Klone Reference Manual

V 1.2
August 1995

Colas Nahaboo
Koala Project
BULL Research
colas@sa.inria.fr
http://www.inria.fr/koala/colas

Copyright

Author:	Colas NAHABOO -- BULL Research FRANCE -- Koala Project Klone Language and GWM X11 Window Manager Architect.
Internet:	colas@sophia.inria.fr
Surface Mail:	Colas NAHABOO INRIA B.P.93 - 06902 Sophia Antipolis Cedex FRANCE
Voice phone:	(33) 93.65.77.70
Fax:	(33) 93 65 77 65

Contents

Conventions

When describing the syntax of contructs, the following conventions are used:

exemples are typed in this font and slightly indented:

```
This is an exemple
```

return value of functions are signalled by a right arrow: $\longrightarrow$
Printed values and other side effects are signalled by a double-right arrow: $\Longrightarrow$

```
(+ 2 2)                          → 4
(progn (print "foo") "bar")      ⇒ foo
                                 → bar
^&*                  ⇒ ERROR: undefined variable
```

Actual text to be typed is printed this way:

`(), nil, setq`

expression in this *font* Indicates one of the following:

- an expression: *form, expr , inst*
- an expression of a precise type: *string, list, number , symbol, object, stream*
- an expression of a precise meaning: *variable, value*

***** Indicates the previous item is optional and can be repeated any number of times. E.g., for the rule **(list** *item ** **)**, the following are correct:

```
(list)    (list "hello")            (list "Hello" "world")
```

+ Indicates the previous item can be repeated any number of times but must be present at least once. E.g, the rule **(and** *form +* **)** is equivalent to **(and** *form form **), and the following examples satisfy this rule:

```
(and (= a 8))           (and (= a 8) (<= b 7))
```

{} Indicates that the group enclosed in braces is considered as a single item. Examples for the rule: **(with (***{variable value}+* **)** *instruction ** **)** are:

```
(with (a 1) (print a))
(with (a 1 b 2) (print a b))
```

[] Indicates that the group enclosed in brackets is optional. Examples for the rule **(defvar** *variable [value [documentation]]* **)** are:

```
(defvar *foo*)          (defvar bar 2327)
(defvar foo-bar 314159 "This is only a test")
```

| Is a separator between alternatives. Valid examples for the rule **(***{variable |* **(v***ariable value* **)** *}****)** are:

```
() (a b (c 7) d)
```

CHAPTER 1 Klone reference

This is the Klone reference for all standard klone constructs and primitives for Klone. It contains all information on the features provided by all applications embedding Klone, but should be completed by the reference manual of application-specific primitives added to this kernel.

This manual is grouped by topics, i.e. functions are grouped by types on which they operate, or by the service they provide.

Entries will be shown in the form:

Function name *Function description* *Syntax* *Comments*

defconstant define a constant CL p86

(defconstant *variable initial-value [documentation]* **)**

Description

This function declares a variable to be a constant. Such a constant

Comments include Common Lisp compatibility, and is listed in as reference to pages in Common Lisp: The language, second edition[1]. It can be:

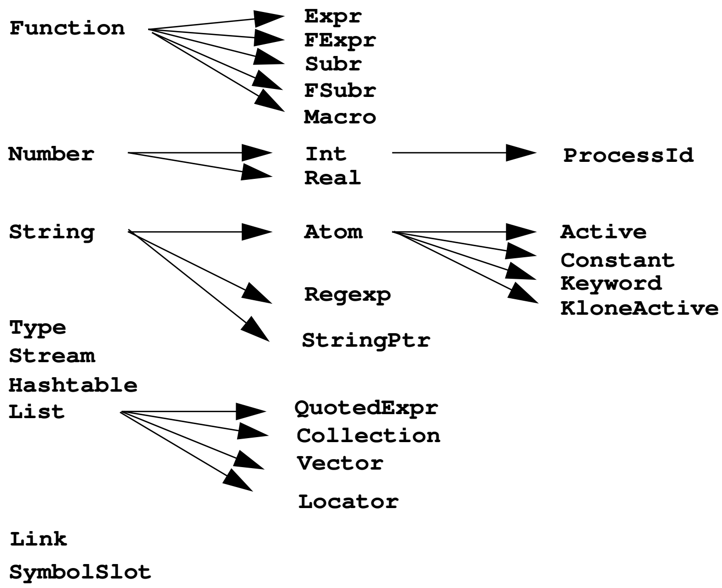
- CL meaning that this function is a subset of the Common Lisp command
- CL# meaning that this function is close to the Common Lisp one, but is incompatible in some points with its Common Lisp counterpart. DIfferences with Common Lisp will be listed in paragraphs labelled: **CL-Diff**
- nothing if this function doesn't exist in Common Lisp
- other possible comments: INTERNAL, TEMPORARY, EXPERT, etc...

1. Steele, Guy L. Jr. Common LISP: The Language. Second Edition. Digital Press, 1990.

1.1 Types

1.1.1 Type Hierarchy

The type hierarchy of the built-in base Klone types is:



1.1.2 Primitive types

The primitive types found in any Klone system are:

- **Function** is not an actual type, but the father of 5 subtypes. In Klone, as in Scheme, functions are first-class objects, i.e. `(defun foo (x) x)` is equivalent to `(setq foo (lambda (x) x))`.

  Expr, **FExpr** and **Macro** are user-defined functions. Expr evaluates the arguments before binding them, FExpr do not evaluate, and Macro do not evaluate the arguments, but evaluate the value returned by the execution of the function body (statements).

  Subr and **FSubr** are built-in C functions callable from Klone. Subr evaluates their argument before executing the C function, whereas FSubr (special forms) do not.
- **Number** is not an actual type, but the father of 2 subtypes:

  Int are 32 bit signed integer numbers,

  Real are floating point numbers, coded on 64 bits. Reals can be used anywhere integers are expected, and their integer part is used.
- **String** is implemented as byte strings (array of 8-bit characters). String characters can take the ASCII values 0 to 255. There is no limit on the size of strings[1], and strings can include thus any binary value, including null bytes, as string length is stored in a field and Klone do not use a terminating null byte (except for regexps). Special strings are **Regexp**, compiled patterns used for regular expression matching on strings.

  Note: This a new feature of Klone. In previous versions, Strings were plain C strings, so were null-terminated and could not include null bytes. For better C interfacing, a redundant null byte is still appended to all Klone strings, though.

1. In fact Strings are limited to 2^32 -1= 4294967297 characters

Strings can also be written as **raw strings** (page 11), whith no need to quote any charcters inside them, useful for embedding strings that are texts of other languages, where having to quote the embedded quotes or backslashes can be a pain. Raw strings are just strings between `^^` (Control-^, ascii code 30 / 0x1e / 036) characters instead of double quotes. Raw strings cannot contain null bytes nor `^^`characters, however.

Stringptr is a subtype of String referring to a portion (a substring) of an existing string. This is an efficiency hack but should not be modified (see `substring-ptr`, page 46)

- **Atom** is the type of normal symbols. There is no limit on the size of atoms names[1]. In Klone (as in Scheme), atoms are mono-valuated, that is a symbol can have only one value, whereas in Lisp, atoms can refer to a function or a value depending of the context.

  Special types of atoms are **Keyword**, symbols prefixed by a colon (`:`) that evaluate to themselves and are used as options specifier for arguments to functions, **Constant** that are symbols whose value cannot be changed once defined, and **Active** (active values), objects performing side-effects when evaluated or set.
- **Regexp** are regular expressions. These are special objects, compiled from strings, that hold the details of the preceding match to extract all informations of the match of the substrings for further use.
- **Type** is the type of type objects (used to coerce objects into, for instance). Although types are printed as symbols, they are actually a distinct primitive type.
- **Stream** is the type of Input/Output media on which Klone can read and write. They can be opened on Unix files (or sockets) or on character strings. (see the `open` function page 49).
- **QuotedExpr** is the type of quoted expressions, used only for efficiency. It can be thought as a pair (list of two elements), first element being the quoting symbol (`' ` , , ,@`) and the second element being the quoted symbol.
- **Hashtable** is the type of the Klone hashtables, used to manage very efficiently Klone object indexed by other objects. It replaces p-lists with a structure trading space for speed. Roughly access time to an element grow linearly in a p-list whereas it is nearly constant in a hashtable, the times being equivalent for both solutions for less than 5 elements
- **List** is the type of Klone lists, which are in fact arrays (see page 11). **Collection** is a special type of lists, optimized for parsing, that you shouldn't encounter in normal Klone use.
- **Vector** are just like lists, but evaluate to themselves, and are printed and read as lists prefixed by the sharp `#` character, as in `#(a 1)`, for Common Lisp compatibilty, or as square brackets, as in `[a 1]` for readability.
- **Link** is the type of "soft links" used to refer to objects without incrementing the reference count on them, to be able to create circular lists which otherwise couldn't be properly de-allocated by the garbage collector
- **SymbolSlot** is a pointer to a slot in the global symbol hashtable.
- **Locators** are a shorthand to write series of calls to `get` and `put`, which are the cornerstone functions of Klone programming. They are writen as `#[...]`.

1.1.3 primitive objects

- **()**, aka `nil`, (the logical *false* value "*nil*") is an empty list. In Klone, due to the nature of the implementation of lists, multiple different empty lists can co-exist (they are not `eq`, but `equal`), but they are all considered to represent the logical *false* value. One

1. Actually, in the current implementation, symbols are limited to 8192 characters, but this value may be increased at compile time.

empty list is used as "*nil*", and is printed as **()**, other empty lists are printed as **( )**, and the system will prevent you from physically modifying **()**.

```
(list)                  ⇒ (  )
(= () (list))           ⇒ t
(eq () (list))          ⇒ ()
(if (list) 'a 'b)       ⇒ b
(put () 'a 0)           ⇒ ERROR: Cannot put in ()
```

Warning: the symbol **nil** is not a special symbol, unlike other lisps. it is just a symbol having **()** as value. thus unlike traditional lisps, **(eq () 'nil)** is false!

__NOTE__: the symbol **nil** is predefined to have the value **()**, and in the rest of this document, we will often use the term "nil" to designate **()**. To create an empty list different from the empty list **()**, use the calls **(list)** or **(copy ())**.

__NOTE__: Unlike traditional lisps, the symbol **nil** is not a special token. Thus **'nil** is not equivalent to **nil** or **()**.

- **t** (the logical *true* value) is the symbol **t** which evaluates to itself. This is not the only logical true value, however. Every non-false value is considered logically true in Klone. If you are used to C programming, beware that the number **0** is **not** a logical false value in Klone.

```
(if 0 'true 'false)          → true
```

1.2 Syntax

Here is the external syntax of all objects than can be read by Klone. The actual implementations of the types are detailed in The Klone C Reference Manual.

; Comments

Comments CL p526

; end-of-line

In Klone, comments begin with a semicolon "**;**" and end at the end of the line

Numbers

Numbers (integers and reals)

Numbers can be either integers or reals (floating-point numbers).

[+-]?[0-9]+

Integers are 32 bits signed integers. There is no special character type, they are just represented as integers. Integers (and thus characters) can be notated:

- in decimal (default), optionally prefixed by **+** or **–**: **74 +12 -3**
- in hexadecimal, prefixed by **#x** or **0x**: **#x4a #x4A 0x4a**
- in ASCII code, prefixed by **#**: **#\J** for the ascii code of J (74). Control characters can be specified prefixed by a caret (**#\^J** means Control-J, ascii code 10).
- by name for specific characters (name prefixed by **#**). Currently defined names are:

name	meaning	code
\space	blank character	32
\newline	division between lines (Control-J)	10
\backspace	backspace character (Control-H)	8
\rubout	delete	127
\tab	horizontal tab (Control-I)	9

{ [+|-] <digit> }+ [. <digit>*] [`e|E][+|-]<digit>}<digit>}]}

Reals are implemented using the underlying C compiler ***double*** floating point type (usually on 64 bits). They can be used anywhere integers are expected, the Klone interpreter will convert them to integers as needed.

Examples of reals:

```
1. 1.0 -1.6e06 7.6543E-02          ; example of reals
(type-of 1.)                       ⇒ Real
(type-of -3)                       ⇒ Integer
(typep -2.5 Number)                ⇒ t
```

The Klone type of integers is **`Int`**, of reals is **`Real`**, and of numbers is **`Number,`** which is a pseudo type used to coerce into, but with no actual instance

CL-Diff: There is no separate character type as in CL, characters are just represented in Klone by their ASCII code as 8 bit numbers.

Symbols

Klone identifiers CL# p27

*[^0-9();"'‘,@{}][^();"'‘,@{}]**

The symbols are identifiers allowing the naming of Klone objects such as functions and variables. Upper case and lower case letters are distinct, meaning that **`foo`** is not the same symbol as **`Foo`**.

As in scheme, Symbols are ***mono-valuated,*** i.e. a symbol has only one meaning, whereas in traditional lisp systems symbols have different values when they are used as variables or functions. For instance you should not type **`(setq list ())`**, since this would have the undesired side-effect to make Klone forget the definition of the **`list`** function.

A symbol can be used everywhere a string is expected (except for functions performing physical modifications of strings such as **`put`**). The only exception is equality, since atoms are not equal to strings. Use coerce to coerce a symbol into a string.

Packages are not supported fully in Klone. A parse-time facility is provided with the prefix character `%` (see **`*package*`** page 29).

Scope Klone implements Dynamic Scoping. This was the traditional method of scoping used in the lisp systems until recently. Now most modern lisps are statically scoped, which ensures safer operation, and compiles more efficiently, but is slower and more memory-consuming while interpreted.

Dynamic Scoping means that a variable can have only one value, and it can be accessed from anywhere at any time, e.g:

```
(defun reset-a (setq a 0))         ; side effect on a
(let ((a 1)) (reset-a) a)          → 1 in static scoping
(let ((a 1)) (reset-a) a)          → 0 in dynamic scoping
```

CL-Diff: in CL, symbols case is not significant, symbols can begin with a digit (1+ is valid in Common Lisp, but not in Klone) scope is static, and symbols are multi-valuated.

Strings
"..."
^^...^^

Strings of characters CL p33

A string is a sequence of characters enclosed in double quotes ("). The type of strings is **`String`**. Strings can span multiple lines, and will include the newline characters, unless the line ends with a backslash (\) character. Special characters can be entered into a string by the means of the following escape sequences:

sequence	stands for	Code
`\n`	newline (Control-J)	10
`\r`	carriage return (Control-M)	13

sequence	stands for	Code
\t	tab (Control-I)	9
\e	escape (Control-[)	27
****	the backslash character itself \	92
\”	an imbedded double-quote	34
\xNN	the character of ASCII code **NN** (in hexadecimal)	
\nnn	the character of ASCII code nnn (in octal)	
\c	**c** if **c** is any other character	

a followed by **b** on another line can be written as the following forms:

```
"a\nb" "a\x0ab" "a\012b" "a
b" "a\n\
b"
```

Raw strings: The third form (between Control-caret characters, ascii code 30 or 0x1e) are **raw strings**, with no escape sequence interpretation. Useful to avoid “quoting hell” when preparing texts to be given to other programs using also double-quotes for strings, such as tcl, tk, perl... Unfortunately these Raw Strings cannot contain Control-caret characters. They can be printed via the global flag

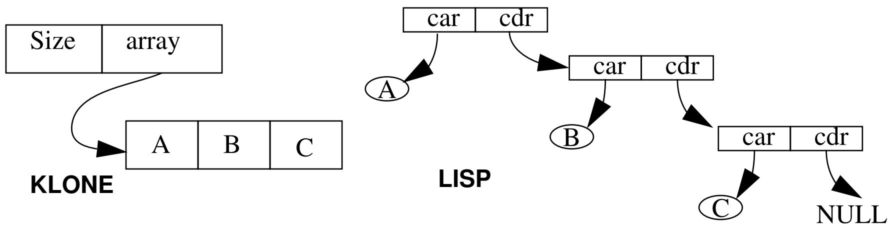
Lists
(...)

Lists of elements CL# p 29

(*{elements} ** **)**

Lists of elements are written enclosed in parentheses and with elements separated by spaces (space, tab, newline characters)

CL-Diff: The main difference between KLONE and a traditional LISP system is that in KLONE lists are represented as arrays of contiguous elements, whereas in LISP, lists are implemented as linked cells, the ***cons*** cells.



This allows Klone to be more easily integrated with C applications, since the Klone list type reflects the “argc, argv” typical C approach of handling arrays and passing a variable number of parameters to functions. And, since in lisp and klone function code is stored as lists, code execution is faster and takes less memory space, at the expense of a slower execution of list modification functions. However Klone is fast enough that in practice this list modification overhead is small. For instance, if you are to add an element to the head of a list, it is done in constant time (one atomic operation) in lisp dialects, whereas is Klone it is an operation proportional to the length of the list, but experiences has shown that in xlisp[1], inserting an element at the head of a list (a cons) is as slow as inserting an element to the head of a 400-element list in Klone!, which involves copying the 400 elements.

Efficient Klone coding is thus done by using iterative functions (such as **map**, **dolist**, etc...) rather than splitting lists into “car” and “cdr” (head and tail). Long lists are also more efficiently handled than lots of small lists, thus alternatives to traditional lisp functions such **cond** and **let**, (**if** and **with**) have been provided to reduce list fragmentation.

1. a small lisp interpreter comparable to Klone, written by David Betz

Property Lists, Associative Lists: Although not a separate Klone type, we will often refer to property lists (or **p-lists**). These are even-numbered lists, with even elements used as keys and the following element used as a value. Property lists are a simple and natural way to store properties of various kinds, and notably the symbol p-lists (See “symbol-plist object-plist” on page 29.). Property lists are handled via the put, get and delete functions, and can be coerced to and from Hashtables.

```
(get '(a 1 b 2 c 3) 'b)          → 2   ; used as a p-list
(get '(a 1 b 2 c 3) 2)           → b    ; used as a list
```

Associative lists (or **a-lists**), although natural in traditional lisp systems, are not supported in Klone, as they are less natural due to the underlying list implementation. Associative lists are lists of (key, value) sublists. For instance, **`with`** (page 26) uses p-lists for variable declarations, whereas **`let*`** (page 26) is exactly the same function, but requiring a-lists for its declarations.

Note: keys can be of any type, except numbers. If you must index a table by numbers, use a Hashtable instead of a p-list.

Vectors
#(...)
[...]

lists that evaluate to themselves CL p32

#(*elements...* **)**

Vectors are exactly like lists, but they just don’t print in the same way and more importantly evaluate to themselves. Thus the #(...) read form quotes its contents when reading it:

```
#(a 1)              →            #(a 1)
(a 1)               ;; will try to evaluate a as a function
```

Vectors are better suited to hold data than lists, that should be normally reserved for program texts. All list functions applies to vectors, of course, and the conversion between lists and vectors can be made as a copy by **`coerce`** or in place by the **`vector!`** and **`list!`** functions.

For instance, it is recommended to write: **`(with [a 1] (+ x a))`**, rather than **`(with (a 1) (+ x a))`**, as it makes patent that **`(with..)`** and **`(+..)`** are executable statements and **`[a 1]`** is just a list used as data, which should not be executed. This style is particulary useful for beginners, who are often confused by the implicit evaluation of lisp arguments, leading to obscure errors when a list is inadvertently evaluated.

Locators
#[...]

shorthand for get and put calls

#[*object a b c* **]**
(setq #[*object a b c* **]** *d***)**

The two forms are respectively equivalent to:

```
(getn (getn (getn object a) b) c)
(put (getn (getn object a) b) c d)
```

Locators are more readable, terse and efficient. **`#[a b]`** means the “location” of the “field” **`b`** in **`a`**.

```
#['(a b c) 1]                    → b
#[#(a 1 b 2) 'b]                 → 2
(setq l '(a b c))
(with (#[l 1] "B") (print l))    ⇒(a B c)
```

As a convenience, locators can be used as a function with an argument to put objects in them:

```
(put l 'x 1) ↔ (setq #[l 'x] 1) ↔ (#[l 'x] 1)
```

A shorthand for the setq form can be used with a pseudo infix notation with :=

```
#[a b c := d] is equivalent to: (setq #[object a b c] d)
```

`in-line functions`
`{...}`

lists evaluated at parsing time

{ *function-call* **}**

If you write a list surrounded by braces (**{}**) instead of parentheses, the list is evaluated and replaced by the result of the evaluation as soon as it is read. this can be seen as the opposite of the quoting mechanism. This feature, together with the **^** function (page 68) (which returns the object whose address in memory is given by the second argument) is used by Klone to print some types for which there exist no practical printed representation, and to be able to re-read this representation to return the same object.

```
(coerce (list) Hashtable)⟶ {^ Hashtable 0x4f740}
```

CL-diffs: This implements the functionality of the # read-macro character in lisp, but in a cleaner and simpler way.

Warning: In line functions should not call read or any other parsing functions (load) since the parser is not reentrant in this version of Klone.

NOTE: if the flag **`*quote-inlines*`** is set, the inline functiona are NOT evaluated at parse time, they are returned "as is" as objects with the type **`Collection`**, a subtype of **`List`**. Useful to implement Klone preprocessors in Klone.

```
(type-of {+ 1 2})        ⟶ Int
(setq *quote-inlines* t)
(type-of {+ 1 2})       Æ⟶ Collection
```

1.3 Common functions

Klone has some functions which can be applied to objects of different types and which perform actions relevant to this type, such as print. We list here these main functions, which actually represent the kernel of the interpreter, which can be thought of a collection of types implementing the following methods:

`eval`

evaluate expression CL p490

(eval *form* **)**

This function evaluates *form* in the current environment. Note that there is a double evaluation, since the argument is evaluated by Klone before being evaluated by eval.

Eval is the kernel of the Klone system. Evaluation of *form* is done in the following way:

- Lists are evaluated by evaluating the first element of the list, and applying it as a function on the rest of the list which becomes the arguments passed to the function. Only Functions can be applied.
- Atoms return their current value.
- QuotedExprs return the expression they point to for quote, perform conditional quoting for backquote, and trigger an error for comma and comma-arobas.
- Active values return the value computed by their associated “get” function.
- All other types evaluate to themselves.

```
(eval (append '(+ ) '(1 2))))        ⟶ 3
```

print

print klone objects

print prints the external representation of any Klone object. See "? print" on page 51. and the associated functions to get the complete list of Input/Output facilities under Klone.

get
put
delete
insert

handle elements of complex types

(get *object key [default-value]* **)**
(put *object key value* **)**
(delete *object key* **)**
(insert *object key value* **)**

This is the generic way of accessing elements of complex types, i.e.:

- elements of a list, property list or hashtables
- characters of a string (or subtypes: atom, constants, keywords, active)
- slots of ExOs (EXternal Objects)
- internal fields of various other types...

get returns the value of the element indexed by **key** in **object**, or if no element was present the evaluation of **default-value** (or if none was given, triggers the error **Errors:NoElement**)
put sets the **value** associated with **key** in **object**, and returns the **object**
delete removes the **key** and associated value (if any) from **object**, and returns the object
insert inserts a new value associated with key in object, without removing the old value like put.

The precise semantics of put, get , delete and insert will be detailed in each section on the functions operating on each types

getn

get-or-nil

(getn *object key* **)**

Is equivalent to the call: (get *object key* ()), and is defined for backward-compatibility purposes. It is also a little more more efficient than get.

copy

create a copy of an object

(copy *object* **)**

Returns a duplicate of *object*. This function is necessary to avoid all side-effects on objects physically modifiable by functions such as **put**.

+

generic addition

(+ *{object} ** **)**

Returns the "addition" of objects, whatever this means based upon the type of the first object. This function is used to concatenate strings and lists (see + on sequences page 36), add numbers (see arithmetic operations page 33), and should be used to code the union of application-defined types.

(+) returns **()**.

length

length of a composite object

(length *object* **)**

Returns the “length” (number of elements in most cases) of a Klone object.

1.4 Types

Types in Klone are stand-alone objects. See Types, Section 1.1 on page 7.

type-of

type of object <u>CL 65</u>

(type-of *object* **)**

Returns the actual type of the object as an object of type Type.

```
(type-of 'a)                    → Atom
(type-of Type)                  → Type
(type-of 'Type)                 → Constant
(defclass Point () (x y))
(type-of (type-of Atom))        → Type
```

typep

test if object is of type <u>CL p96</u>

(typep *object* *type* **)**

This function returns the object type if *object* is of type *type* , or a subtype, otherwise nil is returned. Base Klone types are defined in Section 1.1 on page 7. Types are first-class objects, but the symbols representing the types are defined to have these types as values, and moreover the symbols are accepted as types too.

<u>**NOTE**</u>: An object can be “of” more than one type, since one type can include another.

```
(typep 1 Number)                → Int
(typep 1 Int)                   → Int
(typep 1.0 Int)                 → ()
(typep typep Function)          → Subr
```

type can be **()**, in which case **typep** succeeds and returns the type of the object, acting like **type-of**.

subtypep

test if type is a subtype of another <u>CL p97</u>

(subtypep *subtype* *type* **)**

Returns *type* if *subtype* is inherited from *type* . Works either with primitive types or External Objects.

```
(subtypep Constant Atom)        → Atom
(subtypep Point Object)         → Object
(subtypep List Hashtable)       → ()
```

type-name

external name of a type

(type-name *type* **)**

Returns the type name (atom) of a given type.

```
(type-name (type-of 1))            → Int
```

coerce

type conversions CL p63

(coerce *object destination-type* **)**

Objects of one type can be ***coerced*** (i.e. converted) into another type by the use of the **coerce** function. This is not an in-place conversion, coerces returns a copy of the *object*, which has been used to create a new object of the type *destination-type.*

```
(coerce 1 Real)                    → 1.0
(coerce 1.0 Int)                   → 1
(coerce 113 String)                → "113"
```

Note: coercion can also be made by applying the destination-type to the object, this form being easier to read, the previous example can be written:

```
(Real 1)                           → 1.0
(Int 1.0)                          → 1
(String 113)                       → "113"
```

Current available coercions are:

Objects \ Types	Function	Number	Int	Real	String	Atom	Stream	List	Hashtable
Function								lambda define	
Int				convert	itoa				
Real			integer value		itoa				
String		atoi	atoi	atoi		intern	open on string	explode chars	
Atom					print-name				
Stream					whole file				
List					implode chars				create hashtabl
Hashtable								makes P-List	

Grey squares means that the coercions is the identity (returns the same object), and blank squares means that no coercion is defined and thus generate an error.

Here are the details of the different conversions:
convert: just changes the type (used to convert into a subtype).
itoa: generates a string representing the printable representation of the number, just like printing the number to a string.
atoi: generates a number (Int or Real) as represented in the string, just like reading the number from the string.
integer value: truncates a real to its integer value.
lambda define: gives the lambda call (list) used to define the function. This is the recommended way to obtain the definition of a Klone function.
intern: does an intern (see **intern** page 28) of the string (creates a symbol having the string as its external name, or returns the already existing one).
open on string: creates (see **open** page 49) a new stream of type **:string** on the string in **:io** mode. The stream is opened on a **copy** of the string, writing on the stream will thus **not** modify the string.
explode chars: returns the list of characters (ascii codes as integers) composing the

string.
print-name: returns the printable name of than atom as a string.
whole file: read all the file till EOF, and returns the read buffer as a single string.
implode chars: creates a string from a list on integers interpreted as ascii codes.
create hashtable: create hashtable from a p-list (see “Hash Tables” on page 41).
makes p-list: dumps contents of a hashtable as a p-list. This is the way to list the contents of hash tables.

Implicit coercions: A convenient shortcut to the coerce notation is to execute the destination type to perform the conversion on the argument. This way, the two following expressions are equivalent:

`(coerce 1 String)` ⇔ `(String 1)`

Note: if *destination-type* is `()`, then coerce returns the object untouched.

`types-list`

list of all defined types

`(types-list)`

Returns the list of all defined Klone types, primitives ones and application-defined ones.

`type-replace`

physically change the type of an object INTERNAL EXPERT

`(type-replace` *object type* **`)`**

Physically changes the type of an object. This is a very unsafe function, as no sanity check is performed. Should be used to convert between similar types such as between FExpr and Expr only. Changing types carelessly can lead to crashes of klone and the embedding application.

1.5 Functions

Functions are of 5 subtypes:

Subrs are C-coded functions which evaluate their arguments.
FSubrs are C-coded functions which do not evaluate their arguments (also called *special forms* in Common Lisp).
Exprs are klone-defined functions which evaluate their arguments.
FExprs are klone-defined functions which do not evaluate their arguments.
Macros are klone-defined macros.

Functions have two fields than can be accessed via **`get`** and **`put`**:

lambda which is the lambda list declaration of the parameters, or just a number describing the arity for Subrs and FSubrs (-1 meaning a Subr with variable number of arguments).

body which is the list of statements that are executed (via progn) when calling an Expr, FExpr or Macro, or a number which is the address of the C function executed for Subr and FSubr.

Note: This access allows to redefine functions very efficiently, but can be **very** dangerous. No check is made to see if the arity is kept coherent between the lambda and the body part, for instance, and so the use of put on functions should be reserved to experts.

Note: If you just want to get access to the definition of a function just **`coerce`** it into a List.

Functions can be cloned by the **copy** function. Use of copy, get and put of functions allows to dynamically create new Subrs (or any other functions) in a very flexible way at run-time.

Two functions are equal if their external representations (as a lambda declarations in lists) are equal

```
(= (lambda (x) (+ 1 x)) (lambda (x) (+ 1 x)))→ t
(= (lambda (x) (+ 1 x)) (lambda (y) (+ 1 y)))→ ()
```

The list evaluation process:
As with all traditional lisps, a function is called when evaluating a list of the function and its arguments, such as **(foo bar gee)**. In this case:

- foo is evaluated recursively (up to 10 levels, where it triggers an error to avoid infinite recursion) until Klone founds an executable object (a function, active or number)
- Then, if the function evaluates its arguments (is an **Expr** or a **Subr**), the arguments are evaluated in **left-to-right** order (**bar** then **gee**)
- Then the function is applied to arguments and the result returned

<u>**Note**</u>: executing a number is a shorthand for get/put with this number as key:

```
(5 obj)          ⇔     (getn obj 5)
(5 obj foo)      ⇔     (put obj 5 foo)
```

defun

define a Klone function CL p76

(defun *function-name lambda-list [documentation] {form} ** **)**

The use of **defun** is the usual way to define a new function. It defines *function-name* to be the name of a function using the parameters declared in *lambda-list* and the body *form* s. When this function is applied, the parameters will be evaluated before being bound to the parameters.

This function sets *function-name* to point to the lambda expression **(lambda** *lambda-list form** **)**. See the definition of **lambda** (page 19) for more details on the *lambda-list* parameters. Thus,

```
(defun symbol (parameters) body)
```

is equivalent to:

```
(setq symbol (lambda (parameters) body))
```

```
(defun zero ()
   " Just a simple example with a \"documentation string\" ."
   0)
```

The famous Fibonacci function is written:

```
(defun fib (n) (if (< n 2) 1
   (+ (fib (- n 1)) (fib (- n 2)))))
```

defunq

define a Klone special form

(defunq *function-name lambda-list [documentation] {form} ** **)**

defunq is identical to **defun**, but the arguments are not evaluated before passing them to the function.

<u>**Warning**</u>: defunq allows to perform more easily and efficiently some tasks that could have been handled by macros, but there is a potential danger of name conflicts if you pass as a parameter the same symbol as the parameter itself, making the value of this

symbol inaccessible within the body of the defunq. It is therefore recommended that either you define with defunq only symbol-handling functions (which do not evaluate them), or that you use peculiar enough parameter names on all parameters you are going to evaluate in the body of the defun, so that no name conflict can occur.

```
;; example of a name conflict on N
(defunq incr (N) (set N (+ (eval N) 1)))
(setq N 1)
(incr N)                 ⇒ ERROR, since (eval N)→N
;; recommended definition (uses the function name as package)
(defunq incr (incr:N) (set incr:N (+ (eval incr:N) 1)))
```

defmacro

define a Klone macro CL p195

(defmacro *macro-name lambda-list [documentation] {form} ** **)**

The use of **defmacro** is the usual way to define a new macro. It defines *macro-name* to be the name of a macro using the parameters declared in *lambda-list* and the body *form* s.

This function sets the symbol *macro-name* to point to the lambdam expression **(lambdam** *lambda-list form** **)**. See the definition of **lambdam** (page 19) for more details on macros and the lambda-list parameters. Thus, the following call:

```
(defmacro symbol (parameters) body)
```

is equivalent to:

```
(setq symbol (lambda-macro (parameters) body))
```

```
(defmacro inc (var)
    `(setq ,var (+ 1 ,var)))
```

Macros do not suffer from the potential danger of name conflicts that defunq bears, but are a bit less efficient.

lambda
lambdaq
lambdam

lambda expressions CL p76

```
(lambda ({required-parameter} *
            [&optional {o-parameter | (o-parameter initial-value )} *]
            [&rest r-parameter]
            [&whole w-parameter]
            [&key {keyword
                 | (keyword initial-value )
                 | ((keyword variable ) initial-value )
                 } *]
            [&allow-other-keys]
            [&aux {variable | (variable initial-value )} *] )
  [documentation]
  {form} * )
```

lambda is the basic form to define an anonymous function, using lambda expressions (see also **defun**, page 18). In Klone, a lambda expression is a first class-object[1].

The *lambda-list* , the list which is the first argument to lambda specifies the parameters. When a function is called, its arguments are matched to the parameters of the corre-

1. Many traditional Lisps implement lambdas as lists whose first element is the symbol lambda.

sponding lambda list. The following description of lambda lists also apply to **defun**, **defunq** and **defmacro**.

If *required-parameter* s are specified, the corresponding arguments must be present when the lambda expression is applied.

A lambda list can use **markers**, starting with **&**, defined as follows:

&optional optional parameters; if these arguments are not used, they are bound to *initial-value* , if present, or nil otherwise.
&rest rest parameter; when all required and optional parameters are bound, then any following arguments are made into a list and are bound to the rest parameter *r-parameter*.
&whole whole parameter; the whole non-evaluated call list is bound to the *w-parameter* .
&key keyword parameters; these are arguments that consist of pairs with the named keyword (prefixed by :) and its value. The value is bound to a variable which is the keyword stripped from its : prefix, or *variable* if provided. The keyword can be given with or without the **:** prefix.
&allow-other-keys if present, allows to give other keywords than specified when calling the function, the default behavior being to signal an error of type Errors:Invalid-Keyword.
&aux local variables, not really parameters (this feature is equivalent to **let*** or **with**, but more readable and efficient).

For all optional parameters (**&optional**, **&key**, and **&aux**), if an *initial-value* is specified, it is evaluated and bound to the parameter if no value is passed for this parameter to the function, otherwise nil is used.

Warning: the markers must occur in the above order.

```
((lambda (a b) (+ a b)) 4 5)             → 9
                                ;; Use of &optional and &rest parameters
((lambda (&optional (a 2) (c 3) &rest x)
   (list a c x)))                        → (2 3 ())
((lambda (&optional (a 2) (c 3) &rest x)
   (list a c x)) 6)                      → (6 3 ())
((lambda (&optional (a 2) (c 3) &rest x)
   (list a c x)) 6 4)                    → (6 4 ())
((lambda (&optional (a 2) (c 3) &rest x)
(list a c x)) 6 4 5 6)                   → (6 4 (5 6))
                                ;; Use of &key parameters
((lambda (a b &key c d)
   (list a b c d)) 1 2)                  → (1 2 () ())
((lambda (a b &key c d)
   (list a b c d)) 1 2 :d 8)             → (1 2 () 8)
((lambda (a b &key c d)
   (list a b c d)) 1 2 :c 6 :d 8)        → (1 2 6 8)
((lambda (a b &key c d)
   (list a b c d)) :a :b :c :d)          → (:a :b :d ())
```

lambda creates an **Expr** object which evaluates all its arguments before binding them to the parameters in the lambda list (except **&whole** if present)

lambdaq creates a lambda which does not evaluates its arguments (**FExpr**).

lambdam creates a macro (**MExpr**), whose arguments passed to it are not evaluated, bound to the parameters, and the body is evaluated once to obtain a more complex form, the ***expansion***, which is then evaluated.

```
(setq x 1)
((lambdam (p) `(setq ,p (+ ,p 1))) x)
x                                        → 2
```

The call to lambdam defines a macro which is then applied to the variable x. It produces the expansion (setq x (+ 1 x)), which is evaluated. Note the use of the ***backquote*** character ` to build the expansion, this is how most macros are defined .(see ` page 22)

&rest and **&key** can be combined, and in fact must be combined to make use of the **&allow-other-keys**, with the Common Lisp restriction that the list of arguments stored in &rest must be a valid keyword description, i.e:

```
(defun foo (&rest args &key mine &allow-other-keys) ...)
(foo :mine 1 :other 2)              ; ok
(foo :mine 1 :other 2 3)            ; ERROR!
```

CL-Diffs: in Common Lisp, neither lambdaq nor lambdam exist, and the **&whole** marker can only be used in defmacro. Note that lambdas in Klone are first-class objects, unlike traditional lisp systems where they were just lists whose first element were the symbol **'lambda**

apply

apply a function to a list of arguments CL p145

(apply *function* *list-of-arguments* **)**

Executes the *function* on the arguments contained in the *list-of-arguments* .

Unlike Common Lisp, *function* can be a lambda, lambdaq or even a macro.

```
(apply + '(1 2))         → 3
```

There is no **funcall** function in Klone, since symbols have only one value, whereas in Lisps, **funcall** was needed to access the *function* as a function as **apply** used it as a variable.

Note: Do not build yourself a list and evaluate it, as you may end up evaluating twice the arguments for Exprs, use **apply** instead.

```
;; this does not work, as its evaluates twice its argument 'a
(defun funcall (function &rest arguments)
    (eval (+ (list function) arguments)))
(funcall length 'a)     → Error: undefined variable: a

;; this works
(defun funcall (function &rest arguments)
    (apply function arguments))
(funcall length 'a)     Æ→ 1
```

1.6 Control Structure

'
quote

prevent evaluation of expression CP p115

'*form*
(quote *form* **)**

This function returns the form without evaluating it.

```
(+ 1 2)                  → 3
'(+ 1 2)                 → (+ 1 2)
```

If you want the opposite of quoting (forcing evaluation at read time), use in-line functions (see page 13).

Note: Quoting is perhaps the most basic source of confusion for lisp beginners. In most scripting languages, such as the Unix shells or scripting languages such as TCL, quoting is implicit, i.e. writing a symbol **foo** gives you the symbol, if you want the value you must explicitly say so by some operation, **`$foo`** in shells. In Klone as in all lists, by default all arguments are first evaluated and you must take extra steps to prevent it: **`'foo`** is the symbol, but **`foo`** is its value.

`` ` ``
`,`
`,@`

prevent evaluation with possible escapes CL p527

`` `(x ,y ,@z) ``

This function creates a copy of the specified list, using the specified list as a template for creating the copy, as follows:

- Items prefixed by `,` (comma) are evaluated;
- Items prefixed by `,@` are evaluated and must return a list which will be destructured (i.e. all elements will be inserted as top-level elements in the copied list);
- All other items are copied without being evaluated.

This construct is useful for defining macros (see defmacro page 19).

```
(setq x '(13 14))
`(a b ,x c)                          → (a b (13 14) c)
`(a b ,@x c)                         → (a b 13 14 c)
```

`catch`
`throw`

exception handling CL p187

`(catch '`*tag {form} **** `)`**
`(throw '`*tag {form} **** `)`**

These functions provide a powerful mechanism for exiting from a complex process, with the use of dynamic non-local gotos.

The catch function evaluates the *form* s in the same way as for **`progn`** but if a **`throw`** with the same *tag* occurs during the *form* s evaluation, the evaluation is aborted and **`catch`** returns the evaluation of the forms passed as argument to **`throw`** instead.

The *tag* may be defined as the target for transfer of control. It matches a catch with a throw. Tags are compared with the eq predicate, so symbols can be used (numbers or strings are illegal). Several catch/throw combinations can be used with different tags. A throw must have a corresponding catch -if a throw is encountered with an unknown tag, an error is signalled.

See also unwind-protect (page 23) for controlling a throw from an inner block.

```
(defun foo (x)                       ; catches error
  (catch 'my-tag (bar x)))

(defun bar (y)
  (if (= y 0)                        ; jumps to error
    (throw 'my-tag (print "Invalid argument"))
    (gee x)))
(foo 0)                              ⇒ Invalid argument
```

<u>**Note**</u>: **`(catch 'ALL`** *forms* **`)`** catches throws to **any** tag done in *form* s.

`*last-caught-tag*`

the tag being caught

variable

This variable holds the last tag (symbol) caught by a **`catch`** has been done. Very useful in conjunction with the **`(catch 'ALL`** *forms* **`)`** form above. It is not set in the protected instructions of an **`unwind-protect`** call, however. It is initialized to **`()`**.

If *forms* do not throw to a tag, catch does not modify the value of **`*last-caught-tag*`**, which is thus untouched.

```
(catch 'ALL (+ 2 "a")) ⇒ *last-caught-tag* is ERROR
(catch 'ALL (+ 2 3))    ⇒ *last-caught-tag* is still ERROR
(setq *last-caught-tag* ())
(catch ‘ALL (+ 2 3))   ﬁ *last-caught-tag* is ()
```

unwind-protect

guarantee execution of code even in case of throws or errors CL p188

(unwind-protect *protected-form {form} ** **)**

This function returns the evaluation of protected-form, but ensures that if there is an attempt to exit by an inner call to **throw** (see page 22), each form will be evaluated in sequence before obeying the throw.

```
(setq f (open “(+ 1 2)(+ 2 3)” :type :string))
(catch 'EOF

 (unwind-protect
 (while t (print (read f)))
 (print “EOF encountered”)
))                          ⇒ (+ 1 2)(+ 2 3) EOF encountered
```

dolist

iteration through the elements of a list CL p 169

(dolist (*variable list [result]* **)** *{form} ** **)**

This function successively assigns each element of the *list* to *variable* and evaluates each *form* in it. Then *result* is evaluated and its value is returned. If *result* is omitted, nil is returned.

```
(dolist (x '(0 1 2))
    (print x))          ⇒ prints: 012
                        → and returns nil
```

Note: This function is the best way to scan lists. It is more efficient and more readable than **map** (see page 37). Note that you can easily build lists of results just like **map**, for instance by the following code:

```
;; build a list of squares from a list of values: numbers
;; map version: (slow)
(map List (lambda (n) (** n 2)) numbers)
;; dolist version (faster, recommended)
(with (result (list))
    (dolist (n numbers) (put result -1 (** n 2)))
    result)
```

dolist works also on strings, in which case the variable is set to all the characters of the strings as their ASCII number.

WARNING: as with **dohash**, the list should **not** be **modified** (elements added or deleted) in the body (forms) of dolist, otherwise unpredictable results could happen. If you muist modify the list, scan a copy, e.g:

```
;; delete items in list L satisfying predicate P
(dolist (item (copy L))              ; NOT: (dolist (item L)
    (if (P item) (delete L (seekq L item))))
```

dotimes

loop from 0 to n-1 CL p169

(dotimes (*variable countform [result]* **)** *{form} ** **)**
(dotimes *countform {form}** **)**

This function operates as follows:

1. first, evaluate *countform* which must be an integer
2. then successively assign integers from 0 to *countform* -1 to *variable* . For each iteration, the instructions of the body of each *forms* are executed.
3. The result of **dotimes** is the evaluation of *result* . If *result* is omitted, the result is nil.

Variable is local to the dotimes body and is reset to its previous value on exit.

If the value of *countform* is evaluated to zero or a negative value, then the instructions are not executed.

The second form can be used if you do not need to know the number of the iteration in the loop, and is faster (useful for benchmarks for instance)

do
do*

General iteration construct CP p164

(do (*{* **(***variable [initial-value [step]]* **)** *} ** **)**
(*end-test {results} ** **)** *{form} ** **)**

This function provides a generalized mechanism for iteration with an arbitrary number of indexes.

First, *variables* are bound to their *initial value* (if initial-value is omitted, the default is nil). binding is done in parallel for **do** and sequentially with **do***, just as for **let** and **let***.

Then the function loops, operating as follows:

1. evaluate *end-test* . If this predicate is true, and *results* is specified, evaluate *results* and return the last evaluation. If *results* is not specified, return nil.
2. evaluate each *form* .
3. evaluate the *step* expressions, from left to right; each result is bound to the corresponding index variable. If *step* is omitted, the variable is unchanged.

```
;; Equivalent to (dotimes (i 3) (print i ","))
(do ((i 0 (+ i 1)))
    ((>= i 3))
    (print i ","))
```

eval

evaluate expression CL p490

(eval *form* **)**

See “eval” on page 13.

identity

no-operation

(identity *form* **)**

identity can be seen as a medium term between eval and quote. identity just returns its argument, which is evaluated as it is a Subr, but do not evaluate it twice like eval. Useful as a lambda value in some cases.

exit

terminate the application

(exit *[return-code]* **)**

This terminates Klone, and the execution of the application embedding it. The unix process embedding klone will return the integer *return-code* , if present, or **0** which means “successful completion”.

if

multi-way conditional CL# p157

(if *{test form} *[else]* **)**

This function is similar to **cond** but with a level of parentheses suppressed. It executes the *form* following the first true *test* , or the *else* part (if present) if no previous condition is true. Whereas **cond** is the true Common-lisp compatible function, **if** is more efficient.

CL-Diff: In Common Lisp, only one test is allowed.

```
(defun fib (n)
 (if
   (= n 0) 1

   (= n 1) 1
   (+ (fib (- n 1)) (fib (- n 2)))))
```

cond

multi-case conditional test CL p158

(cond **(** *test {form} *) ** **)**

This function evaluates in **progn** fashion the *form* s that are associated with the first *test* that returns a true condition.

The function returns the evaluation of the last of these forms. If no forms are present, then the single (and necessarily non null) value of the test is returned. When every test produces nil, the cond function returns nil.

```
(defun fib (n)
  (cond
   ((= n 0) 1)
   ((= n 1) 1)
   (t (+  (fib (-n1))
          (fib (- n 2))))))
```

NOTE: For efficiency, use **if** (see page 25) rather than **cond**.

let
let*

declare local variables CL p148

(let **(***{variable|* **(** *variable initial-value* **)** *} ** **)** *{form} ** **)**

This function operates as follows:

1. evaluates all the *initial-values* and binds them to the corresponding variables. the binding is do in parallel for **let**, and sequentially for **let***.
2. sequentially evaluates each form.

If no initial value is associated with a variable, it is initialized to nil. The function’s returned value is the one returned by the last *form* .

```
(setq a 5)
(let ((a 3) b (c (+ a 1)))
   (print a " " b " " c))            ⇒ 3 () 6
```

```
(let* ((a 3) b (c (+ a 1)))
   (print a " " b " " c))              ⟹ 3 () 4
(print a)                              ⟹ 5
```

This shows that the value of 3 for **a** is local to the scope of the **let** function. The last line above is outside this scope, so the value of 5 corresponds to the **setq** function, also outside the **let** scope.

Note: **let*** is faster than **let**, you should use it when the order of binding is not relevant. Note that **with** (see page 26) is even faster, though.

---

progn

sequence of instructions CL p147

(progn *{form} ** **)**

This function evaluates each *form* and returns the result of the last evaluation. **progn** is useful in places where more then one Lisp instruction is expected, such as in the then fields of the **if** instruction. It is much like the “begin-end block” of other languages.

---

setq
set
:=

assign a value to a variable CL p121

(setq *variable value* **)**
(set *object value* **)**
(:= *variable value* **)**

This is the assignment function of all Lisp dialects. In the first form (**setq**), the first argument is not evaluated, whereas in the second (**set**) it is.

Both functions evaluate their second argument and set the value of the first argument to the resulting value. Setting active values does not actually modify their value, but calls a predefined function on the value that may perform more side-effects than just changing a value.

:= is a synonym for setq.

```
(setq b 'c)
(setq a (+ 1 2))
(set b 4)              ⟹ a=3 and c=4
(:= c (+ 2 3))
```

---

while

loop while a condition is true

(while *test {form} ** **)**

This function executes each *form* in sequence until *test* becomes nil. The function always returns nil.

Note: (while t forms...) is specially optimized for fastest operation on infinite loops

CL-Diff: **while** is not a common lisp function, but can be used as a marker in loops.

---

with
with-eval

declare local variables

(with (*{variable value} ** **)** *{form} ** **)**
(with *context {form} ** **)**
(with-eval *expression {form} ** **)**

These functions are constructs used to declare and initialize *variables* that are local to a group of instructions (*forms*). Active-values and functions are handled as expected, resetting their initial value after the execution of the body of the **with**. The *values* are evaluated sequentially. **`with`** is equivalent of the traditional **`let*`** function (see page 26), with one level of parentheses stripped, and for this reason is more efficient than **`let`**, but less readable for lisp users.

The argument *context* is a list of variables and associated values (a Property List) that can be re-used in many **`with`** functions.

`with-eval` first evaluates the *expression* and then uses it as a context, so that the two following three **`with`** calls are equivalent:

```
(with (a 1 b 2) c)
(with-eval (append '(a 1) '(b 2)) c)
(setq a-context '(a 1 b 2))
(with a-context c)
(with-eval a-context c)
```

Due to the structure of the Klone interpreter, **`with`** also works with active-values and functions. For example, the following call can be made to move the window “my-window” to the upper left corner of the screen, with the X Toolkit interface:

```
(with (widget my-widget move-ul
    (lambda () (XtMoveWidget 0 0)))
  (move-ul))
```

`with` can also evaluate its context if the context is prefixed by a comma, with acting thus like it embedded its context in a backquote environment.

```
(with ,(+ main-context local-context) some-code...)
```

Note: the more traditional lisp functions **`let`** and **`let*`** are provided as autoloadable Klone displacing macros. try to use let* but not let if you can as let is much slower than let* or with, and if possible use the &aux constructs in function declaration which is faster than both and more readable. As a reminder, let* evaluates its arguments sequentially, like with, whereas let evaluates them in parallel, e.g:

```
(setq x 1) (setq y 2)
(let ((x y) (y x)) (vector x y))     → [2 1]
(let* ((x y) (y x)) (vector x y))    → [2 2]
(with (x y y x) (vector x y))        → [2 2]
```

1.7 Symbols

Klone symbols are monovaluated and have dynamic scope. See Symbols, page 10.

Keywords are symbols prefixed by a colon **:**, and evaluate to themselves. Note that keyword are not equal to their corresponding symbol or string:

```
(= :foo 'foo)          → ()
(= 'foo :foo)          → ()
(= 'foo "foo")         → ()
```

Any symbol name can be created by intern, even if it is not parsable, but no symbol can begin with a colon, the corresponding keyword is created instead.

```
1+                       ;; is read a 1 followed by +
(type-of (intern "1+"))→ Atom
(type-of (intern ":1")) → Keyword
```

Coercing a keyword into an atom returns the corresponding atom, and vice-versa.

```
(Atom :foo)            → foo
(Keyword 'foo)         → :foo
```

boundp

test if an atom has already been defined CL p120

(boundp *symbol* **)**

This function returns the (evaluated) *symbol* if it has been defined, otherwise nil is returned. **boundp** is true if the variable named *symbol* has a value.

```
(setq foo 1)
(boundp 'foo)           → foo
(boundp 'bar)           → ()
```

defconstant

define a constant CL p86

(defconstant *variable initial-value [documentation]* **)**

This function declares a variable to be a constant. Such a constant cannot be referenced in a **setq** nor a **let** function. *documentation* is a string associated with the constant and may be retrieved with the **documentation** (see page 28) function. *variable* needs not to be quoted, and the initial value is returned.

documentation

manage symbols documentation strings CL# p695

(documentation *symbol [string]* **)**

If *string* is not specified, this function retrieves the documentation for the specified *symbol*. nil is returned if the documentation does not exist. If *string* is specified, it is associated with *symbol*. An error occurs if the symbol is not defined.

Documentation may be defined with the defvar, defconstant, defun and defmacro functions.

```
(defun zero()
   "Complex function that returns zero!"
   0 )
(documentation zero)--> "Complex function that returns zero!"
(documentation zero "Simple function that returns zero!" )
```

CL-Diff: the Common Lisp documentation function can only retrieve a documentation string, but cannot set or modify it (i.e. *string* is not allowed in CL).

Note: the use of documentation strings uses up memory. You may want to use the **smartdoc** (see Section 1.23.5 on page 77) facility for more space efficiency.

intern

create a symbol from a string CL p266

(intern *string* **)**

This function creates and returns the symbol whose name is *string*. If a symbol with the same name is found, it is returned. The function is useful for creating atoms with imbedded special characters, such as ' or blanks, or for automatically generating symbols, as in the following example:

```
;; example of intern use: the gensym function
(setq gensym:seed 0)
(defun gensym ()
   "gensym: generates a unique symbol each time it is called"
   (setq gensym:seed (+ gensym:seed 1))
   (intern (+ "gensym::" (coerce gensym:seed String))))
```

If the *string* begins by a colon `:`, then a keyword is built. there is no way to build a non-keyword symbol beginning with a colon.

makunbound

remove a symbol definition CL p123

(makunbound *symbol* **)**

This function undefines the (evaluated) *symbol*, so that if the function **boundp** (see page 28) is applied on it, nil will be returned. Storage for the symbol is reclaimed if it is not used elsewhere.

```
(setq a 1)
(boundp 'a)                    → a
(makunbound 'a)
(boundp 'a)                    → ()
```

atom-of

atom of symbol

(atom-of *symbol* **)**

Return atom of a symbol: for a keyword, it is the base part (without the leading colon `:`), for aother atom it is the atom itself. For other types, atom-of tries to coerce to atom first.

```
(atom-of 'foo)                 → foo
(atom-of :foo)                 → foo
```

package

current package name CL# p262

string

Klone supports explicit packaging: i.e. when reading a symbol prefixed with '`%`', the Klone reader will replace the `%` character by the current value of **`*package*`**, which must be a string, with a colon '`:`' appended to it. If **`*package*`** is undefined (default) or nil, no expansion is done, the '`%`' char and the following colon are just stripped off.

```
(setq *package* ())                  ; or undefined
'%a                    → a
(setq *package* 'my-package)  ; symbols can be used as strings
'%a                    → my-package:a
'%:a                   → my-package::a
'b                     → b         ; no implicit packaging
```

It is a good programming practice to package all symbols defined in klone files provided as libraries to be used by other programs, with the convention of singe-colon (my-package:a) names being publicly available, whereas double-colon names (my-package::a) being reserved for private use in the package itself, and should not be used elsewhere.

Note: The value of *package* is local to the file being loaded, so you can put a line like:

```
(setq *package* "FooTheBar")
```

at the start of a file, only the symbols read in this file will take into account this setting.

CL-Diff: in Common Lisp, the package system is much more complex, and is implicit, i.e. in the last line of the above examples, **`'b`** would have been read as **`my-package:b`**.

symbol-plist
object-plist

properties associated to a symbol CL p241

(symbol-plist *symbol* *[property-list]* **)**
(object-plist *object* *[property-list]* **)**

Each *symbol* has a property list (see “lists”, page 12) associated to it. **symbol-plist** allows to get its value if no second argument is given, or set it to *property-list* otherwise.

Predefined properties: Klone recognize special properties put on symbols:

- **documentation**: The documentation for symbols is stored under the key **documentation** in the symbol’s p-list.

```
(defun foo (a) "The fabulous foo" a)
(setq *print-readably* t)
(symbol-plist 'foo)     → (documentation "The fabulous foo")
```

object-plist is a similar function that can be available on application-specific types, you can test for it with by calling object-plist on an obect and trapping the **Errors:NoElement** error.

```
(symbol-plist 'a)       →           ( )
(symbol-plist 1)        → No element named plist for 1
                        Error  code: Errors:NoElement
```

do-all-symbols

iterate on all symbols

(do-all-symbols *(variable [result]) forms** **)**

This function successively assigns each defined symbol in the system to *variable* and evaluates each *form* in it. Then *result* is evaluated and its value is returned. If *result* is omitted, nil is returned.

```
(defun oblist (&aux (all (list)))
       (do-all-symbols (symbol all) (put all -1 symbol)))
```

defactive

define a Klone active value

(defactive *name get-func put-func slot-initial-value* **)**

Defines and returns an active value, i.e. a symbol of p-name the string *name* and of type **KloneActive** which will apply the *get-func* function to itself to compute the returned value when evaluated, and will apply the *put-func* function to itself and the new value to implement the **setq** method on itself. Klone active values maintain their own storage, their «**slot**» (otherwise known as a closure), which is initialized to the value of *slot-initial-value* .

The value of the slot can be accessed by the generic **get** and **put** functions with the key **slot**, and the *get-func* and *put-func* by the keys **get** and **put** respectively.

If *get-func* is (), evaluating the active value just returns the contents of its slot, and if *put-func* is (), **setq**-ing the variable to a value just sets the contents of the slot to the value.

defactive do not evaluate name but evaluate its other arguments, like **setq**, **defconstant**, etc...

```
;; tracing changes to variable circle-color
(defactive circle-color
    ()                          ; natural get
    (lambda (active new-color) ; trace the setqs
        (print-format () "Circle color now: %0\n" new-color)
        (put active 'slot new-color))           ; update slot
    "black")
circle-color                      → black  (natural get used)
(setq circle-color "red")         → red
                                  ⇒ Circle color now: red
(with (circle-color «blue») ())   ⇒ Circle color now: blue
                                  ⇒ Circle color now: black
```

`*:symbol-slot`

physically modifies a symbol INTERNAL

(*:symbol-slot *name* **)**

Creates a pointer to a symbol, an object of type **`SymbolSlot`**. Evaluating this pointer returns a «snapshot» (a copy) of the symbol, while setting it to another symbol physically replaces the symbol with the characteristics (i.e. the actual type) of the new symbol (Atom, Constant, Active, KloneActive), while keeping its natural attributes which are its name and its p-list. This is especially useful to be able to redefine locally a symbol to another type of symbol in the body of a with. Setting a SlotSymbol to nil undefines this symbol.

```
;; we want to define a synonym «stdout» to *standard-output*, which is a built-in C active value
(set (*:symbol-slot "stdout") '*standard-output*)
;; we want to execute the code (foo) while being sure that color is not changed by it:
(with ({*:symbol-slot "color"} ())
    (defconstant color "black")
    (foo))
;; color stopping being a constant outside of the with scope
```

1.8 Predicates (logical operations)

A predicate is a function testing a condition, and returning a false (nil) or true (non-nil) value.

`=`
`equal`

test equality of any two objects CL p107

(= *object1 object2* **)**
(equal *object1 object2* **)**

This function return *object1* if it is the same as *object2* , otherwise nil is returned. Equality of lists is tested by testing the equality of all their elements recursively.

```
(= "abc" (+ "a" "bc"))              → "abc"
(= () (list))                       ÆÆÆÆ→ t
(= '(1 (2 3)) '(1 (2 4)))           → ()
```

Warning: strings are not equal to the corresponding atom.

```
(= "foo" 'foo)                      → ()
```

`eq`

test strict equality of any two objects CL p103

(eq *object1 object2* **)**

This returns true only if the two object are exactly the same, i.e. they are pointers to the same memory location.

```
(setq l '(a b c))
(setq lc '(a b c))
(= l lc)                              → (a b c)
(eq l lc)                             → ()
(eq (list-get l 1) (list-get lc 1))   → t
(eq () (list))                        → ()
```

Note: In Klone numbers are allocated each time they are encountered, so that two **`equal`** numbers are **not** necessarily b:

```
(= 32 32)                           → 32
(eq 32 32)                          → ()
```

```
(setq a 32) (eq a a)          → 32
(eq 2 2)                      → 2 ; small numbers are cached
```

/=

test inequality of objects CL# p293

(/= *object1* *object2* **)**

Is equivalent to **(not (=** *object1* *object2* **))**. Return **t** if objects are different for equal, **()** otherwise.

CL-Diff: In Common lisp, applies only to numbers, but is not limited to two arguments.

<
<=
>
>=

ordering predicates CL p293

(< *object1* *object2* **)**
(<= *object1* *object2* **)**
(> *object1* *object2* **)**
(>= *object1* *object2* **)**

This predicate compares two numbers or two strings and returns **t** if *object1* is strictly inferior (or inferior-or-equal, strictly superior, superior-or-equal respectively), to *object2* ; otherwise nil is returned. Strings are compared alphabetically, with uppercase and lower-case letters being different. If *object1* and *object2* are not both numbers or strings, and error is generated.

```
(< 2 2)                       → t
(>= -3 7)                     → ()
(>= "foo" "bar")              → ()
(< "Code" "abc")              → t
```

CL-Diff: In Common lisp, applies only to numbers, but is not limited to two arguments.

and
or
not

logical AND, OR, and NOT of expressions CL p110

(and *{form} ** **)**
(or *{form} ** **)**
(not *form* **)**

This predicates evaluates each *form* , from left to right. For:

- **and**, if any *form* is evaluated to nil, the function returns nil, otherwise, t is returned. If no argument is specified, the function returns **t**.
- **or**, if any *form* is evaluated to non-nil, the function returns it, otherwise, **()** is returned. If no argument is specified, the function returns **()**.
- **not**, the function returns **t** if *form* evaluates to an empty list, **()** otherwise.

and and **or** only evaluate needed arguments, so evaluating **(and () (setq a 1))** does not modify **a**. This is called ***short-circuit*** evaluation.

compare

ordering comparison

(compare *number1* *number2* **)**
(compare *string1* *string2* **)**
(compare *list1* *list2* **)**

This function compares two numbers or two strings and returns one of the integer values:

- a strictly positive integer if the first argument is greater than the second;
- zero if the first argument is equal to the second;

- a strictly negative integer if the first argument is less than the second.

Comparing two lists does a lexicographic comparison of all elements. Comparing two strings is equivalent to comparing the lists of their characters.

WARNING: Please note that these numbers can be other numbers than just -1 and 1.

```
(compare "abc" "def")          → -3
(compare '(1 2) '(1 2 3))      → -1
(compare '(1 2 3) '(1 2))      → 1
```

NOTE: compare is very useful with the **sort** (see page 40) function.

NOTE: case is significant in string comparaisons. You can use the Klone autoloadable function **compare-nocase** if you want to do a case-unsensitive comparison.

max
min

return the maximum or minimum value

(max *elements* **+)**
(min *elements+* **)**

Return the largest (or the smallest) *element* s as ordered by the **compare** (see page 32) function.

1.9 Numbers

All number functions return a integer if all arguments are integers, but use real arithmetic if an argument is a real, except for the **mod** function.

```
(/ 3 2)          → 1      ;; integer division
(/ 3 2.0)        → 1.5    ;; real division
(mod 17.2 3)     → 2      ;; integer modulo
```

Execution of numbers: An useful shortcut is that executing or applying a number to an object performs a getn on this object with key the number, and if a second argument is given, performs a put with the second argument as news value. This is invaluable to perform fast access to elements of lists.

```
(N object)          ⇔     (get object ())
(N object value)    ⇔     (put object value)
```

+
-
*****
/
mod

arithmetic operations on numbers CL p295

(+ *{number}+* **)**
(- *number* *{numbers} ** **)**
(- *number* **)**
(* *number1* *number2* **)**
(/ *number1* *number2* **)**
(mod *number1* *number2* **)**

+ Adds any number of numbers together,
***** and **/** multiply and divide numbers,
 - substracts the sum of *numbers* to *number* in the first form, or just takes the opposite of *number* in the second form.,
mod gives the modulo (remainder of the division of number1 by numer2).

```
(- 3)          → -3
(- 17 2 3)     → 12
(/ 17 3)       → 5      ; division
```

```
(mod 17 3)        → 2        ; remainder
(mod -17 3)       → 1
```

+ on numbers will silently convert `()` arguments to `0`.

`**`

elevation to power

(** *base-number power-number* **)**

Elevates *base-number* to the power *power-number*, which **must** be an integer.

```
(** 5 2)          → 25
(** 4 -1)         → 0.25
```

`abs`

absolute value CL p303

(abs *number* **)**

Returns the absolute value of *number*. Could be defined as:

```
(defun abs (n) (if (>= n 0) n (- n)))
```

`logand` `logior` `lognot` `logxor` `logshift`

Bitwise operators. CL p357

(logand *number1 number2 +* **)**
(logior *number1 number2 +* **)**
(lognot *number* **)**
(logxor *number1 number2 +* **)**
(logshift *number1 number2* **)**

These operators correspond respectively to the following operations: logical and, logical inclusive or, logical not, and logical exclusive or. These function are called bitwise functions since they perform boolean functions on the binary bits composing the number in parallel. Numbers are ensured to be at least 32 bits long. logshift shifts bits to the left if number2 is positive, to the right if negative

```
(logior 1 4)      → 5
(logand 3 5)      → 1
(logxor 3 5)      → 6
(lognot 5)        → -6
(logshift 8 2)    → 32
(logshift 8 -2)   → 2
```

`*real-precision*`

control the number of digits printed for reals

integer

Sets the maximum number of significant digits after the dot with which real numbers are printed.

```
(/ 1.0 3)         → 0.333333             ;default format
(setq *real-precision* 1000)
(/ 1.0 3)         → 0.33333333333333331          ; max
(setq *real-precision* 2)
(/ 1.0 3)         → 0.33
```

Note: if the global variable **`*print-readably*`** is true, then reals are printed with the maximum possible precision.

incf

increments the value of a variable CL p297

(incf *variable [increment]* **)**

Is equivalent to (setq variable (+ variable increment)). *increment* defaults to 1.

```
(while (> n 0) (incf n -1))
```

1.10 Math functions

Klone math function are simple double precision ones, implementing a subset o the Common Lisp specification which implements lots of functions which work all on both real and complex numbers. In fact, it is a raw interface to the Unix math library.

pi
e
pi
e

mathematical constants CL p307

reals

these names holds the value of the constants pi and e. **pi** and **e** are the Common-Lisp standard names, but for avoiding accidental shadowing by user variables, ***pi*** and ***e*** are provided as synonyms.

```
*pi*              → 3.14159265358979323846
*e*               → 2.7182818284590452354
```

expt

elevation to power CL p300

(expt *base number* **)**

(expt x y) returns x^y

convenience functions are defined as autoloadable Klone functions:

- (**sqrt** x) is (expt x 0.5), i.e : $\sqrt{x}$
- (**exp** x) is (expt e x), i.e: e^x

sin
cos
tan

trigonometric functions CL p304

(sin *radians* **)**
(cos *radians* **)**
(tan *radians* **)**

Returns the appropriate value. In this implementation, **tan** is an autoloadable Klone function.

asin
acos
atan

inverse trigonometric functions CL p305

(asin *x* **)**
(acos *x* **)**
(atan *x* **)**

Returns a value in radians.

sinh
cosh
tanh

hyperbolic functions CL p308

(sinh *x* **)**
(cosh *x* **)**
(tanh *x* **)**

Returns the appropriate value. All of these functions are autoloadable Klone functions.

asinh
acosh
atanh

inverse trigonometric functions CL p305

(asinh *x* **)**
(acosh *x* **)**
(atanh *x* **)**

Returns the appropriate value. All of these functions are autoloadable Klone functions.

1.11 Sequences

Sequences are ordered sets of objects. In this section are listed all functions that can be applied both on lists and on strings (strings are then viewed as lists of characters).

+

concatenate sequences

(+ {string} *)
(+ {list} *)

This function returns a string (or a list, depending on the type of the first element) made by the concatenation of the arguments. All the arguments must be of the same type, and none is physically modified, a new object is created

```
(+ "abc" 'DEF "." "1")          → abcDEF.1
(+ "a" '(1 2))                  ⇒ Errors:BadArgType
(+ '(1 2) (list 3 4))           → (1 2 3 4)
```

+ on strings will try to convert numeric arguments to their string representation, i.e. you can say (+ "foo" (+ n 1)) instead of the normal (+ "foo" (String (+ n 1)))

copy

copy a list or string

(copy *object* **)**

Returns a copy of the list or string. In case of lists, only the top-level list is copied, the elements are not copied.

```
(setq l1 '("a" "b"))
(setq l2 (copy l1))             → ("a" "b")
(eq l1 l2)                      → ()
(equal l1 l2)                   → ("a" "b")
(eq (get l1 0) (get l2 0))      → t
```

length

length of lists or strings CL p393

(length *sequence* **)**

Returns the number of elements of a list or the number of characters of a string. This function is quite fast as the length of a list or a string is stored in the objet and there is no need to scan the string for a terminating NULL byte as in C.

map

execute (map) a function on a sequence CL p395

(map *result-type function list +* **)**

This function applies *function* to the elements of the specified *list* or *lists* returning a list or a string depending on the value of *result-type* .

function must take as many arguments as the number (**N**) of *list* arguments provided. All **N** lists must have the same length (**L**). map returns a list or string of length **L** having as elements the result of calling the function with the **L** elements of the list, if one list is specified, or, for several lists, the first elements of each list, then the second elements of each list, and so on up to **L** (see example below).

result-type determines what is constructed, and is one of the types **List** (build a list), **String** (build a string) or **nil** (build nothing).

If a string is created, it is composed of each ascii code returned by the function, which implies that only non-zero numbers must be returned.

```
;; one input list - return a list:
(map List '(lambda (x) (+ x 95)) '(2 3 4))       → (97 98 99)
;; one input list - return a string
(map String '(lambda (x) (+ x 95)) '(2 3 4))         → "abc"
;; two input lists - return a list
(map List '(lambda (x y) (+ x y) '(1 2 3) '(4 5 6)) → (5 7 9)
```

Note: Due to the implementation of lists in Klone, this is the recommended way (with the **dolist** function, see page 23, which is even more efficient and readable) to do operations on lists, as opposed to the traditional lisp method of processing the car of the list and recursively processing the cdr.

nconc

physically concatenate a list or string CL p419

(nconc *list1 {list} ** **)**

This function concatenates *list1* with each *list* . The function physically modifies and returns *list1* . To concatenate lists without modifying them, see + page 36.

```
(setq x '(a b c))                    → ( a b c )
(nconc x '(d e f) '(g h i))          → ( a b c d e f g h i )
x                                    → ( a b c d e f g h i )
(setq x "abc") (nconc x "123")
x                                    → "abc123"
```

seek

find position of an element in a sequence CL p404

Thi is
(seek *string substring [offset]* **)**
(seek *string character [offset]* **)**

In the first form, the *list* is scanned (with the **equal** predicate). If *item* is present, the function returns its index (number) in the list starting from 0 (or *offset*); otherwise nil is returned.

In the second form, the string is searched for an occurrence of substring. If the substring is present, the function returns the character position at which it starts; otherwise nil is returned.

In the third form, the first occurrence of character in string is returned, or nil if not found.

```
(seek '(a b c) 'a)              → 0
(seek "abcd" "cd")              → 2
(seek "abc" #\f)                → ()
(seek "another a" #\a 1)        → 8
```

position is the Common Lisp function and is also available as a Klone function (slower) for compatibility with Common Lisp. **seek** is the primitive function however, to respect the Klone conventions of having always the set before the element in function arguments. position is just like seek, but with the order of the arguments reversed.

```
(position 'a '(a b c))          → 0
```

---

seekq

find position of an element in a list by eq

(seekq *list item [offset]* **)**

Just like seek (but only on lists) but checks for the element presence with **eq** instead of **equal**.

---

subseq

extract a sub-part from a sequence CL# p393

(subseq *list start [end]* **)**

This function returns the sublist of *list* from the element at position *start* (inclusive) to *end* (exclusive). If *end* is not specified, the end of the list is assumed. Both *start* and *end* are numbers starting at **0**. If no elements exists at the specified positions, their places are filled with nil. Thus **subseq** always returns a list of size *end* – *start* .

The function can also be used with a string, with non-existing elements filled by blanks.

```
(subseq '(1 2 3 4) 2 4)         → (3 4)
(subseq '(1 2 3 4) 5)           → ( )
(subseq () 5 9)                 → (() () () ())
(subseq '(1 2) 10 -8)           → ( )
(subseq "abc" -1 4)             → " abc "
```

CL-Diffs: An error is triggered in common list if start or end are out of bounds for list.

---

replace-seq

replace a subsequence in a sequnce

(replace-seq *list sublist start-pos end-pos* **)**
(replace-seq *string substring start-pos end-pos* **)**

Replaces physically in the first argument all the elements between positions *start-pos* (inclusive) and *end-pos* (exclusive) by the second argument. Returns the modified first arhument.

```
(replace-seq '(1 2 3 4) '(42) 0 3)   → (42 4)
(replace-seq "I mean us!" "you" 7 9) → "I mean you!"
```

1.12 Lists

---

list

construct a list CL p417

(list *{form} ** **)**

This function returns the list formed by the evaluation of its arguments. Without arguments, an empty list[1] (not nil) is created, which can then be expanded later, since nil is not modifiable.

```
(list (+ 1 2) 4 '(6 7))          → (3 4 (6 7))
(list)                           → ( )
(put () 0 'a)                    ⇒ Errors:NoPut
(put (list) 0 'a)                → (a)
```

Warning: be careful not to define local variables named list, as with Klone monovaluation, this would hide the definition of list in the scope of the declaration. A common error is:

```
(defun car-cdr (list)   ;; list is mistakenly overridden
   "returns first element of list and rest in a pair"
   (list                ;; ERROR: not the function anymore
          (get list 0)
          (subseq list 1)))
```

make-list

construct a list of a given size — CL p418

(make-list *size* *[* **:initial-element** *form* *]* **)**
(make-list *size* *[* *form* *]* **)**

This function constructs and returns a list containing *size* elements, where each element is initialized to *form* , if present, or nil otherwise. The first form is here for Common Lisp compatibility, the second form for convenience.

```
(make-list 3 :initial-element 'a)   → (a a a)
(make-list 3 'a)                    → (a a a)
```

get put delete insert

access elements of lists

(get *list key [default-value]* **)**
(put *list key value* **)**
(delete *list key* **)**
(insert *list key value* **)**

If *key* is a number, then the element at offset *key* (first element is at offset 0) is returned, set or deleted. A negative value of *key* means to append *value* to the end of *list* for **put**, to return the last element for **get**, and to delete the last element for **delete**. In **put**, if *key* is greater than the length of list, list is extended to have length *key*, the created elements being initialized to **()** for lists, and to blank characters for strings. **insert** inserts the value at offset, pushing the rest of the elements.

```
(get '(1 2 3) 1)                 → 2
(insert '(1 2 3) 0 0)            → (0 1 2 3)
(put '(1 2 3) 0 0)               → (0 2 3)
(put () 3 1)                     ⇒ ERROR
(put (list) 3 1)                 → (() () () 1)
```

key can also be of a non-numeric type, which means that *list* is treated as a **property list.** (see page 12). key is then an element of the list, and value is the element following it. Comparisons are done via equal, not eq. This does **not** work with **insert**.

1. Klone prints **the** nil list: **()**, and any other empty list with a space: **( )** to distinguish them.

```
(get '(a 1 b 2 c 3) 'b)                →  2
(put (list) "abc" 2)                   →  ("abc" 2)
(delete '(a 1 b 2 c 3) 'a)             →  (b 2 c 3)
```

sort

sort a list in place CL# p408

(sort *list comparison-function*)

This function sorts in place the *list* in increasing order, according to the comparison function. The function named by *comparison-function* is called with two items to compare, and must return an integer. Its value will be negative, null or positive if the first argument is respectively less than, equal to, or greater than the second.

NOTE: A common error is to use predicates such as **<** as the comparison function. This usage is incorrect because the comparison function must return an integer value. Correct comparison functions to use would be **–** or **compare** (see page 32).

```
(sort '("Bordeaux" "Beaujolais" "Bourgogne")
   (lambda (x y) (compare x y)))
                        →  (Beaujolais Bordeaux Bourgogne)
```

Cl-Diffs: In Common Lisp, the *function* is a predicate and must return () if the first item is strictly less than the second one. This incompatibility is justified by performance reasons.

replace-list

physically displace a list

(replace-list *old new* **)**

This replaces all the elements of the *old* list by the elements of the *new* list, thus making *old* a clone of *new* .

```
(setq l '(a b))
(setq l2 l)
(replace-list l '(1 2 3))              →  (1 2 3)
l2                                     →  (1 2 3)
(eq l2 l3)                             →  ()
```

append-to-list
lappend

appends an element to the end of a list

(append-to-list *list element* **)**
(lappend *list element* **)**

This is just equivalent to (put list -1 element), but is faster and more readable, and easier for people accustomed to other lisp dialects. **lappend** and **append-to-list** are equivalent.

delete-nth

delete nth item in a list if present

(delete-nth *list position* **)**

Just like (delete list position), but only if position is a number marking an existing position in the list, otherwise silently does nothing. Very useful to be passed the return value of a seek, which can be () if not found without having to check the value before calling delete.

delete-nth returns the deleted element, or () if none found.

1.13 Vectors

All functions applying to lists work on vectors.

Note: lists are equal to vectors with the same elements:

```
(= '(a 1) #(a 1))          → (a 1)
(eq () #())                → ()
```

vector

create a vector CL p 447

(vector *elements...* **)**

Creates and returns a vector from its elements. Similar to the **list** function.

```
(vector 'a 1)                        → #(a 1)
```

vector!
list!

converts in place (cast) vectors and lists

(vector! *list* **)**
(list! *vector* **)**

Converts in place lists or vectors to vectors or lists. vectors! does not converts the nil list to a vector, but just returns an empty vector.

```
(setq l '(a 1))
(list! l)                  ;; does not change l
(vector! l)                → #(a 1)
l                          → #(a 1)     ; l physically modified
(setq n ())
(vector! n)                → #( )           ; but n is still ()
```

1.14 Hash Tables

Creating hash tables: There is no specific hash table creation function, all hash tables must be built from a property list or nil via the coerce function.

```
(setq my-table (coerce '(a 1 b 2)) Hashtable)) ; a 2-entry HT
(setq my-table (Hashtable '(a 1 b 2)))   ;same, easier to read
(get my-table 'b)                    → 2
```

Two hashtables are equal if recursively all keys and values are equal.

get
put
delete

access elements of hashtables

(get *hashtable key [default-value]* **)**
(put *hashtable key value* **)**
(delete *hashtable key* **)**

get retrieves a value associated to a *key* in a *hashtable* (or the result of the evaluation of *default-value* if *key* was not in *hashtable*, or triggers the error **Errors:NoElement** if no default was given), **put** sets or modifies it to *value*, and **delete** removes the pair *key*, *value* from the *hashtable*.

```
(setq ht (coerce '(a 1 b 2 c 3) Hashtable))
(get ht 'b)                          → 2
(get ht 2)                           → ()
(get ht 2 (+ 1 2))                   → 3
(delete ht 'b)
(coerce ht List)                     → (a 1 c 3)
```

length

number of entries in a hashtable

(length *hashtable* **)**

Returns the number of entries (keys) in a hashtable. This is half the length of the list used to build one.

dohash

apply function on each entry of a hash table

(dohash (*key-var value-var hashtable* **)** *forms** **)**
(dohash (*key-var value-var p-list* **)** *forms** **)**

This function scans the whole hashtable (or p-list) and executes the forms in sequence for each entry after binding the variable key-var to the current key and the variable value-var to the corresponding value. This function always returns nil.
Be warned that the order of the scan is not predictable (except for p-lists)

```
; Print all keys whose value is negative
(setq my_list '(k1 2 k2 -1 k3 -2))
(dohash (key val my_list)                          ⇒ k2 k3
   (if (< val 0) (print key " ")))
```

Note: The more traditional maphash function can be expressed in terms of **dohash** by:

```
(defmacro maphash (func table)
     `(dohash (maphash:k .maphash:v ,table)
         (apply ,func (list .maphash:k .maphash:v))))
```

WARNING: dohash generates an error if the forms try to modify the scanned hashtable. If you must modify it, you **must** perform the scan on a copy of the table or store the deletions/insertions to be performed after the dohash. The only modification allowed is the deletion of the current entry. As with **dolist**, It is forbidden however to modify in any way scanned **p-lists**, as in this case the results are unpredictable.

```
(dohash (k v table)
   (if (= k old-key) (delete table k)))              ; OK
(dohash (k v table)
   (if (= v value) (put table key value)))        ; ERROR!
(dohash (k v (copy table))
   (if (= v value) (put table key value)))            ; OK
```

***:hashtable-links**

toggles reference counting via hashtables INTERNAL

(*:hashtable-links *hashtable flags* **)**

Sets (or gets if *flags* is **()**) the flags telling whether a hashtable increments the reference count of objects it holds. This is useful to maintain a database of objects to avoid circular references, but care must be taken to delete the entry of garbage-collected objects, otherwise access to the hashtable could crash the system. The flags value are:

flag	refcounted
1	only the keys are refcounted
2	only the values are refcounted
3	both keys and values are refcounted (default)

1.15 Strings

Note: all functions on sequences (see “Sequences” on page 36) apply to strings.

Predicate functions (see page 31) such as **`= /= < <= > >= compare max min`** apply to strings, too.

regular expressions

syntax of Klone regular expressions

Klone regular expressions use a slightly modified (for portability and with some enhancements, notablity the illimited number of sub-parentehses which was limited to 10 in the originla implementation) version of the freely distributable **REGEXP** package, Copyright (c) 1986 by University of Toronto, Written by Henry Spencer, Not derived from licensed software[1]. The rest of this text is copied from the REGEXP manual:

Regular expression syntax: A regular expression is zero or more branches, separated by '**|**'. It matches anything that matches one of the branches.

A branch is zero or more pieces, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an atom possibly followed by '*****', '**+**', or '**?**'. An atom followed by '*****' matches a sequence of 0 or more matches of the atom. An atom followed by '**+**' matches a sequence of 1 or more matches of the atom. An atom followed by '**?**' matches a match of the atom, or the null string.

An atom is a regular expression in parentheses (matching a match for the regular expression), a range (see below), '**.**' (matching any single character), '**^**' (matching the null string at the beginning of the input string), '**$**' (matching the null string at the end of the input string), a '****' followed by a single character (matching that character), or a single character with no other significance (matching that character).

A range is a sequence of characters enclosed in '**[]**'. It normally matches any single character from the sequence. If the sequence begins with '**^**', it matches any single character not from the rest of the sequence. If two characters in the sequence are separated by '**–**', this is shorthand for the full list of ASCII characters between them (e.g. '**[0–9]**' matches any decimal digit). To include a literal '**]**' in the sequence, make it the first character (following a possible '**^**'). To include a literal '**–**', make it the first or last character.

Ambiguity: If a regular expression could match two different parts of the input string, it will match the one which begins earliest. If both begin in the same place but match different lengths, or match the same length in different ways, life gets messier, as follows.

In general, the possibilities in a list of branches are considered in left-to-right order, the possibilities for '*****', '**+**', and '**?**' are considered longest-first, nested constructs are considered from the outermost in, and concatenated constructs are considered leftmost-first. The match that will be chosen is the one that uses the earliest possibility in the first choice that has to be made. If there is more than one choice, the next will be made in the same manner (earliest possibility) subject to the decision on the first choice. And so forth.

For example, '**(ab|a)b*c**' could match '**abc**' in one of two ways. The first choice is between '**ab**' and '**a**'; since '**ab**' is earlier, and does lead to a successful overall match, it is chosen. Since the '**b**' is already spoken for, the '**b***' must match its last possibility-the empty string-since it must respect the earlier choice.

In the particular case where no '**|**'s are present and there is only one '*****', '**+**', or '**?**', the net effect is that the longest possible match will be chosen. So '**ab***', presented with '**xabbbby**', will match '**abbbb**'. Note that if '**ab***' is tried against '**xabyabbbz**', it will match '**ab**' just after '**x**', due to the begins-earliest rule. (In effect, the decision on where to start the match is the first choice to be made, hence subsequent choices must respect it even if this leads them to less-preferred alternatives.)

1. This package is available from many sites, for instance by anonymous ftp koalai.inria.fr

Limitations: The restriction against applying '*' or '+' to a possibly- null operand is an artifact of the simplistic implementation.

Due to emphasis on compactness and simplicity, it's not strikingly fast. It does give special attention to handling simple cases quickly.

NULL BYTES: the regular expression system do not handle null bytes neither in the regular expression nor in the strings matched in the current implementation. This limitation will be removed in the future.

match

general regular expression matching package

(match *regular-expression string {number} ** **)**

This function matches and extracts substrings from a string. The *string* is matched against the pattern in *regular-expression* ; if the pattern is found, the function returns the appropriate substring, otherwise nil is returned. regular-expression can be a string or a regular expression compiled by **regcomp** (see page 44) for more efficiency.

If *number* is present, match returns the substring matching the part of the regular expression enclosed between the number-th open parenthesis and the matching closing parenthesis (the first parenthesis is numbered 1, and 0 stands for the part of *string* matching the whole regular-expression).

If number is absent, and the pattern is found, the function returns *string* .

If more than one number argument is given, the function returns a list composed of all the specified substrings in the same way as for a single number argument. The order of the list elements is determined by the number arguments (see example, below).

A maximum of 9 parts of regular-expression can be enclosed in parentheses.

```
(match ":(.*)" "foo:bar")              → "foo:bar"
(match ":(.*)" "foo:bar" 1)            → "bar"
(match "(.*):(.*)" "foo:bar" 2 1)      → ("bar" "foo")
```

The following is incorrect (too many parenthesized parts):

```
(match "(.*) (.*) (.*) (.*) (.*) (.*) (.*) (.*) (.*) (.*)"
   "string"
    1 2 3 4 5 6 7 8 9 10)
```

This example parses an X geometry string such as 80x24+100+150 into a list - x y width height.

```
(defun parse-x-geometry (string)
   (map List
      (lambda (x) (coerce x Int))
        (match "([0-9]*)x([0-9]*)[+-]([0-9]*)[+-]([0-9]*)"
                      string 3 4 1 2)))
(parse-x-geometry "80x24+100+150")           → (100 150 80 24)
```

NOTE: If there is a syntax error in the given regular expression, the function returns the error **Errors:BadRegexp**.

regcomp regexec regsub

regular expression package

(regcomp *pattern* **)**
(regexec *regexp source [source-offset]* **)**
(regsub *regexp source [source-offset]* **)**

These function provides a more complete control of regular expressions for more performance.

regcomp compiles a pattern (a string containing a regular expression) into a regular expression object, a **Regexp**, that can be stored for later use. If you are going to match the same pattern against many strings, this will allow you to save CPU time (match recompiles the regexp each time it is called).

regexec tries to match the *regexp* Regexp (returned by a previous call to **regcomp**) with the string *source* (eventually starting at *source-offset* character positions from the start of the string). regexec returns **t** in case of a match, or nil otherwise.

regsub uses *source* as a pattern, and returns a copy of sources where all occurrences of **\n**[1], where n is a digit, are expanded into the n-th matched string, \0 being the whole matched string in the last call to **regexec** done on the *regexp*. **&** can also be used instead of **\0**. See the **match** function for more details. This mecanism allows only to access the first ten matches, to access the other you must use other form where source is in fact a number, not a string.

```
(setq re (regcomp "A(.*)B")); matches chars between a A and a B
(regexec re "Foo And The Bar")        → t
(regsub re "&/\\1")                   → "And The B/nd The "
```

source can also be a number **n** to expand just the **n**-th matched string. In this case the argument *source-offset* is meaningless.

```
(regsub re 1)                         → "nd The "
```

---

get put delete insert

access characters in a string

(get *string position [default-value]* **)**
(put *string position character* **)**
(put *string position substring* **)**
(delete *string position* **)**
(insert *string position character* **)**
(insert *string position substring* **)**

get returns the character (8-bit integer for the ASCII code of the character) at *position* in the *string* (first character being at position 0), **put** sets the character at *position* to character, delete destroys the character, shortening the string length by one, and insert inserts the character at position.

A negative position means the last character of the string.

put and **insert** can overwrite a part of the string with a new string or insert a sub-string into the string if they are given a string to insert instead of a character.

```
(put "abc" -1 #\d)                    → "abcd"
(put "fooandgee" 3 "bar")             → "foobargee"
(insert "abc" 0 "-> ")                → "-> abc"
(get "abc" 1)                         → 98 ; ascii code of b
(delete "abc" 1)                      → "ac"
(put "abc" -1 #\X)                    → "abcX"
(put "abc" 5 #\X)                     → "abc  X"
```

---

get

give offsets of matched strings in a regexp

(get *regexp number* **)**

1. Do not forget to quote the backslash character in strings.

When a match on a regexp has been found by applying **regexec** (page 44) or **match** (page 44) on a regular expression, the offsets of the matched string corresponding to parenthesized sub-parts of the regular expression can be found (as a list of starting-offset (included) and ending-offset (excluded)) by getting the number of the parenthesized expression, as for **regsub** (page 44). these offsets can be directly used as arguments to **subseq** (page 38) or used for some elaborate processing.

```
;; parses a directory into path and name
(setq re (regcomp "^(.*)/([^/]*)$”))
*current-directory* Æ  → /u/avahi/0/koala/colas
(regexec re *current-directory*)                    → t
(get re 0)                                     → (0 27)
(get re 1)                                     → (0 21)
(get re 2)                                    → (22 27)
(get re 3)                           → Errors:NoElement
```

The -1 offset gives the whole string to which the regexp was last matched, including the non-matched parts

```
(setq re (regcomp "window"))
(if (regexec re (read-line))
   (print-format "%0\n" (get re -1))); prints whole line
```

(get *regexp* **'depth)**

This form returns the number of sub-parentheses contained in the regexp. There is no limit on this number.

```
(get (regcomp "foo") 'depth)         → 0
(get (regcomp "((f)oo) (bar)" 'depth)→ 3
```

---

toupper
tolower

converts case of strings

(toupper *string* **)**
(tolower *string* **)**

Return a copy of the *string* argument with all the alphabetic characters transposed to the correct case.

```
(toupper "89 aBcD")                → "89 ABCD"
(tolower "89 aBcD")                → "89 abcd"
```

---

substring-ptr

makes a virtual substring (pointer to another string) <u>INTERNAL</u>

(substring-ptr *string start-position [end-position]* **)**

returns a "virtual" string pointing to the string that would be returned by the call to:

```
(subseq string start-position [ end-position ] )
```

but only pointing to the relevant zone of string rather than copying the characters. Thus substring-ptr is much faster than subseq , especially for big substrings, but the returned substring should not be modified, otherwise it could truncate the original string to the end of the substring (altough no crash could happen this way).

---

make-string

creates a new string

(make-string *N [char]* **)**

Create and return a string composed of *N* (number) characters of ASCII code *char* (space, 32 by default). If N is negative or null, an empty string is returned.

```
(make-string 12 #\=)                    →  "============"
```

1.16 Streams and Input/Output

IO error handling

- All output functions trigger the error **Errors:StreamError** if an error happens while writing, most of the time a «device full» kind of error
- Input functions throw to the tag **EOF** when an end-of-file is read on the input stream.
- Output functions throw to the tag **EOP** when writing on a pipe to a process and this process dies. If **EOP** is uncaught (by a **catch**), the process exits silently.

Flushing the bufferisation
All Klone stream are buffered by default, so when printing on a stream, if you want to be sure that the output is actually sent (especially when writing non newline-terminated string to the tty), you need to flush explicitely the stream, as the Klone write primitives dont flush, except **?** and **print**, high level "user" primitives.

---

load parses and execute a Klone file CL p657

(load *filename [* **:if-does-not-exist** *action]* **)**

This function loads and executes a Klone file specified by *filename* , using the search path specified by the variable ***load-pathname*** (see page 48), and using the extensions provided in the variable ***load-extensions*** (see page 48) (both of them being a user-settable list of strings). The argument *filename* can be a string or a symbol, in which case the print name of the symbol is used.

If the filename begins with:

- /, it is considered as an absolute pathname and no path research and no attempt to add an extension to the file is done
- *~/name* , it is considered as an absolute pathname *name* in the current user home directory (no expansions are done)
- *~user/name* it is considered as an absolute pathname *name* in the user *user* 's home directory (no expansions are done)
- any other character, the filename is expanded.

The expansion of filenames is done in the following way: for each directory contained in the list ***load-pathname***, attempts to find an existing readable file consisting of the name *filename* , with all the extensions present in the list ***load-extensions*** being appended in turn to it (if filename doesn't end already with this expansion). This expansion mechanism can be used with the function **find-file** (see page 48) if needed.

The default value of the variable ***load-pathname*** is specified by the shell variable **KLONEPATH** by default and the default value of ***load-extensions*** is **(".kl" "")**, but this can be redefined by the application.

If the file is found, the function returns the actual full pathname of the loaded file as a string. If the file is not found:

- If the keyword **:if-does-not-exist** is not specified, or is specified to **:error**, an error is signalled,
- otherwise *action* is evaluated and the result returned with no error message**nil**.

__NOTE__: As many packages redefine load, the name ***:load** always points to the original load function to be safely called.

continue-reading-on-error what to do if an error occurs during load

boolean

Normally, if an error occurs during a **load**, the rest of the file is not processed. Setting this flag to true will make Klone try to load the rest of the file anyway.

Warning: be warned that this may induce quite a lot of error messages.

load-pathname
load-extensions control loading of files CL p657

lists of strings

These lists (initially set by the application, the sample Klone top-level setting them to **("." "/usr/local/lib/klone")** and **(".kl")** respectively) controls the search for files which is done by the functions **load** and **find-file**.

For ***load-pathname***, nil strings **""** are taken as the current directory. nil is valid for ***load-extensions*** (only the raw file name is tested), but ***load-pathname*** must have at least one element, otherwise no directory will be tried.

find-file search for file in path

(find-file *filename* **)**

This function applies the same algorithm as **load**, but only returns the complete pathname of the file if found or nil, and does not perform any other processing.

expand-filename expands special characters in filename

(expand-filename *pathname* **)**

Expands the special characters in filename and returns the expanded pathname, or nil if expansion was unsuccessful. Currently only the ~ character is expanded at the start of the path, **~foo** meaning the homedir of user foo, and **~/bar** meaning the file in the home directory of the user running the process.

```
(expand-filename "~foo/bar") → () ; user foo do not exist
(expand-filename "~/.profile") → "/u/avahi/0/koala/colas"
```

file-position get or set current point in an open file CL p655

(file-position *stream [offset [from_where]]* **)**

Without *offset* the current file pointer position (as an offset in bytes from the start) in the open file stream *stream* .

If the number *offset* is specified, the file pointer is set to offset bytes, from the start (absolute position) if *from_where* is **0** (default), relatively to the current position for **1** and from the end of the file if **2**. The true value **t** is returned if the file position could be set (i.e., if the offset was positive and the file was a true file, not a pipe or an interactive tty), or **()** if not.

If the number *offset* is past the end of file, the file or string is extended past the end of file. Strings are filled with null bytes, and file contents are not specified.

As an example, the **is-seekable** function to know if a stream can be seeked (if its current position can be changed, i.e. it is not a pipe):

```
(defun is-seekable (filedesc)
   (file-position filedesc (file-position filedesc)))
```

open

open (create) a stream CL p646

(open *filename* **[:direction** *direction* **]**
[:if-exists *ifexists* **]**
[:type *type* **]**
[:buffered *buffered* **]**
[:writer *file-for-write* **]**
[:blocking *blocking* **]**
[:error *return-value* **]**
)

This function creates a stream by opening the file *filename* and returns an object of type **stream**. The keywords and their corresponding values determine the type of stream to be created:

:direction specifies if the stream is readable, writable or both, if the value of *direction* is **:input**, **:output**, or **:io** respectively. (defaults to :input)

:if-exists specifies the action to be taken if the file already exists and is opened in :output or :io mode (if it does not exist, it is created). If value of *ifexists* is

- **:append** the contents of the file is preserved and the current position is at the end of the file (default value)
- **:overwrite** the starting position is the beginning of the file. The file is not truncated to zero length with this operation; thus, file contents at the beginning are overwritten, while subsequent file contents may remain
- **:supersede** the starting position is the beginning of the file. The file is truncated to zero length with this operation; thus all previous file contents are overwritten
- **:error** an open error is triggered, or the value of the **:error** keyword is evaluated and returned if it was specified. This macnism can implement thus a simple file-locking mecanism

:type specifies the type of the file. The value of *type* is one of:

- **:file** the stream is created by opening the UNIX file of name *filename* (default value)
- **:string** the stream will be created on the provided string *filename* . Reading this stream will return the characters in the string, while writing on it will physically add characters to this string, and all I/O functions such as read or print can be applied. as well as string-manipulation commands on *filename* itself.
- **:tcp :pipe :udp** for opening inter-process channels, sovckets or pipes, see below: **Network interface: Sockets**, and **Unix Pipes**.

:buffered specifies whether the buffer should be flushed or not after each (output) operation. The value of buffered can be nil (flushing after each character) or :flushing (default value, use the underlying unix bufferization)

:writer in the case of streams of type **:file** opened in **:io** mode, allows to specify an alternate UNIX file to write to, *filename* being then accessed only for reads. This is to use UNIX devices implementing reading and writing to a same entity, but via two different file descriptors, such as pipes or sockets. Otherwise, the same file descriptor is used both for read and write operations.

```
(setq string "(+ 1 2)")(setq string "(+ 1 2)")
(setq stream-string (open string :type :string))
(print (eval (read stream-string)))                    ⇒ 3
```

:blocking If *blocking* is true (default), IO operations on the stream will block, waiting for completion. If false, reading or writing the file may return a premature EOF (for instance due to network delays or if the steam is a pipe and the process at the other end is not communicating). For best results, such stream should only be handled via the **read-chars** and **write-chars** functions (see page 55). The blocking state can be toggled on the fly at any time via the **stream-mode** (see page 51) function.

:error If the file cannot be opened, an error **Errors:ErrorOpeningFile** is triggered, unless a *return-value* has been specified, in which case this return value is silently evaluated and returned. Note that this behavior is slightly different from the Common Lips keyword **:if-does-not-exists**, which makes open fail even if :output mode.

```
(setq s (open "/fghrjgh/dffg"))
Error: Cannot open file /fghrjgh/dffg with mode read/write
setq s (open "/fghrjgh/dffg" :error ())→ ()
```

Network interface: Sockets

open can also create sockets of connected (stream) type with the **:type :tcp** declaration, and of unconnected (datagram) type with the **:type :udp** keywords. In this case the filename must be of the form:

- **hostname:port-number** for a client socket, that will connect to a remote server. open will perform the connect automatically. Non blocking connect can be done by specifying the keyword **:buffered ()** to open, a later inspection by **select**ing the socket for writing readyness can determine when the connect suceeded.
- **:port-number** for a listening socket, the klone process being a server on current host at port number. open will perform the bind, automatically incrementing the port number till a free one is found. The actual port number bound to can be found in the global variable **socket:last-bind**. adding a trailing bang sign to port number, such as in **":5423!"** will prevent this automatic incrementation.
  When a connection arrives on such a listening socket, you must re-open it to get a newly created socket to read actual data on.

```
(setq listening-sock (open ":5432" :direction :io
```

Keywords :buffered and :blocking apply also to socket streams.

WARNING: default modes are the same as for other streams, but normally you want rather use the following modes: **:buffered () :blocking () :direction :io**, do not forget to set them on each open, especially when re-opening a listening socket

Unix Pipes
open can create a bidirectional unix pipe with **:type :pipe**. The filename is then just ignored (can be used as a comment). Useful with the ***: fork** function.

---

close

close a stream object CL p505

(close *stream* [**:writer** *flag*] **)**

This function is automatically applied when a stream object is freed, so it is not needed to explicitly close the stream if it is held in a temporary variable, such as in the following example:

```
(with (*standard-input* (open "my-file")) (read-line))
```

Reading a closed file will trigger the error Errors:StreamError:

```
(setq stream (open "khoros-FAQ"))
(read-line stream)                        → "..."
(close stream)
```

```
(read-line stream)
⇒ Error: cannot read on stream {^ Stream khoros-FAQ closed}
```

:writer keyword: When the stream is opened in **:io** mode (bidirectional) , close only closes the input direction if *flag* is nil, putting the stream in input-only mode, or if *flag* is not nil, closes the the output direction, putting the stream in output-only mode. If **:writer** is not specified, both directions are closed at once.

Closing a stream can cause write errors to happen, since close forces flushing the buffers to disk, at a time when disk space might not be available anymore.

---

stream-mode

control the mode of open streams

(stream-mode *stream* *[***:blocking** *blocking]*
*[***:writer** *writer]*
*[***:buffered** *flag]* **)**

Sets the mode of the stream (or get if no mode keyword is specified). currently only the mode **:blocking** and **:buffered** are implemented. If *writer* is specified to **t**, the mode of the writing side of the stream in case of io (bidirectional) streams is set, otherwise only the reading side of the stream is set.. Without mode keywords:

(stream-mode *stream* *[***:writer** *writer]* **)**

returns a p-list of the mode of the file. currently only returns **(:blocking t)** or **(blocking ())**

:buffered must be set after opening the

?
print

print the external representation of an object

(print *{form} ** **)**
(? *{form} ** **)**

Print outputs on the current output stream the print representation of the forms passed as arguments. **?** is just a synonym for print, since print has another semantic in Common Lisp, some Klone implementations may want to redefine print as the Common Lisp function, so **?** will guarantee you to have the Klone semantics. Print does not add any space between representation of objects and do not append a newline at the end.

print will try to present objects in a human-readable form, unless the value of the flag ***print-readably*** is not nil, in which case it will print the object in a machine-readable form (which can be re-parsed by klone and yield the same object)

```
(setq *print-readably* ())          ; default
(print "a\tb")                      ⇒ a   b
(print 'a)                          ⇒ a
(print (type-of 1))                 ⇒ Int
(setq *print-readably* t)
(print "a\tb")                      ⇒ "a\tb"
(print 'a)                          ⇒ a
(print (type-of 1))                 ⇒ {TYPE Int}
```

print prints on the default output stream specified by ***standard-output***.

This function is the only one to automatically perform a flush on the stream after each invocation.

<u>**Cl-Diff**</u>: print in Common Lisp first output a newline, then print the object.

print-format

print formatted output

(print-format *[output-stream* | **String** *]*
{output-format | *convert-function}*
*{arguments}** **)**

This powerful and very useful function prints a formatted string to the stream *output-stream* . The printed string is built by integrating the specified *arguments* to which a conversion specification is applied into the string *output-format* . The function **print-format** always returns nil.

If *output-format* is specified as a string, it includes conversion specifications as follows:

- occurrences of **%N** are replaced by the Nth argument, where **N** is a positive or null integer (no limits on size).
- occurrences of **%tN** are replaced by the type of the Nth argument
- occurrences of **%rN** are replaced by the "readable" representation of the Nth argument, as if ***print-readably*** was set.
- occurrences of **%lN** are replaced by the "readable" representation of the Nth argu ment on only one single line, by quoting also the newlines in strings, as if ***print-readably*** and ***quote-newlines*** were set.
- occurrences of **%nN** are replaced by the "non-readable" representation of the Nth argument, as if ***print-readably*** was unset.
- occurences of **%sN** means printing the argument **N**, which must be a String, (if not, a coercion to String is attempted) as a **raw string** (see page 11) form, i.e. the raw characters of the string framed by a **^^** (control-^, ascii code 30 / 0x19 / 036) before and after, that can be re-read by the Klone parser.
- occurences of **%f***function-name* **N** are replaced by the printing of the result of the evaluation of the call **(***function-name argument* **N).** function-name can only be composed of letters, colons, dash and underscores, numbers are not allowed
- occurrences of **%%** are replaced by the character '**%**'

Option letters can be combined. **%rffoo3** means "print readably the result of applying foo to the 3rd argument". The r and n are the only combinable modifiers in practice.

If *convert-function* is specified as a function, this a function is applied to the rest of arguments, and which must return a string that will be used as a format that is applied to arguments by print-format.

If *output-stream* is nil or omitted, output is directed to the standard output.

Range errors are silently ignored, you can supply extraneous parameters that will be ignored (but nonetheless evaluated), or can refer to non existent parameters with %N orders with N greater than the number of parameters with no problems, they will expand into the null string.

If *output-stream* is the type **String** (or the atom **'String**) then print-formats prints on a string and returns it (otherwise nil is returned in the other forms).

```
(setq var t)
(print-format ()
   "The first argument is %r0, the second is "
   "a string"
   var)   → ()
          ⇒ The first argument is a string, the second is t

(defun converter (&rest parameters)
   "This function returns the following:\
   A list of <N> elements: <element1> ..."
   (let ((list-length (length parameters))
```

```
            (convert-returns (append "A list of "
            (coerce list-length String)
            " elements: ")))
            (dotimes (pos list-length)
            (nconc convert-returns " r%" (coerce pos String)))
    convert-returns)
)
(print-format () converter 1 2 3)
                      ⟹ A list of 3 elements: 1 2 3
(print-format () converter "a" 2 "b" 4)
                      ⟹ A list of 4 elements: "a" 2 "b" 4
(print-format String "%0:1" 'foo 'bar)⟶ "foo:bar"
```

print-readably

control printing detail

boolean

print-readably is a variable. If its value is **nil** (default), all output is «user-readable», i.e. objects are printed in a terse form to give the most pleasant results. Strings are printed without the surrounding quotes, and with non-printing characters non escaped.

If its value is **t**, all output is printed in a «machine-readable» form allowing them to be re-read and give the same object by Klone. Strings are surrounded by double quotes, and special characters such as quotes and backslashes are escaped to be able to be re-read in Klone afterwards, Reals are printed to the maximum precision, types are printed by a coerce inline call, and functions are printed by their entire definition intead of just their header.

```
(setq *print-readably* ())
(print "This i\s \x61 string\n")      ⟹ This is a string
(print "This is a \t string")         ⟹ This is a      string
(setq *print-readably* t)

(print "This i\s \x61 string\n")      ⟹ "This is a string\n"
(print "This is a \t string")         ⟹ "This is a \t string"
```

print-readably-as-raw-strings

control printing form of strings

boolean

When printing readably via ***print-readably***, prints strings in the "Raw Strings" format (literally, surrounded by Control-Caret characters), see page 11

quote-newlines

prints newlines in strings as \n

boolean

if set to **t** will expand newlines in strings into **"\n"** instead of printing them as true newlines when printing with ***print-readably*** set.

print-level

control printing depth of lists CL p564

number

print-level is a variable that controls the maximum depth at which the lists will be printed. Lists that cannot be printed are holophrasted into 3 dots "...". Default is 9. Setting it to a negative

```
(setq *print-level* 2)
'(1 (2 (3 (4 1))))                    → (1 (2 (...)))
```

`*print-binary*`

control printing form of strings — TEMPORARY

boolean

If true, strings are printed “as-is” in a raw form intended for fast parsing and printing.

Note: This feature is an experimental one. Its presence in future Klone versions is not guaranteed.

`read`

read and interpret a string — CL p569

(read *[input-stream [error]]* **)**

This function reads the next expression on the stream *input-stream*, if specified, or **`*standard-input*`** otherwise, parses it and returns it. When the end of the file is reached, the function executes a throw to the tag **`EOF`** or returns the evaluation of the optional *error* form if specified.

```
(defun my-read (stream)
;; next line is documentation for this function
 "Reads a stream up to end and prints each object"
 (catch 'EOF
   (while t
          (print "Read: " (read stream) "\n")
))))
```

Warning: In the current function the parser is not reentrant, so that you cannot call read while reading a file. The only possibility to call the parser from within the parser is via the in-line function facility (braces, see page 13), where no read operation should be attempted, otherwise an error could occur.

`read-char`

read a character — CL p573

(read-char *[input-stream [error]]* **)**

This function reads one character on the stream *input-stream*, if specified, or **`*standard-input*`** otherwise, and returns it as a 8-bit number. **`read-char`** can be useful with binary files, where bytes can be read in this way. When the end of the file is reached, the function executes a throw to the tag **`EOF`** or returns the evaluation of the optional *error* form if specified.

`read-line`

read a a string — CL p572

(read-line *[input-stream [error]]* **)**

This function reads a string of characters on the stream *input-stream*, if specified, or **`*standard-input*`** otherwise, and returns it, discarding the newline at the end. When the end of the file is reached, the function executes a throw to the tag **`EOF`** or returns the evaluation of the optional *error* form if specified.

```
;; very common code:
(while (setq line (read-line fd ()))
    (process-line line))
```

This example defines a function **`read-file`** which reads to the end of a stream and returns it as a string, using a catch on the tag **`EOF`**. This function is then applied to a file named **`/tmp/data`**.

```
;; The file /tmp/data contains:
"This is a string"
(setq a 1) (setq b 2)

(defun read-file (stream)
;; next line is function documentation:
 "Reads a stream to the end and returns it as a string \
 discarding newlines"
 (let ((res (copy "")))
   (catch 'EOF
          (while t (nconc res (read-line stream))))
   res))
(read-file (open "/tmp/data"))
          ⇒ "This is a string" (setq a 1) (setq b 2)
```

WARNING: do not use read-line to read binary data as read-line will take any null character in the input as meaning EOF. Use **read-chars** or **read-char** instead. read-line can read lines of unlimited length, however.

read-lines-pooled

read lines into a pool until a complete expression is found

(read-lines-pooled *[input-stream [error]]* **)**

Read the input, line by line, until (at least) a complete expression is found, and returns the pool as a string with embedded newlines. The pool may consist of more than one expression, and which may trigger syntax errors or evaluation error if parsed and evaluated later. The expression can even be non-closed if and only if the end of file was reached in the middle of an expression. The system will issue the normal prompts determined by ***:prompt** (see page 69).

Throws to tag EOF on end of file, or returns the evaluation of *error* if given

```
(with (*:prompt '(">" ">>" " " " = " " "))
   (read-lines-pooled))
> (+
>>    1 2
>>    ) (and 3 4)
                          → "(+\n1 2\n) (and 3 4)"
```

read-chars

reads a number of characters

(read-chars *[N [stream]]* **)**

returns a string made by reading at most *N* chars on the *stream* . If *N* is not specified or set to (), or negative, returns as much as it can read. *stream* defaults to ***standard-input***. This function is useful to read characters from streams in non-blocking mode.

write-chars

writes a number of characters

(write-chars *string [N [stream [offset]]]* **)**

writes at most *N* first chars (from start or from *offset*) of *string* to *stream* (defaults to ***standard-output*** Returns the number of bytes actually written. if *N* is not given or is (), or negative, writes up to the end of the string. This function is useful to rewrite characters to streams in non-blocking mode.

standard-error
standard-input
standard-output

standard streams CL p497

variables

These are global variables that are, by default, respectively set to the stdin, stdout and stderr streams of the UNIX process. If any variable is set to nil, it is reset to its default value, values which are saved in ***standard-error-orig***, ***standard-input-orig***, and ***standard-output-orig*** respectively.

```
(with (*standard-output* *standard-error-orig*)
   (print "I cannot do THAT!\n"))    ;; send text to stderr
```

write

write a Klone object on a stream CL p577

(write *object* *[output-stream]* **)**

This function writes (prints) a klone expression *object* on the stream *output-stream* , if specified, or ***standard-output*** otherwise, and returns it.

write-char

write a character on a stream CL p572

(write-char *number* *[output-stream]* **)**

This function writes one character (specified by ASCII code *number*) on the stream *output-stream* , if specified, or ***standard-output*** otherwise, and returns it. The function can be useful for creating binary files, where bytes can be written in this way, even the NULL byte that cannot be handled by string-oriented functions.

write-line

write a string on a stream CL p579

(write-line *string* *[output-stream]* **)**

This function writes a string of characters *string* on the stream *output-stream* , if specified, or ***standard-output*** otherwise, and returns it. It appends a newline at the end of the line, just like read-line strips it.

flush

flushes a stream

(flush *stream* **)**

Flushes all the output pending in the file buffers on the given stream, or ***standard-output*** and ***standard-error*** if the *stream* is **()**. Needed after all writes, except via the **?** or **print** primitives.

execute-string

evaluate an expression given as a string

(execute-string *string* **)**

This is a handy function to **read** all klone expressions in *string* , and **eval** them. The result of the last evaluation is returned.

```
(execute-string "(+ 2 2)")            → 4
```

select

wait for input (or readiness) on multiple streams

(select *[***:input***] streams...*
*[***:output** *streams...]*
*[***:error** *streams]*
*[***:timeout** *milliseconds-or-nil]* **)**

select is an interface to the Unix select system call, allowing to do passive waits on multiple file descriptors. The Klone version has the unusual advantage however of working with Klone streams, which are buffered (Unix select works only on unbuffered file desciptors, not buffered file pointers), and may be strings. Select returns the lists of streams that are ready for input (reading), output (writing), and on which an exception is pending (error). This latter case is best avoided in practice unless you know exactly what you are doing. Beware also when testing writing availability as it only tells you that you can write one byte, but may block on the following one. In actual use, select must be used in combination with non-blocking I/O (see **read-chars** and **write-chars**) for best performance and safety.

Select returns immediately if some streams are string streams, or if some characters are present in the buffer of input streams. In these cases, the unix select call is not triggered, so that string streams have precedence over file streams.

Select can return:

- () no stream were ready, and timeout expired or signal received)
- a list of 3 lists being the lists of streams ready for input, output, and having an exceptional condition ("error") pending.

Select is very flexible (more than normal keyword matching rules) n the way its parameters can be specified:

- the **:input** keyword is optional, as it is the only one nearly always used
- streams can be either streams or lists of streams which are expanded. you can write to monitor the 3 streams **a**, **b**, and **c** one of the forms:

```
(select a b c)
(select :input (list a b) c)
(select [a b c])
```

- any stream can be () and is then silently ignored
- keywords can appear in any order

This flexibility is aimed easying the task of maintening lists of streams to be watched by select.

The **:timeout** keyword can be given as argument either:

- **()** (default), meaning that select will block indefinitely until a stream becomes ready
- **0**, meaning just poll the state of descriptors and immediately return the list of the ones that were ready or () if none were.
- a number of milliseconds, meaning if no descriptor were ready, wait at most this time before returning nil. of course, as soon as a descriptor is ready, select returns. Note that **(select :timeout N)** is a handy way of implementing delays with a good precision.

Warning: (select) blocks indefinitely!

Warning: select on a file already at EOF returns true! (unix behavior), since a read on this stream would not block, so plan for this case in your code reading this ready stream after the select call.

1.17 Links

Links are indirections that do not increment the reference count of the object pointed to. This makes it possible to design circular lists or data whose dependence graphs contains cycles in Klone without making these structures ungarbageable. As a general rule, you should design your structures with all the “forward” references (son-of, next-element) implemented as normal fields, and all “backward” references (father-of, previous-element) implemented via links.

link-set

create a link to an object

(link-set *object* **)**

Creates and returns a link pointing to object

link-get

de-reference a link

(link-get *link* **)**

Returns the object pointed to by the link. If the object have been destroyed, **link-get** returns nil. If link is not a link, it is returned as-is, to be able to implements fields as links or plain references transparently.

Note: link-get tries to recognize when the object do not exist anymore (the famous case of “dangling pointers” as in C), but can be fooled if a new object has taken the place of the old one with the same type. But it will never crash the system (bus error or memory fault), if fooled, it will return a valid Klone object of the same type as the one that was pointed to.

```
(setq message "Hello, world!")
(setq ptr (link-set message))        → {^ Link 0x78e80}
(link-get ptr)                       → "Hello, world!"
(setq message "foobar")              ; previous text garbaged
(link-get ptr)                       → ()
```

1.18 Operating System interface

Klone supports a minimal but complete interface to the underlying Operating System (Unix). With this primitives, Klone could be used as an Unix shell. Note that Klone uses **vfork** if available, to allow efficient use of system even if the application embedding Klone is large.

system

run an external sub-process

(system *command* [**:input** *in*] [**:output** *out*] [**:io** *io*]
[**:error** *err*] [**:nohup** *flag*] **)**

Forks a shell (**/bin/sh**) to execute the string *command* , and optionally creates Unix pipes connected to this sub-process. *in* , *out* and *err* are symbols (must be quoted) to which will be set the newly created streams. an *io* mode is a bidirectional stream connected to both the input and output stream of the forked process. Giving the same atom for the keywords :output and :input has the same effect as declaring it via io. Writing on the :input pipe to the son will trigger a throw to 'EOP when the son dies and reads on :output and :error will trigger a throw to 'EOF

command can also be a list of strings, in which case the first string is taken as the name of an executable file which is then executed by Klone with the rest of the list as parameters (via the Unix primitive **execvp**). This forms induces less overhead and allows easier inclusion of parameters with special characters, since they must be quoted in the first form to prevent them from being evaluated by the shell.

This function returns the process ID of the forked process, which can then be used afterwards with the **wait** function (see page 63) to wait for termination of the child.

If the command was not found, the exit return value that will be returned by wait will be 255, but only if the call did not fork a shell (argument as a list, not a string). If /bin/sh is forked, it will be the return status of the shell when the command specified by -c cannot be found, 127 on most machines.

in , *out* and *err* can also be **strings**, in which case they are used a the names of files to open to connect to the forked process standard input, output and error file descriptors. Of course these can be any legal UNIX name, such as **"/dev/null"** if you do not want to see the program output.

If *out* is equal to *in* , then only one bi-directional stream will be created for input and output. Similarily, if *out* and *err* are the same, the forked process input and output streams will be merged (via the unix **dup2** function) into the same stream.

If **:nohup** sets *flag* to **t**, the forked son will ignore the signal SIGHUP that will be sent to him when the current Klone process terminates.

Warning: There is of course a potential deadlock if you open streams both on the input and the output of the command, and write too big chunks of data on the input pipe.

```
;; getting the date of the day as a string
(with (stream ())
   (system '("date") :output 'stream); fork the unix date
   (coerce stream String))         ; read all stream

;; forking the desk calculator to do "2 + 2"
(system '("dc") :input 'i :output 'o)          → 12118
(write "2 2 +p\n" i)
(read-line o)                                     → 4
(close i)                  ⇒ kills the dc process No 12118

;; writing the date into file /etc/today
(system (list "date" :output "/etc/today")
```

Note: a simplified version is defined in Klone. See **sh** and **sh:open** page 80

Note: When the Klone process terminates, it will send the signal 1 (SIGHUP) to all its forked sons, unless the global varianle ***SIGHUP-on-exit*** is set to () (defaults to t). Setting this variable only affects the sons forked after its setting, so it is recommended to set it at the beginning of the Klone session.

!
!!

execute an external command OBSOLETE

(! *command arguments ** **)**
(!! *command arguments** **)**

Warning: These commands are made obsolete by the new **system** function above (page 58). They are still defined in Klone for backwards compatibility, but please use **system** instead.

! executes (forks) the command given as a string with its given arguments, and does not wait for termination (so you don't need to add a final "&" to the shell command).

!!* is like **!**, but blocks waiting for command completion.

```
(! "date")
(!! "cat" "/etc/passwd");; you must separate the arguments
```

NOTE: These functions provide a direct interface to the UNIX execvp call: the command is searched for in the PATH variable, and is either executed via "/bin/sh" if it is a command file, or directly executed otherwise. If you want to pass a shell command (with command redirection for instance), you must use something like:

```
(! "/bin/sh" "-c" "cat /etc/termcap | grep BULL > /tmp/foo")
```

***:exec** replaces the current process with another executable

(*:exec *arguments...* **)**

terminates the running process and exec (in the unix sense) another in replacement, keeping the same open standard file descriptors.

***:fork** forks a clone process

(*:fork)

Direct interface to the unix call FORK, will create an identical process. The only way to distinguish between the son and the father is that ***:fork** in the father will return a process id (like **system**), the process id of the son on which **wait** calls can be performed, and **()** in the son.

---

keep-streams-exec keeps streams open on fork or execs

List of Streams

If set to a list of streams, these streams will not be closed when executing a subprocess in the subprocess itself. Normally, **system** and ***:exec** closes all files descriptors excepot the standard ones (stdin, stdout, stderr).

---

get-internal-run-time get the current process time in milliseconds CL p705

(get-internal-run-time
[**:type** *{* **:real** | **:sys:** | **:user** | **:cpu** | **:all** *}]* **)**

This function returns a time value in milliseconds. If no argument is provided, the returned value is the elapsed time, otherwise the value depends on the keyword:

:real The actual elapsed time for which Klone has been started (default value).

:user The CPU time used while executing Klone instructions.

:sys The CPU time used by the system on behalf of Klone.

:cpu The total CPU time used. useful for benchmarking, is the sum of user and system times.

:all The list of the 3 values (real-time user-time sys-time)

---

get-current-time get the absolute time in seconds since Jan 1 1970

(get-current-time)

Returns the absolute time in seconds since Jan 1 1970 (greenwhich meridian). Use the loadable date functions time-to-date, etc... to convert to more usable formats.

getenv
putenv

handle the Unix environment variables

(getenv *variable-name* **)**
(putenv *variable-name value* **)**

If the string *variable-name* is the name of an exported shell variable, **getenv** returns its value as a string, otherwise the function returns nil.

putenv can modify the value of shell variables, either for successive **getenv**s in the same Klone session, or for spawned processes. If value is (), then the variable is unset (deleted).

```
(getenv "Foo")                        → ()
(putenv "Foo" "Bar")
(!! "sh" "-c" "echo $Foo")            ⇒ Bar
```

listenv

lists the Unix environment variables

(listenv)

returns a p-list of all the defined environment variables with the keys being the variable names and the values being the value of these variables as strings

hostname

name of the machine running the process

string

Has for value the host name on which is running klone.

arguments

command line arguments of the current process

list of strings

This is the list of all arguments of the process. First one in the list is the name of the process, and the rest are the actual command line arguments.

trap-signal

catch an Unix signal

(trap-signal *signal handler* ***)**

This function make the Klone process execute sequentially the actions specified by the *handler* s when receiving the Unix signal *signal* (specified by its number[1]). If handler is:

- **()** nothing is done, and the signal performs its default action which depends on the signal (nothing, terminating the process, and maybe dumping a core file).
- **t** the signal is ignored, nothing is done.
- **a string** a process is asynchronously forked (via **/bin/sh**) to execute this shell command. The command is first protected from sinal 1 (death of the klone process) to allow the parent process to exit immediately, e.g:

```
(trap-signal 2 "clean-temporary-files" 1) ; on ^C, clean and
exit with code 1
```

- **a symbol** sets its value to **()**, and will set it to **t** as soon as a signal is received.

1. names of all the unix signals are provided in the autoloadable file signalnames.kl

- **a number** makes the Klone process terminate with this return code.
- a **subr** a specific application-provided handler is executed. (called without arguments). Refer to each Klone-embedding application manual for the available handlers. No such handler is provided as part of the klone code, however.
- a vector of the form **[()** *lambda-expresssion* **]**, in which case the Klone lambda-expresssion will be immediately executed when the signal arrives and the Klone process will forcibly exits immediately afterwards without returning, as its internals may have been garbled
- a vector of the form **[t** *lambda-expresssion* **]**, in which case the process will fork a sub-process and will immediately resume its operation. The forked Klone process will execute the lambda-expresssion and exit immediately afterwards.

Note: It is not possible in this version of Klone to specify an arbitrary Klone code as a handler.

```
(trap 1 t)              ;; ignore the termination of the parent
(trap 2 "rm /tmp/lock") ;; do not forget to remove lock file if interrupted
```

exithook

code executed on exit

Function

If set to a function, this function will be called before exiting the process. This can be a function or a vector of functions that are called sequentially.

errno

text of the last unix error message

String

holds the last unix error message in its textual form.

directory

list the contents of a directory

(directory *[pathname]* **)**

Returns a list of all filenames (as strings) of the directory *pathname* , or the current directory if none was specified. "." and "..", the current and parent directories are omitted from the list but all the entries of the directory are listed, even those beginning by a dot, but in no significant order. If the directory does not exists or is protected, the error **Errors:BadDirectory** is returned.

```
;; ls: list the files in current dir alphabetically sorted:
(map () (lambda (filename) (? filename "\n"))
   (sort (directory) compare))
```

file-stats

attributes of an Unix file

(file-stats *filename* *[do-not-follow-symbolic-links]* **)**

This function return a structure describing the filename of file *filename* , or () if the file could not be found. The Unix fields of the STAT(2) function are accessible by **get** on this structure, with the following keys:

name	meaning	type
dev	device number	Int
ino	inode number	Int
mode	modes of the file (see below)	flags
nlink	number of hard links on this inode	Int
uid	user-id of the owner	Int

name	meaning	type
gid	group-id of the owner	Int
rdev	raw device number	Int
size	size of the file in bytes	Int
atime	date (in seconds since 1-1-1970) of last access (read)	Int
mtime	date of last modification (write)	Int
ctime	date of last mode change or creation (open)	Int
blksize	size of the file system blocks in bytes (BSD only)	Int
blocks	number of disk blocks used by the file (BSD only)	Int

mode is an integer describing the mode of the file. See the unix man page for STAT(2), or the files **file-info.kl** or **file-type.kl** included in the Klone distribution. **stat-params.kl** contains the definition of useful constants for decoding the mode field.

If *do-not-follow-symbolic-links* is present and non nil, information is returned on the file itself, even if it is a symbolic link, instead of following the link and returning stat on the file pointed to which is the default option.

```
;; file-type returns the type of the file as an atom.
(defun file-type (name &aux
        (stat (file-stats name))
         pos)
  (if stat (progn
      (if (setq pos (position
              (logand S_IFMT (getn stat 'mode))
              file-type:plist))
        (getn file-type:plist (+ pos 1))
))))

(setq file-type:plist
  `(
  ,S_IFREG file
  ,S_IFDIR directory
  ,S_IFLNK symbolic-link
  ,S_IFCHR character-special
  ,S_IFBLK block-special
  ,S_IFSOCK socket
  ,S_IFIFO fifo
))
```

file-truncate

trucate or set file length

(file-truncate *path length* **)**

Sets the length of the file named *path* to the size *length* . truncates or expand the file if needed, filling it with undetermined values. The function returns **()** on success or raises the error **Errors:ErrorOpeningFile** if it doesnt exists or if the access was denied.

wait

wait termination of a process

(wait *[process-ID* | **(** *process-IDs** **)** | **()** *] [***:blocking** *flag]***)**

Wait is a very powerful function and allows quite a bit of smart external programming by forking external commands via system.

This function operates on sub-process created by the **system** (see page 58) command, and whose processes IDs (values returned by system) have been stored in a variable. If the flag for the keyword **:blocking** is provided with value **()** (defaults to **t**), then wait never blocks, and **wait** returns () if it may have blocked. Processes whose process-ID has not been stored at the time of the **system** command are totally ignored by the **wait** command. If the argument to wait is:

- **A process ID**: wait blocks till the process is terminated (or returns immediately if the process has already terminated). The return value is **0** if the process ended normally, or a value comprised between 1 and 127 (Unix return code, see **exit** page 25) if the process returned an error code, or a negative number if the process was killed by an Unix signal (the number is then the opposite of the number of the signal)

`()` is returned immediately if the process has already ended and its return value has already been returned by a previous call to wait, or if `:blocking ()` has been specified.

- **A list of process IDs**: waits for **All** the processes of the list to terminate, then returns a list of all the result values of the processes, which are coded as above.

  If `:blocking ()` has been specified, returns the list immediately, with elements of the list being the return code of the command or () if it is still alive, or nil if all processes have already been reported..

  If `:blocking 1` has been specified, waits and returns as soon as a process of the list finished (or returns immediately if one has already returned) a list of two elements, the first one being the process ID which has finished, and the second the return code of the command. Returns nil if no process remains to be reported.

- `()`: waits for any child to terminate, and returns a list of two elements, the process ID and its return code, of any child who died since the last called to wait, or () if all children are either already reported, or still alive. (wait () :blocking ()) returns immediately () if no child has already died and has not been already reported by a wait command.

```
;; forks two compilations and do the link if no errors happened
(setq compiles (list
 (system '("cc" "-c" "foo.c")) (system '("cc" "-c" "bar.c"))))
(if (= '(0 0)(wait compiles))
    (system "cc -o foobar foo.o bar.o")              ; success
    (? "Compilation error, no link done\n"))         ; failure
;; timeout on a command implemented by forking a sleep command:
(setq ftp (system "getit avahi.inria.fr:README")) ; get a file by ftp
(setq timeout (system "sleep 100"))     ; start a timeout at 100 seconds
(setq first-to-complete (wait (list ftp timeout) :blocking 1))
(if (= timeout (get first-to-complete 0))
    (print "ftp timed out")                         ; timed out a 100s
    (if (= 0 (get first-to-complete 1))
          (print "ftp succeeded")                 ; Ok, return code = 0
          (print "ftp error")))             ; Error (other than timeout)
```

`*current-directory*` get and set current directory

String

Returns or changes the current directory. (setq *current-directory* foo) returns foo. If it was not possible to go into this directory, triggers the error **`Errors:BadDirectory`**.

`*current-process-id*` the process ID of the current process

Int

Returns the Unix process ID of the application running the Klone interpreter.

```
;; create a unique temporary file
(open (+ "/tmp/foo." (coerce *current-process-id* String)))
```

`*umask*` default file creation mode

Int

Sets the file creation mask of the Klone process. See the UMASK(2) unix manual page.

`*machine*` host machine type

Atom

Constant holding an atom telling which kind of machine/OS you are running. Currently defined atom are **`sun4`**, **`sunsolaris`**, and **`amiga`**, but others can be defined at compile time. The principle being that two binaries with the same **`*machine*`** tag should be linkable together, and the name being suitable to name a directory to hold the relevant binaries (so not including characters illegal in file names such as «**`/`**»).

1.19 Internal functions

These functions provide access to the internal engine of the interpreter. Some of them can break the Klone interpreter, and thus should be reserved to Klone experts, and will be flagged by the tag EXPERT.

`*version*` version number of the Klone library

string

This constant holds the string representing the version number of the klone interpreter. It should be of the form main-version-number <dot> minor-version-number release-letter, e.g.: **`"2.1q"`** where main-version is **`2`**, minor-version is **`1`**, and release is q.

`*application-name*` name of the application embedding the Klone interpreter

string

Default is “Klone”, but can be “klone” for the stand-alone klone interpreter, or “gwm”, “egeria”, etc...

`*max-stack-size*` maximum stack depth

number

Makes Klone trigger an error when the stack size becomes greater than this value (in machine words). Useful to catch unwanted infinite recursions, and on systems not providing a dynamically expandable C stack. Default is 32767 (128K).

`*:dsp` stack dump of debugging code INTERNAL

boolean

If unset (default), the stack frames belonging to debugging klone code (error handlers and correctors) is skipped. If set, it is also printed.

This option should only be used while testing and debugging klone debuggers such as **`kdb`**.

`C:&` return the address of the C pointer to a klone object INTERNAL

`(C:&` *object* **`)`**

returns the address (a number) of the object in the process memory.

C:ptr-size

the size of a C pointer in bytes — INTERNAL

number

holds the size of a C pointer, i.e. the size of a pointer to a klone object and a Klone integer. 4 for 32bit machines and 8 for 64 bit machines, usually.

C:*
C:*-int
C:*-byte

peeks at memory value — DANGER

(C:* *address* **)**

Returns the contents of memory at address (number), as a klone object for **C:***, a 32 bit signed int for **C:*-int,** and as an unsigned byte for **C:*-byte**

C:*=-int
C:*=-byte

pokes values into memory — DANGER

(C:*=-int *address value* **)**

Puts the value at address, either the 4 bytes of the 32 bit integer value for **C:*=-int** or the byte for **C:*=-byte**

evalhook
evalhook

control of the evaluator — CL p491

evalhook
(evalhook *form* **)**

The value of the ***evalhook*** variable is used to replace the **eval** primitive in the Klone evaluation engine. It can be set to either () (in which case the normal evaluation process is used), or to a lambda form, which will be called on each form to be evaluated to perform all the internal evaluations of Klone. To help in building such lambdas, the **evalhook** function is provided, which performs the raw eval once on its argument, but reclusive calls the user-redefined eval on all subsequent evaluations.

This mechanism is very handy to implement debuggers or run-time checks. The Klone debugger **wbd** is written in Klone on top of this primitive.

Example: how to trace all function calls:

```
(setq *evalhook*
 (lambda (form &aux result)
 (if (typep form List)
 (progn
        (? "Evaluating: " form "\n")
        (setq result (evalhook form))
        (? "Returns: " result "\n")
        result
 )
 (eval form)
)))

(+ 1 2 (- 7 8))
               ⇒ Evaluating: (+ 1 2 (- 7 8))
               ⇒ Evaluating: (- 7 8)
               ⇒ Returns: -1
               ⇒ Returns: 2
               → 2
```

stack-frame

dump the stack

(stack-frame *position* **)**

Returns a list of 5 elements describing the stack frame at *position* (offset in the stack), or 0 for the current stack frame (topmost frame of the stack). The format of this list is:.

offset	description	type
0	the function call which triggered this stack frame	list
1	the stack pointer of this frame (position)	int
2	the next stack pointer (or 0 at the bottom of stack)	int
3	a p-list of pairs (variable, old value) of local dynamic bindings	p-list
4	the type of the call: () for a Klone function (Expr), t for a C one (Subr)	bool
5	properties of the stack	p-list

Dumping the whole stack is thus recursively listing the frames, stopping when the next stack pointer is 0.

The properties (argument at offset **5**) is a p-list of parameters to describe info pertaining to the specific caller. In the core Klone, these properties are not used.

Internal: stack-frame omits to print the frames of the debugger code (or any code called via *evalhook*). If the library has been compiled in -DDEBUG mode, you can make it print all the frames, in order to debug a debugger for instance, by setting the symbol **`*:dsp`** to **`0`**.

loadinfo

information on current file being loaded

`(loadinfo)`

Returns a list of the name of the current file being loaded and the current line number, or nil if not called in a file being loaded.

meminfo gcinfo hashinfo klcfinfo

miscellaneous system information INTERNAL

functions without parameters

These functions print on the standard output informations about:

- meminfo: the memory used
- gcinfo: the number of garbageable objects
- hashinfo: information on the hash table of all symbols
- klcfinfo: information on the parser internal structures cache

Of these functions, only **`meminfo`** is of general use.

*:memory

memory allocation status

`(*:memory)`

Returns a list of three elements:

- the number of used bytes
- the number of free bytes, i.e. blocks of memory used by the process but still free of actual data
- a list of triplets, listing for each bucket the size (in bytes) of the bucket, the number of used blocks and the number of allocated but free blocks

```
(*:memory)
= (61392 28720 ((8 60 452) (16 355 157) (32 790 106) (64 36 28)
(128 48 16) (256 16 0) (512 0 8) (1024 1 3) (2048 2 4) (4096 3
0)))
```

^

object at address (de-referencing) INTERNAL EXPERT

(^ *name address comments* * **)**

Returns the Klone object which is supposed to be at memory location *address. name* and *comments* are silently ignored, but it is customary to put as name the name of the type of the object. This function is used to print in a re-readable form unprintable objects.

```
(coerce '(a 1) Hashtable)          → {^ Hashtable 0x74b80}
```

Warning: no verification is done of the validity of the address. An invalid address will almost surely result in a bus error or memory fault, terminating the application embedding Klone.

mleak
mleak-print

memory leak detection package INTERNAL

mleak
(mleak-print)

If your Klone library has been compiled with the -DMLEAK flag, this provides a way to detect malloc leaks (blocks that should be freed but are not). This should be done in conjunction with a C debugger. To test a code segment, say **(foo)** that should not leave allocated blocks, you should run something like:

```
(foo)
(setq mleak 1)
(foo)
(setq mleak 0)
(foo)
(mleak-print)
(break)
```

now, look at the printed list of still malloced blocks (if any), and re-run exactly the same code under the control of a C debugger with the C variable MLEAK_num set to the number of the block you want to stop when malloced, and a breakpoint set to the C function **MLEAK_break.**

mcheck

memory sanity check INTERNAL

(mcheck)
(mcheck *[* **t** | **()** *]* **)**

If the Klone library has been compiled with the -DMALLOCDEBUG option, the first form (call without arguments) performs an extensive check on all malloc-ed blocks, and the second one set the value of a flag which makes the system perform such a computation before each internal call to any memory allocation or de-allocation routine (malloc, free realloc and al.). In this case Klone runs approximately 20 times slower.

moncontrol

interface to the C profiler INTERNAL

(moncontrol 0|1)

If the Klone library has been compiled with profiling enabled (see Unix man pages for MONITOR(3)), **(moncontrol 1)** enables recording profiling information, and **(moncontrol 0)** stops it and dumps the result on a mon.out file.

flush-gc

triggers a garbage collection (GC) INTERNAL

(flush-gc *form* **)**

Evaluate *form* , discards its return value and garbage collects all the temporary objects that have been created by the evaluation of form. Not used normally, may be used in special cases (benchmarks).

***:prompt**

sets the prompt appearance of the standalone klone interpreter

list: **(** *main_prompt secondary_prompt level_fill result_prefix padding* **)**

When Klone is run as a standalone interpreter (if the application decides to use the standard C toplevel routines provided) then the prompt strings can be set by this list whose elements are strings.

At the toplevel will be printed:

```
<main_prompt><padding>
```

At a parenthesis level N:

```
<secondary_prompt><N * level_fill><padding>
```

printing the result will be preceded by:

```
<result_prefix>
```

The default for this list is:

```
("?" "?" " " " = " " ")
```

***:hash**
***:hash-version**

internal hash function

(*hash *object* **)**
***:hash-version**

Returns the hash number for *object* , as used internally by Klone for indexing in hashtables when using *object* as key. Two equal object **must** have the same hash number, even if they may not be eq, but two differing objects **can** have the same hash number of course.

***:hash-version** is a number indicating the version number of ***:hash**. If you store hash values in a file, you should archive the ***:hash-version** number with it since the hash function may change in the future.

1.20 Dynamic Loading

On systems supporting it (systems providing the **dlopen** unix call in this version), Klone provides a facility to dynamically load shared object modules. Shared modules are C modules written as if they would be linked to the Klone library, with optionally an initialisation C function named **KloneModuleInit** and a termination C function named **KloneModuleFini**, which are automatically called upon loading and closing of the module if present. You can test the availability of the dynamic loading facilities by testing if the symbol **DL:open** is bound.

For instance, on SUN OS, you can dynamically load a «KlMath» module by compiling it by:

```
cc -pic -c KlMath.c        # compile in position-independent (sharable)
ld -o KlMath.so KlMath.o -lm# generate a loadable .so
```

and then, in Klone:

```
? (setq KlMath.so (Dl:open "KlMath.so"))
 = {^ DlHandle 0x976e0}
? (setq expt (Dl:symbol KlMath.so "KlExpt" Subr 2))
 = {^ Subr 0x98280 E 2args}
? (expt 2 3)
 = 8.0
```

DL:open

dynamically load a sharable object module

(DL:open *module-filename* **)**

Dynamically loads a module (.so file) via the unix **dlopen** (3) system call. If a C function named **KloneModuleInit** exists in the module it is executed without arguments.

Returns a module handle which will be used to call the **DL:close** and **DL:symbol** functions.

NOTE: the module will be closed (unloaded) when the lasty reference to the module handle is destroyed, so you **MUST** store the returned handle in a variable.

```
(setq KlMath.so (Dl:open «KlMath.so»))
```

DL:close

unloads a dynamically loaded sharable object module

(DL:close *module-handle* **)**

Closes and unloads a module loaded by **DL:open**. Automatically called when the module handle is garbaged. Calls the C function **KloneModuleFini** with no arguments before unloading if defined in the module.

DL:symbol

binds a C symbol from a dynamically loaded sharable object module

(DL:symbol *module-handle C-name type arity* **)**

Searches into the module defined by its handle for a C symbol with name *C-name* , and creates a Klone object with it. Only C functions are supported for now, and you must provide the type (either **Subr** of **FSubr**, if you want Klone to evaluate or not the arguments before passing them to it), and the arity (number of arguments, can be **0** to **5**, or **-1** for N-ary functions accepting a variable number arguments via the argc, argv traditional C method. Returns the Klone function thus created

```
(setq KlMath.so (Dl:open «KlMath.so»))→ {^ DlHandle 0x976e0}
(setq expt (Dl:symbol KlMath.so «KlExpt» Subr 2))
expt                       → {^ Subr 0x98280 E 2args}
```

1.21 Errors

Error handling in Klone is treated by **error handlers**, i.e. user-defined functions which are called whenever an error occurs. Klone uses two lists of handlers (handlers can be functions or names of functions): ***error-correctors*** and ***error-handlers***. This system provides a powerful base to build sophisticated error handlers.

Standard errors are defined as atoms whose value is a string (or lambda) used to format the message via **print-format** (see page 52). This allows redefinition of error messages for internationalization.

```
Errors:UndefinedVariable → "undefined variable: %0"
(setq Errors:UndefinedVariable "variable indéfinie: %0")
   ;; for French error messages
```

`*error-handlers-orig*`
`*error-correctors*`
`*error-handlers*`

error handlers

lists-of-handlers (handlers are functions or function names)

When an error occurs, the elements of these lists are called in sequence with the error name and parameters (if any) as arguments. First, *error-correctors* is called, then *error-handlers*. *error-handlers-orig* contains the built-in error handler(s). Handlers are called with the error name (an atom who should have as value a string that can be used as a format to print the rest of arguments) and other arguments whose number and types depend on each error types. When an error handler is called, it can return either:

- **`:nil`** meaning that the handler could not correct or handle the error, and that the next handler should be applied.
- **`:true`** meaning that the error has been handled and that no other handler in the list should be applied, and the system should immediately return the control flow to the first error-trapping statement (or the top-level)
- *any-other-form* meaning that the handler successfully corrected the error and that the execution should resume normally, the statement having triggered the error returning this form as a result. For instance, the Klone autoloader automatically tracks errors of the type "undefined variable" and tries to load files defining this variable. If it succeed, it will just return the value of this variable and thus this variable has been transparently been loaded on the fly.

  Warning: only some error types can be safely returned from. See the list of error messages for details.

List of handlers are applied in reverse, from end to start, since it is more natural to treat list of handlers as stacks of handlers.

By convention, *error-correctors* should be used for non-interactive "behind the scene" correctors which will try to silently fix the error or return :nil otherwise (such as the autoloader). *error-handlers* is intended for holding only one interactive handler, which should play the role of a high level debugger for instance.

When running handlers of *error-correctors*, the handlers are still active, whereas when running handlers of *error-handlers*, *error-handlers* is reset to *error-handlers-orig*, and if an error happens into it, to a minimal built-in handler.

Warning: debugging Klone error handlers can be a tricky process, as errors in them often makes the interpreter go into intricate recursive handler calls. When an error occurs in an error handler, the default built-in handler is called, and if an error occurs in it then just the error name is printed, so recursion is error handlers should not happen, at the expense of less clear messages. Errors correctors on the other hand are not disabled, and thus can introduce recursive loops.

`error`

trigger an error — CL p666

`(error` *error-name args ** **`)`**

This function triggers the error handling mechanism (see above). The convention is that the *error-name* should be an atom whose value is a format string that can be used with **`print-format`** (see page 52) to print the rest of the arguments.

`break`

call the Klone debugger

`(break` *[expression [condition [data]]]* **`)`**

Triggers an error of type **`Errors:Break`** which invokes klone debugger if present. **`break`** return the evaluation of *expression* (so that a breakpoint can be set before a statement by including it as the expression of a break call). If condition is present, break will generate an error only if expression evals to a non-nil value. data is just passed as parameter of the error along with expression and condition.

`cbreak` call the C debugger INTERNAL

`(cbreak` *[non-evaluated-tag [condition [value]]]* **`)`**

If the C code of the Klone interpreter has been compiled with the -DDEBUG compilation flag, and the code is running under the control of a symbolic debugger (dbx, gdb), this function calls the C debugger via a call to the C function **`KlCBreakPoint`**, on which a breakpoint should be set from C. This function is only invoked if condition evals to true if present, and **`KlCBreakPoint`** is called with the two arguments *non-evaluated-tag* (which can be a string, atom, or integer) and *value* .

1.22 Error messages

Here are all the base klone error messages, listed (and sorted) by the symbol itself, then its default associated error text, and possibly comments.
Errors from which you can safely return (from an error corrector) are indicated by the tag: CORRECTABLE. The number of correctable errors may be increased in further releases.

Errors:BadAccesssorUse
"Access by %1 on %0 failed: %2"

Errors:BadArgType
"bad argument: %r0 (of type %t0, at position %1), expecting a %2"

Errors:BadClassDef
"Bad class definition for %0: %1"

Errors:BadCompareCall
"sort: compare function must return a number, not %0 (of type %t0)"

Errors:BadDefun
"bad definition of function: %0"

Errors:BadDoSyntax
"Bad syntax in iteration function"

Errors:BadHandler
"Bad handler for %0"

Errors:BadLambdalist
"Bad lambda list declaration: %0, parameter %1"

Errors:BadLocalSyntax
"bad local variable declaration"

Errors:BadNumberOfArgs
"bad number of arguments: %0"

Errors:BadRegexpr
"Error in regular expression: %0"

Errors:BadSymbchar
"No such character name: %0"

Errors:Break CORRECTABLE
"(break %0), condition %2 = %1"

Errors:CannotGetCValue
"Cannot get C value from %r0 (of type %t0)"

Errors:CannotSet
"Cannot set %r0 (of type %t0)"

Errors:DivideByZero
"Division by zero on %0"

Errors:Error
"Error"

Errors:ErrorCannotLoadFile CORRECTABLE
"file %0 not found"

Errors:ErrorOpeningFile
"Cannot open file %0 with mode %1"

Errors:InternalError
"Internal error: %0"

Errors:InvalidKeyword
"Invalid Keyword: %0"

Errors:MaxArity
"Maximum arity overflowed for function: %0"

Errors:NoCatch
"No catch named %0"

Errors:NoClass
"Not a Class: %0 (of type %t0)"

Errors:NoCoercion CORRECTABLE
"Cannot coerce %r0 (of type %t0) to a %1"

Errors:NoElement
"No element named %0 for %1"

Errors:NoMethod
"No method named %0 for class %1"

Errors:NoPreviousMethod
"call-next-method not called from inside a method"

Errors:NoPut
"cannot put in %0"

Errors:NoSlot
"No slot named %0 for class %1"

Errors:NonKlo
"%0 on a non klone object %1"

Errors:NotAMethod
"for selector %1, method %0 (of type %t0) is not an Expr or a Subr"

Errors:NotAnAtom
"Not an atom: %r0 (of type %t0)"

Errors:NotModifiable
"Cannot modify object: %r0 (of type %t0)"

Errors:NotRedefinable
"Cannot redefine object: %r0 (of type %t0)"

Errors:ReleasingAtom
"Internal error: trying to free atom: %0"

Errors:StackOverflow
"Stack overflow (%0 words). May be an infinite loop"

Errors:StreamError
"cannot %0 on stream %1"

Errors:SyntaxError
"%0"

Errors:TimeExceeded
"Evaluation took longer than %0 second(s) -- Aborted"

Errors:UndefinedFunction CORRECTABLE
"undefined function: %r0"

Errors:UndefinedMethod
"Undefined KLONE C method for object %r0 (of type %t0)"

Errors:UndefinedVariable CORRECTABLE
"undefined variable: %0"

KlErrorNumberOutOfRange
"Number out of range: %0, sould be between %1 and %2 inclusive"

1.23 Provided Packages

With the Klone distribution, comes some packages written in Klone implementing the following functionalities: You should browse all the available function before starting any non-toy work with Klone as these packages implements very useful functions.

A way to dump an ASCII text of the description of all available Klone external functions is by the **doc-autoloadables** function below:

doc-autoloadables

prints the complete documentation of autoloadable functions

(doc-autoloadables *[:paged lines]* **)**

lists on stdout the doc of all klone functions provided in the distribution or locally via the transparent autoloading mecanism If lines is not given, the docs are just printed on the standard output, if a page length in lines is given, it is formatted for prettier output for this page length.

To print onto file Autodocs.txt for 66 lines pages, do:

```
(with (*standard-output* (open "Autodocs.txt"
            :direction :output :if-exists :supersede))
    (doc-autoloadables :paged 66))
```

1.23.1 Common lisp compatibility

This package should normally be included in all running Klone applications. Using it will load the definition of this functions as Klone code, so they are not as efficient as native Klone primitives.

some
every

apply a predicate to each element of list CL p396

(some *predicate list +* **)**
(every *predicate list +* **)**

Both functions apply a function predicate to each element of list:

some applies the function until it returns a non-nil value and then returns this value; otherwise, if all lists are exhausted, nil is returned

every applies the function until it returns a nil value and then returns nil; otherwise, if all the lists are exhausted, every returns the last result obtained

The number of lists given should be equal to the number of arguments of the applied function. The function is applied **p** times, where **p** is the length of the first list given.If one of the other lists has a smaller size than the first one, it will be considered as a list of the same size (p elements) as the first, obtained by adding its last element to the end of the list as many times as necessary. If the first list is smaller than any other lists, the only first **p** elements of the other lists are referenced.

```
(defun numberp (var) (if (typep var Int) t))
(every 'numberp '(1 2 3))          → t
(some 'numberp '(a b c d))         → ()
```

provide
require

on-demand file loading CL p277

(provide *file* **)**
(require *file* **)**

These functions help to determine if a Klone file has been loaded, in order to avoid loading the file a second time.

The common argument *file* must be a symbol which defines the file using its print-name. The symbol *file* is then set to nil by either function.

require searches if the file is present in Klone, that is has been loaded. If the file is not present (that is, the symbol *file* is undefined) then it is loaded, else nothing is done. The assertion that a file is already loaded is given by the provide function.

provide asserts that the file has been loaded (essentially providing information to a subsequent call to require). This function is generally called in the loaded file (see example below).

```
;; Assume an initialization file: events.kl
(require 'network)      ;; The file network.kl is loaded
(require 'network)                     ;; Nothing is done
(provide 'events)       ;; Loading of events.kl is asserted
```

In this example, the file events.kl includes a provide function. If a subsequent application issues the following call:
(load "events")
the file events.kl will be loaded again. If however, the call:
(require 'events)
is used, then the file will not be reloaded due to the assertion provided when events.kl was loaded and executed the first time.

```
;; A definition of require and provide could be:
(defun require (file)
 (if (boundp file) () (progn (load file) (provide file))))
(defun provide (file)
 (if (boundp file) () (setq file ())))
```

---

defvar

declare a variable CL p86

(defvar *variable [initial-value [documentation]]* **)**

This function declares the specified *variable* to be global. Global variables may be bound dynamically by let constructs. *documentation* is a string associated with the global variable and may be retrieved with the **documentation** function. In the current implementation defvar is equivalent to defining a variable by setq and assign to it a documentation string.

The variable is only set if it is unbound, otherwise its value is untouched.

(defvar foo () "Not self-explanatory")

1.23.2 The Klone debugger: kdb

This symbolic klone debugger is entirely written in Klone. It is normally invoked when an error occurs, and must be loaded from the file **kdb.kl.**

---

kdb

klone interactive debugger

(kdb *on* **)**
(kdb)

(kdb t) sets the error handler to call kdb on an error, whereas (kdb ()) restores the previous error handler. (kdb) without arguments calls the debugger, from which breakpoints can be set for further execution.

The complete documentation of kdb is the on-line help under kdb itself, accessible by typing **help** under kdb (in answer to the prompt **kdb>**)

1.23.3 The simple autoloader: autoload

Taking advantage of the powerful error-recovery mechanism of Klone, the **auto-load.kl** file implements an error-corrector trying to correct errors of type undefined symbol (either Errors:UndefinedFunction or Errors:UndefinedVariable) by loading the file of the same name of the undefined symbol.

1.23.4 The smart autoloader: smartloader (default)

Extending on this autoload scheme, the provided default autoload mechanism, **smart-loader**, does basically the same thing, but looks for a dictionary file named **AUTOLOAD.-DIR** into each directory of the ***load-pathname*** for the symbol to find which file to load.

The **AUTOLOAD.DIR** file needs then to be updated each time a file is added, modified, or deleted in your path if you want it to be autoloadable. A klone script **klone-update** is provided in the distribution to update it.

Autoloadable symbols must be declared in the files by a line:

```
;;AUTOLOAD: symbol condition
```

To define the symbol foobar dynamically by loading a definition from a file, you would thus put in the file something like:

```
;;AUTOLOAD: foobar
(defun foobar (a b)...)
```

The symbol can optionally followed by a *condition* which must be defined in the current klone process for this symbol to be visible to the autoloader. It can be either a symbol, which should be defined, or a list which will be evaluated, the file being loadable only if the expression evals to a non-nil value.

For instance, if we define a “map-all-windows” function only usable under gwm, and that the variable *gwm-version* is only defined under gwm, a gwm autoloadable function could be declared as:

```
;;AUTOLOAD: map-all-windows *gwm-version*
```

1.23.5 The smart autodocumentation: smartdoc

In the same fashion as the smartloader package, you can define documentation of functions in your loadable file that can be searched and listed at run-time without consuming memory space in the executable as does the traditional lisp way of using documentation strings attached to symbols. For this you need to put into your files the line:

```
;;AUTODOC: symbol "descriptive text" condition
```

immediately followed by the actual text of the documentation which can be in three forms:

- Comments: every line beginning by a semicolon `;` is used until a non-comment line is reached
- String: the text can be a single string, with the closing quote being followed by nothing but a newline
- Free form: the doc is the characters enclosed between two lines of at least 8 minus characters.

The descriptive text must be a quote-enclosed string describing roughly the command, which will be searched for keywords by the **apropos** command. condition is either a symbol or a list, as for the one used by the smartloader above.

The klone-update Unix command updates the AUTODOC.DIR index as well as the AUTOLOAD.DIR index

apropos

searches for keywords in the smartdoc descriptions

(apropos *text* [**:from** {**doc**| **code**}] **)**

Does a regular-expression search for text as a pattern on all the symbols and descriptive text declared by the **;;AUTODOC:** declarations. Prints the resulting matches in the unix apropos style, i.e. the symbol name and the text separated by a minus sign. Returns the number of matches.

```
(apropos parse)
   ⇒ load - parses and execute a Klone file
   ⇒ getopts - parses command line arguments
        → 2
```

The keyword **:from** restricts the search to either the built-in functions, documented in this document with the value **doc**, or to the autoloadable Klone functions available in the autoloadable Klone files with the value **code**.

```
;; To see what is available in the misc. Klone files:
(apropos "" :from code)
```

help
smartdoc:help

find the documentations for a symbol

(help *symbol-name* **)**
(smartdoc:help *symbol-name* **)**

Does an exact search for documentations associated with the symbol *symbol-name* , and prints the documentations found, separated by rows of stars, for **help**, and returns a list of documentations as strings for **smartdoc:help**. All documentations should have no line longer than 80 characters.

```
(help make-list)          ⇒
*************************************************************
## make-list

Desc: construct a list of a given size

Comments: CL p418

Usage: (make-list size [ :initial-element form ] )

This function constructs and returns a list containing size ele-
ments, where each element is initialized to form, if present, or
nil otherwise.

(make-list 3 :initial-element 'a) (a a a)

*************************************************************
```

klone-update

klone script to update the index files for smartloader and smartdoc systems

klone-update *[options] [directories *]*

This is an Unix executable script using the stand-alone Klone interpreter, klone, supposedly installed somewhere in your **PATH**[1]. it updates the index files **AUTOLOAD.DIR** and **AUTODOC.DIR** in the current directory if no argument is given, or in *directories* . All subdirectories of the scanned directories are automatically and recursively scanned, but only one index file summing up all the information is built at the upmost level.

Options can be:

1. you must edit the first line of this script to make it reference the actual location of the interpreter, or use the unix command: **klone klone-update arguments...**

-v verbose operation, prints a lot of intermediate messages

-path updates all directories in *load-pathname*

The command looks for files **AUTOLOAD.PAT** and **AUTODOC.PAT** in each scanned directory to obtain a list of regular expressions (one per line) that file names must match to be scanned by the process. An empty file thus disables scanning in the directory, the default pattern being **".*[.]kl"** if these files are not present.

trace untrace

trace invocation of Klone functions

(trace *function [:entry entry-expr] [:exit exit-expr]* **)**
(untrace *functions** **)**
(trace)

This provide a simple way of tracing function execution. The keywords **:entry** and **:exit** may define expressions evaluated when entering or exiting the function. The value **#trace:value** is the value returned by the function. **untrace** removes tracing on a function.

(trace) returns the list of traced functions

```
(trace +)              ; trace calls to + in Fibonacci
(fib 3)
[0]+ --> ((fib (- n 1)) (fib (- n 2)))
 [1]+ --> ((fib (- n 1)) (fib (- n 2)))
 [1]+ <-- 1
[0]+ <-- 2
```

trace-all

trace execution of expressions

(trace-all *[t|()]* **)**

When trace mode is enabled (with argument **t** to **trace**), all statements are printed before execution (prefixed by **<=)**, as well as he retuned value (prefixed by **<=)** and the stack level (between square brackets **[]**). Value of evaluated variable are displayed only if they differ from the last time they were printed, prefixed by **|=**. This is very useful in scripts. **(trace-all ())** disables the tracing.

```
(trace-all t)
(fib 1)
=>[0] (fib 1)
=>[1] (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))
=>[2] (< n 2)
|= n = 1
<=[2] t
<=[1] 1
<=[0] 1
```

sh-package

automatic execution of shell commands

(require 'sh-package)
(sh:*shell-command arguments...* **)**

Once required (loaded), Klone will attempt to execute synchronously (wait for the result) all functions prefixed by the package **sh:** as shell commands of the same name, e.g. you can write:

```
(sh:date)              ;; forks the date command
(sh:rm "-i" "foo")     ;; to remove interactively file
```

sh
sh:open

simplified interface to system to execute shell commands

(sh *command arguments...* **)**
(sh:open *command arguments...* **)**

sh provides a natural way to write simple synchronous calls to the underlying operating system via the **system** (see page 58) command. command can be specified in a free-form way (as atoms, strings or numbers that will be converted to strings) and sh will invoke directly the command or fork a shell to parse the whole line of arguments if it detects special characters to be interpreted by the shell such as **$ | > <**, etc...

```
;; no intermediate shell used
(sh date)
(sh ypcat passwd)
;; intermediate shell used
(sh ypcat passwd | grep -i colas |sort -2 >/tmp/foo)
```

sh quotes its arguments so you have to unquote (by inserting a comma before the argument) of items you want to evaluate, just like you have to prefix by $ shell variables.

```
(setq file (+ "/tmp/gee." (String *current-process-id*)))
(sh date > ,file)
```

WARNING: arguments are quoted, thus not evaluated, but they are still parsed by Klone, so take care of symbols having a meaning for the Klone parser (**" ' () {}**), or items not printing the same way they are read:

```
(sh sort +3 foo)        ;; DOES NOT WORK: +3 is printed 3
                        ;; klone tries to execute "sort 3 foo"
(sh sort "+3" foo)      ;; works
```

sh waits for the command to complete, and returns the error code if an error occurs, or () if everything went Ok. **sh:open** opens a stream on the command output and returns it.

```
;; read the date in a string
(setq date (String (sh:open date)))
```

Index

Index

Index