` .

`b` prints one character the code of which is supplied as an argument.

`g` pretty-prints one Lisp object.

`t` tabulates to the column specified by the argument.

`n` terminates the current line.

`/` prints " `/` ".

Case of directive characters is ignored.

Example: The former example is printed by

`(format "The symbol /s appeared /s times./n"` *sy num*`)`

11.7 Indented Printing

Printed representations of Lisp object are not easily examined by human eyes, especially when parentheses are densely nested. The indented printer `prind` will help you producing more readable outputs by giving appropriate indentation.

prind *x* (*width*) (*asblock*) (*level*) (*length*) Function

x is printed with certain indentation. *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`, then the print out 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 should 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 print out of `prind`. The initial value of `usebq` is `nil`.

pp *funcname* Macro

The definition of the symbol *funcname* is printed so that the definition will be recovered when the print out is read in and evaluated. Usual functions are printed as

(`defun` *funcname lambda-list . body*)

Macros defined using `defmacro` are printed as

(`defmacro` *funcname lambda-pattern . body*)

Other macros are printed as

(`macro` *funcname lambda-list . body*)

12 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 itself be used as functions, it is usually more convenient to use their names, i.e., symbols, as functions. `Code pieces` are either predefined by UtiLisp32 or obtained by compiling lambda forms.

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

funcname *code* Function

`funcname` returns the name of *code*.

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 may be examined by the following functions.

minarg *code* Function

maxarg *code* Function

`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 is generated when *code* is applied to less arguments than the result of `minarg` or more than the result of `maxarg`. If *code* allows arbitrarily many arguments, `maxarg` returns -1.

A `code piece` may be constructed by the following function.

load-code *x* Function

`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 half word (16 bits in Suns, 8 bits on Vaxen) of the machine code, and, finally, *quoted* is a list of Lisp objects accessed from the machine code.

program-load *c-library . funcs* Macro

The C program functions in the C library are used as the Lisp functions. *funcs* is a list of the form

(*Lisp-function-name C-function-name list-of-the-parameter-types* `nil` *type-of-the-return-value*)

The return value type may be one of fix, float and string.

```
(program-load '("-lm") '(arctan2 "atan2" (float float)  nil float))
```

code-load *compiled . c-library* Macro

A number of the object files are loaded. *compiled* is a list of the object files generated by a compiler. *c-library* specifies other C libraries. A list of the file names may be given to compiled.

This function can not be used when the incremental load does not exist in System V. In that case, make-a.out may be used for the compiled codes to be executed.

make-a.out *a.out compiled c-library* Macro

`make-a.out` generates the executable file for UtiLisp/C which includes the compiled codes. As the compiled code does not automatically execute lispsys.l, compied must include lispsys.o.

dumpfile *filename* Function

Currently loaded codes and heap information are written out to a file. When this file is specified with -d option at the next rebooting, the system will restart with the same state.

13 Compilation

The Lisp compiler is a program which translates interpretive functions, which have the form of lists, into machine codes which are directly executed by the hardware. The merit of compilation is that the execution speed will be considerably improved.

13.1 Compiling Functions

compile . *function-names* Macro

The compiler is evoked by simply applying the macro `compile` as

`(compile . `*function-names*`)`

where *function-names* is a list of symbolic atoms the definitions of which are `lambda` expressions. The definition of these `symbols` will be replaced by the compiled code. `compile` returns the list *function-names* as its value.

Example: Interpretive functions `f` and `g` are compiled by:

```
(compile f g)
```

revert . *function-names* Macro

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 whether such check codes are required or not.

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 generates check codes for undefined function; otherwise, no run time check code for index range is generated. The initial value of `udfcheck` is `t`.

13.2 Declaration

Various declarations may be required for exact compilation. The macro `declare` and `defconst` and the function `reset-compilation-flags` are provided for such declarations.

declare *item-list indicator* Macro

`declare` is used to declare that the elements of *item-list* have the attribute indicated by *indicator*. Currently, the indicators used are `special`, `redefine` and `fix-value`.

defconst *var val* Macro

When it is evaluated by the interpreter, it has the same effect as (`setq` *var val*).When it is compiled, the code in which *vars* are replaced by *vals* is generated. An error occurs when *var* is assigned a new value in the context in which *var* is assumed a constant.

reset-compilation-flags Function

`reset-compilation-flags` revokes all the declarations effected via the macro `declare` so far.

The compiled object is designed so as to use `static` scope rule for local variables(authentic Lisp scope rule is `dynamic`). For exact compilation of functions which utilize global variables, all the non locally referred variables(i.e., variables referred from functions other than that which binds the variable) should be declared to be `special`.

The declaration of `special` variables is effected by

`(declare` *var-list* `special)`

where *var-list* is a list of non locally referred variables.

Example: When variables `x` and `y` are used non-locally, they should be declared special before compiling functions which binds them by

```
(declare (x y) special)
```

If a non local variable is not properly declared, the compiler treats the variable as a special variable; the value of a local variable is stored somewhere on the system stack access to which can only be possible 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.

By the declaration

(declare *sym-list* fix-value)

you can tell the compiler that the symbols in *sym-list* have only fixnum value.

13.3 Storing Compiled Objects

The compiler puts the compiled code in a relocatable form (in a form of list of numbers and some Lisp objects) in the property list of the name of the compiled function as its `compiled-code` property. This may be printed to a file as a normal Lisp object and may later be read back in. This relocatable form may be converted into 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 Chapter ~see Chapter 12 [CodePiec], page 75,"Code Pieces", for details).

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` may be loaded by:

```
(putd 'f (load-code (read obj)))
```

13.4 Difference from the Interpreter

As the compiled object is designed so as 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. Only available non-local exit structure is that provided by `catch` and `throw`.

13.5 Providing Space for Compiled Codes

Compiled codes are stored in an area called `fixed-heap` which is different from usual `heap` for ordinary Lisp objects. When a large amount of code should be compiled, the size of the `fixed-heap` must be specified to be large enough. This can be achieved by supplying an optional parameter `"-f"` to the Unix command `utilisp` as

% utilisp -f *n*

where *n* is a number indicating how many kilobytes should be provided for compiled objects. The default value of *n* is 64. If enough fix heap space doesn't exist, then Lisp process abnormally terminates.

As the garbage collector does not collect garbages in the `fixed-heap` area, re-compilation of functions leaves uncollectable garbages. See Chapter ~see Chapter 15 [MMS], page 87, "Memory Management System", for details.

14 Errors and Debugging

14.1 The Error System

UtiLisp32 generates an error when some invalid operation is tried by the program; for example, when the `car` of an atom has been taken.

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

Then the value of the symbol `break` is examined, which should be a function of no argument, and this function is then called in the environment where the error has 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 during the evaluation of which the error occurred.

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 Function

`break` first binds `standard-input` and `standard-output` to the streams connected to the terminal, i.e. `terminal-input` and `terminal-output`, `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 may be terminated by the

unbreak . *args* Function

`unbreak` is used to terminate a `break` loop. The inner-most `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.

Following are the variables the values of which are used as the 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.

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)))))
```

err:argument-type Variable

err:argument-type *x* (*where*) Function

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

err:buffer-overflow Variable

err:buffer-overflow *dc* (*where*) Function

A string or a symbol is read in which is longer than the string buffer. The size of the string buffer is, currently, 512 characters. *dc* is always `nil`.

err:catch Variable

err:catch *tag* (*where*) Function

`throw` was called with its first argument being *tag*, but the corresponding `catch` with its first argument `eq` to *tag* was no found.

err:end-of-file Variable

err:end-of-file *stream* (*where*) Function

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*) Function

Overflow of a floating point number occurred. *dc* is always `nil`.

err:function Variable

err:function *x* (*where*) Function

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` expression.

err:go Variable

err:go *tag* (*where*) Function

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*) Function

index was used as an index for a vector or a string, but is out of index range or not even a `fixnum` .

err:io Variable

err:io *stream* (*where*) Function

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*) Function

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*) Function

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

err:read Variable

err:read *dc* (*where*) Function

The character sequence read in cannot be interpreted as a Lisp object. This error is often caused by an improper usage of dots (`.`).

err:return Variable

err:return *dc* (*where*) Function

`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*) Function

var was evaluated but is unbound.

err:undefined-function Variable

err:undefined-function *fn* (*where*) Function

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

err:variable Variable

err:variable *x* (*where*) Function

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

err:zero-division Variable

err:zero-division *dc* (*where*) Function

Division by zero was attempted. This error may occur in both integer and floating arithmetics. *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 the error is printed, all the variables recover their top-level values, and UtiLisp32 resumes the top-level loop.

When a stack overflow occurs during a garbage collection, UtiLisp32 prints out a message and the Lisp session is terminated abnormally, since such a situation is fatal and recovery is impossible.

14.2 Attention Handling

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

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`.

14.3 The Debugger

The debugger is a collection of functions which are useful in debugging Lisp programs. As debugger is designed for interpretive functions, it is recommended to debug programs in interpretive form and then compile them into machine codes (see Section~see Chapter 13 [Compile], page 77,"Compilation", for details).

trace . *funcnames* Macro

`trace` takes arbitrarily many 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 effected by automatically rewriting the definition of the `traced` functions. This alteration can be restored by the function `untrace`.

trace-when *pred* . *funcnames* Macro

`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 value 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 . *funcnames* Macro

`untrace` stops `tracing` of the functions listed in *funcnames*.

backtrace (*n*) Function

With no argument, `backtrace` returns a list of the names of all the functions which are nesting around the current environment. When *n* is supplied, a list of the names of only *n* innerly nested functions is returned. Inside the `break` loop of the standard error handler, this function may be used to examine the calling sequence upto where the error has occurred.

oldvalue (*n*) Function

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 `*UnBoundVariable*`. The order in the list is such that recently bound variables come earlier. When *n* is supplied, only pairs concerning recent *n* bindings are included. This function may be used to get information on the binding history.

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`.

toplevel Function

`toplevel` first undoes all the variable bindings except top-level ones. Then the value of the variable `toplevel` is examined. The value should 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.

14.4 The Low-Level Debugger

The low-level debugger is a collections 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* Function

`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* Function

`peek` returns a string which contains a copy of the machine memory beginning at (`address` *addr*) and *length* long.

15 Memory Management System

The memory space used by the UtiLisp32 is divided into four areas. They are:

`heap`
: for usual Lisp objects (including symbol area)

`fixed-heap`
: for predefined objects and compiled codes

`stack`
: for control information and temporary storage

`kernel`
: for the system kernel

The size of the `heap` and the `fixed-heap` and areas may be specified by the parameters at the initiation of the Lisp process (see Chapter ~see Chapter 1 [Introduc], page 1, "Introduction", for details). The sizes of the `kernel` area is system-defined constants. 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 not enough space is left in the heap area, then the garbage collector is called.

The garbage collector gathers all the Lisp objects which will never be accessed; the memory space occupied by them becomes reusable. Then, the execution of the original program is resumed.

The garbage collector may also be called explicitly by `gc`.

gc — Function

`gc` invokes the garbage collector. It returns `nil` as its value.

Following functions are for asking states and setting parameters of the memory management system. The unit of memory space used in these functions is `byte` in UtiLisp32.

extendheap *size* — Function

`extendheap` expands the size of the heap area specified at the initialization to *size* bytes. Success will return t, failuare nil.

extendheapK *size* — Function

`expandheapK` expands the size of the heap area to *size* kilo bytes. Success will return t, failuare nil.

gc-hook — Variable

When a value is bound to gc-hook, this value will be funcall-ed after GC.

gccount — Function

`gccount` returns a `fixnum` which indicates how many times the garbage collector has been called since the system initiation.

gctime Function

`gctime` returns a `fixnum` indicating CPU time required for `gc` so far. The unit used is one 60th second.

heapsize Function

`heapsize` returns the size of the heap area as `fixnum` .

heapsizeK Function

`heapsizeK` returns (heapsizeK)/1024.

heapfree Function

`heapfree` returns the size of the free heap area as `fixnum` .

heapfreeK Function

`heapfreeK` returns (heapfreeK)/1024.

symsize Function

`symsize` returns the size of the symbol area as `fixnum` . In UtiLisp/C, `symsize` returns 0 since the heap area is not divided for each type.

symfree Function

`symfree` returns the size of the free symbol area as `fixnum` . In UtiLisp/C, `symfree` returns 0 since the heap area is not divided for each type.

fixsize Function

`fixsize` returns the size of the fixed heap area as `fixnum` . In UtiLisp/C, `fixsize` returns 0 since the heap area is not divided for each type.

fixfree Function

`fixfree` returns the size of the free fixed heap area as `fixnum` . In UtiLisp/C, `fixfree` returns 0 since the heap area is not divided for each type.

stacksize Function

`stacksize` returns the size of the stack area as `fixnum` .

stack-used Function

`stack-used` returns the size of the stack area currently in use.

stack-bottom Function

`stack-bottom` returns the bottom of the stack.

stack-top Function

`stack-top` returns the top of the stack.

stack-base Function

`stack-base` returns the current base of the stack.

stackWM Function

`stackWM` returns the list of pairs of the maximal stack size ever used and the max stack size (in bytes) for each of the parameter stack, binding stack, code stack and environment stack.

init-stackWM Function

`init-stackWM` will reset the max record displayed by `stackWM`.

16 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 such as Emacs or vi, 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; arbitrary Lisp form may be evaluated during editing and currently edited structures may 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 with the original ones, but all the `cons` cells are newly created for the copy. Thus, when a USE session is aborted by a `k` command, the original program or data is not affected at all.

16.1 Invoking USE

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

ed *fn* Macro

`ed` invokes USE for editing interpretive functions. The `definition` of *fn* will be edited. `ed` puts edited result in the `definition` cell of *fn* on normal termination.

edv *var* Macro

`edv` invokes USE for editing `values` of variables. The `value` of *var* is edited. `edv` sets edited result in the `value` cell of *var* on normal termination.

edp *sym* Macro

`edp` invokes USE for editing `properties` of `symbols` . The `property list` of *sym* is edited, and the result will become the new `property list` of *sym* on normal termination.

edf *file-name* Macro

`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.

What is edited is a list of all the objects the printed representations of which are stored in the file. The order of the elements in the list is the same as in the file. In other words, an extra left and a right parentheses are assumed at the beginning and at the end of 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 should be evaluated to effect redefinition.

edl *loc* Macro

`edl` invokes USE for editing some data which never be edited by `ed`, `edv` nor `edp`. *loc* should be a form to access a component of certain structure, for example, (car *cons*) to access the `car` of a `cons` cell *cons*, (vref *vec n*) to access *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 to where it was derived from, on normal termination.

use *x* Function

`use` is the USE system itself. It is normally called using above macros, but users may also call `use` directly. `use` returns one component list of the edited result, when the `use` session is normally terminated; it returns `nil` when it is terminated abnormally (by `k` command).

16.2 USE Session

USE prompts `terminal-input` by `"?"` when it expects a command. If a symbol or a number 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 may be used to evaluate an atom.

e *form* Use

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.

Following two commands are for terminating an editor session.

q Use

`q` (quit) terminates the current session normally. The edited result will be restored depending how the editor is called from one of the macros described in the previous secrion.

k Use

`k` (kill) terminates the current session abnormally. The edited result will be merely discarded no matter how the editor is called, and the original definition, value, property list, etc. are not affected.

16.3 Scope and Position Numbers

The editor always have current `scope`, which is a portion of the whole Lisp object being edited. The `scope` may be nested; the current scope may be an element of its parent scope, and this parent scope may again 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 are 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` means the first element and `-1` the last.

Example: s Suppose 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.
```

16.4 Pattern Matching Rules

It is sometimes desired to search a certain pattern of Lisp object, without specifying its detail. For example, a form `setq`ing to a symbol `x` may be of programmer's concern irrespective of the value assigned. This example may be expressed as (setq x ?).

The rules of pattern matching are quite simple:

1. A pattern matches an Lisp object `equal` to it.
2. `?` matches any Lisp object.
3. `???` matches any portion of a list.

Example: s

```
(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) .
```

16.5 Printing Current Scope

p Use

`p` (print) command prints the current scope in usual abbreviated way. See Chapter ~see Chapter 11 [InandOut], page 61, "Input and Output", for abbreviated printing.

pp Use

`pp` (pretty print) command prints the current scope with appropriate indention. See Chapter ~see Chapter 11 [InandOut], page 61, "Input and Output", for pretty-printing.

level *n* Use

length *n* Use

`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 editor session. The values of `printlevel` and `printlength` are not affected.

16.6 Changing the Scope

n Use

-*n* Use

0 Use

Position numbers themselves are commands to 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 Use

`top` command changes the scope to the whole Lisp object being edited.

n Use

`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.

l Use

`l` (last) command moves the scope to the element one 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 Use

`w` (where) command prints where the current scope is beginning from the top level.

16.7 Searching

f *pattern* Use

`f` (find) command searches an Lisp object 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 Lisp object 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* Use

`ff` (find forward) command is the same as `f` command except that the search begins in the next scope, that is after the current scope.

fb *pattern* Use

`fb` (find backward) command is the same as `f` command except that the search is performed in reverse direction (`cdr` before `car`) and the search begins in last scope, that is, before the current scope. current scope.

fn Use

`fn` (find next) command is the same as `ff` command except that the same *pattern* is used as previous `f` `ff` or `fb` command.

16.8 Inserting and Deleting Parentheses

bi *m n* Use

`bi` (both in) command inserts an open parenthesis at the left of *m* and a close parenthesis at the right of *n*. *m* and *n* are position numbers.

Example:

```
(a b c d e)
   bi 2 4
--> (a (b c d) e)
```

bo *n* Use

`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`.

Example:

```
(a (b c d) e)
   bo 2
--> (a b c d e)
```

li *n* Use

`li` (left in) command inserts an open parenthesis at the left of *n*, and a close parenthesis at the end of scope. *n* is a position number.

Example:

```
(a b c d e)
   li 2
--> (a (b c d e))
```

ri *n* Use

`ri` (right in) command inserts an open parenthesis at the beginning of scope and a close parenthesis at the right of *n*. *n* is a position number.

Example:

```
(a b c d e)
   ri 2
--> ((a b) c d e)
```

lo *n* Use

`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.

Example:

```
(a (b c d) e)
   lo 2
--> ((a b c d) e)
```

ro *n* Use

`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.

Example:

```
(a (b c d) e)
   ro 2
--> (a (b c d e))
```

16.9 Inserting and Deleting S-expressions

i *pos sexpr* Use

`i` (insert) command inserts *sexpr* at 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 Lisp object 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.

Example: s

```
(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 to the top of a list can be achieved by specifying 0 for *pos*.

a *sexpr* Use

`a` (append) command replaces the tail of the current scope by *sexpr*. If the current scope is atomic, the whole scope is replaced by *sexpr*.

Example: s

```
nil
   a (a b c)
--> (a b c)
   a d
--> (a b c . d)
```

in *sexpr* Use

`in` (insert next) commands inserts *sexpr* at the right of the current scope in the parent scope.

d *pos* Use

`d` (delete) command deletes *pos* from current scope. The meaning of *pos* is the same as in `i` command.

Example: s

```
(a b c d e)
   d 3
--> (a b d e)
   d b
--> (a d e)
```

y *pos* Use

`y` (yank) command inserts an Lisp object most recently saved by the editor at the right of *pos*. What is saved is either the Lisp object deleted using `d` command, Lisp object 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 `i` command. This command can be used, with `d` command, to move a portion of Lisp object inside the edited structure.

Example: s

```
(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) d e)
```

16.10 Replacing S-expressions

r *pos expr* Use

`r` (replace) command replaces *pos* with *expr*. *pos* has the same meaning as in `i` command.

Example:

```
(a b c d e)
   r 3 (foo bar)
--> (a b (foo bar) d e)
```

ra *pattern expr* Use

`ra` (replace all) command replaces all Lisp object in the current scope which matches *pattern* with *expr*. Number of actual replacements is reported.

Example:

```
(a x b x c)
   ra x y
--> (a y b y c)
2 occurrences are replaced
```

17 Unix Interface

17.1 Calling Shell Commands

call *command-string* Function

This function executes *command-string* and waits its termination. Because the execution is done by a subprocess rather than UtiLisp32 itself, some commands such as `cd` have no effect on the status of UtiLisp32. Return value is a `fixnum` that represents the status of the command execution.

cd (*dir*) Function

`cd` changes the current working directory of UtiLisp32 to *dir* which must be a string. The default value for *dir* is the user's home directory defined by the HOME environment variable.

17.2 Environment Variables

getenv *name* Function

`getenv` searches environment variable list for the name *name* and return its value as a `string` . If the variable is not defined, `nil` is returned.

Example:

```
(getenv "HOME") => "/usr/usr1/bill"
```

putenv *name value* Function

getpid Function

`getpid` returns the process ID of the current process.

syscall ... Function

`syscall` calls the Unix system call.

errno Function

`errno` returns the error number returned by the system call.

17.3 Command Line Arguments

argv Function

`argv` returns the command line that invoked UtiLisp32 as a list of `strings` . Note that the command name itself is also included as the first element of the list.

18 Miscellaneous

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

time *form* Function

With no argument, `time` returns the CPU time elapsed by the Lisp system since its initiation. This includes the time required for garbage collection and for external programs which are called using `call`. If the optional argument *form* is supplied, *form* is evaluated again, and CPU time required for this re-evaluation is returned. The time is returned as a `fixnum` object in one 60th seconds.

quit Function

`quit` will return control to the caller of the UtiLisp32, usually to Unix shell. All the files opened by in the UtiLisp32 will be automatically closed.

abend Function

`abend` abnormally terminates the UtiLisp32.

version Variable

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

date-time Function

`date-time` returns a string containing the date and time.

The string has the format

```
"YYMMDDHHMMSS"
```

where `YY` are two least significant digits of the year, `MM`, month, `DD`, day, `HH`, hour in 24-hour system, `MM`, minute, `SS`, second.

Example: At 5:30 in the evening of January the 20th, 1988,

```
(date-time)  =>  "880120173000"
```

userid Function

`userid` returns user name as a `string` .

utilisp Function

`utilisp` is the top-level loop of the UtiLisp32. An S-expression is read in, evaluated and printed. This is repeated again and again. The prompting character

of the top-level loop is `">"` . This symbol `utilisp` is the initial value of the symbol `toplevel` (see Chapter ~see Chapter 14 [ErrDebug], page 81, "Errors and Debugging", for details).

? Variable

Each time a form read in is evaluated in a `utilisp` loop or in a `break` loop, the result is set to the variable `?`.

Example: s In the top-level UtiLisp loop,

```
(cons 'foo 'bar)  =>  (foo . bar)
?  =>  (foo . bar)
```

setl *loc val* Macro

`setl` is a macro which makes it easy to describe list structure modification or vector element updating. It is particularly useful for defining such macroes that access and update an element simultaneously. *loc* is either a variable name or an expression which indicates the element in list or vector. *val* is a value set to the place indicated by *loc*. `setl` rerurns the value *val*. Example: s

```
(setl x y)   is equivalent to  (setq x y)
(setl (car x) y)   is equivalent to  (car (rplaca x y))
(setl (cadr x) y)   is equivalent to  (car (rplaca (cdr x) y))
(setl (vref v 3) y)   is equivalent to  (vset v 3 y)
(setl (plist x) y)   is equivalent to  (setplist x y)
(setl (nth n x) y)   is equivalent to  (car (rplaca (nthcdr n x) y))
```

exfile *filename* (*show*) Function

`exfile` evaluates (executes) all the S-expression in the file specified by *filename* and returns `nil`. If *show* is non-`nil`, `exfile` output the result of each evaluation. The default value for *show* is `nil`.

package-load *lispfile* . *externals* Macro

19 Common Lisp like Libraries

(to be included)

20 X-Window Interface

(to be included)

Index

(Index is nonexistent)