Implementation Notes for GNU CLISP

These notes document CLISP version 2.43

Bruno Haible

Michael Stoll

Sam Steingold

See *COPYRIGHT* for the list of other contributors and the license.

CLISP Release History		
Release **1**	**April 1987 - July 1992**	
- The project was started when both original authors, Bruno Haible and Michael Stoll, were students in Germany. - The original version was for **Atari** ST only, written in 68000 assembly language and Common Lisp.		
Release **2.0**	**1992-10-09**	
comp.os.linux announcement (**Linux** binaries only)		
Release **2.1**	**1993-01-01**	
- The first portable release, with source, released under GNU GPL. - Supported platforms: **Atari** ST, **Amiga** 500-2000, **DOS** (emx, djgpp), **OS/2** (emx), Unix (**Linux**, Sun4, Sun386, HP9000/800).		
Release **2.1.1**	**1993-01-11**	
Release **2.1.2**	**1993-02-01**	
Release **2.1.3**	**1993-02-03**	
Release **2.2**	**1993-02-21**	
Add test suite.		
Release **2.2.1**	**1993-03-04**	
Release **2.2.2**	**1993-03-19**	
`CUSTOM:*EDITOR*`		
Release **2.3**	**1993-03-30**	

- LOAD-TIME-VALUE
- EXT:DEFAULT-DIRECTORY

Release **2.3.1**	**1993-04-05**	
Release **2.4**	**1993-05-24**	

- DEFPACKAGE
- FUNCTION-LAMBDA-EXPRESSION
- Section 32.1, “Random Screen Access”

Release **2.5**	**1993-06-29**	

- SETF function names.
- PRINT-UNREADABLE-OBJECT
- SYMBOL-MACROLET

Release **2.5.1**	**1993-07-17**	

- immutable objects

Release **2.6**	**1993-08-22**	

- **“CLOS”** package: DEFCLASS, DEFMETHOD, DEFGENERIC, GENERIC-FUNCTION, CLOS:GENERIC-FLET, CLOS:GENERIC-LABELS, WITH-SLOTS, WITH-ACCESSORS, FIND-CLASS, (SETF FIND-CLASS), CLASS-OF, CLASS-NAME, (SETF CLASS-NAME), SLOT-VALUE, SLOT-BOUNDP, SLOT-MAKUNBOUND, SLOT-EXISTS-P, CALL-NEXT-METHOD, NEXT-METHOD-P, NO-APPLICABLE-METHOD, CLOS:NO-PRIMARY-METHOD, NO-NEXT-METHOD, FIND-METHOD, ADD-METHOD, REMOVE-METHOD, COMPUTE-APPLICABLE-METHODS, METHOD-QUALIFIERS, FUNCTION-KEYWORDS, SLOT-MISSING, SLOT-UNBOUND, PRINT-OBJECT, DESCRIBE-OBJECT, MAKE-INSTANCE, INITIALIZE-INSTANCE, REINITIALIZE-INSTANCE, SHARED-INITIALIZE

Release **2.6.1**	**1993-09-01**	
Release **2.7**	**1993-09-27**	

• top-level forms • DECLAIM		
Release **2.8**	**1993-11-08**	
• **“COMMON-LISP”**, **“COMMON-LISP-USER”** • New module: STDWIN		
Release **2.9**	**1994-01-08**	
• DEFINE-CONDITION, IGNORE-ERRORS, HANDLER-CASE, HANDLER-BIND, RESTART-CASE, EXT:WITH-RESTARTS, WITH-SIMPLE-RESTART, RESTART-BIND, WITH-CONDITION-RESTARTS, RESTART, CONDITION, SERIOUS-CONDITION, ERROR, PROGRAM-ERROR, CONTROL-ERROR, ARITHMETIC-ERROR, DIVISION-BY-ZERO, FLOATING-POINT-OVERFLOW, FLOATING-POINT-UNDERFLOW, CELL-ERROR, UNBOUND-VARIABLE, UNDEFINED-FUNCTION, TYPE-ERROR, PACKAGE-ERROR, STREAM-ERROR, END-OF-FILE, FILE-ERROR, STORAGE-CONDITION, WARNING, SIMPLE-CONDITION, SIMPLE-ERROR, SIMPLE-TYPE-ERROR, SIMPLE-WARNING, MAKE-CONDITION, SIGNAL, COMPUTE-RESTARTS, FIND-RESTART, INVOKE-RESTART, INVOKE-RESTART-INTERACTIVELY, ABORT, CONTINUE, MUFFLE-WARNING, STORE-VALUE, USE-VALUE, INVOKE-DEBUGGER, RESTART-NAME, ARITHMETIC-ERROR-OPERATION, ARITHMETIC-ERROR-OPERANDS, CELL-ERROR-NAME, TYPE-ERROR-DATUM, TYPE-ERROR-EXPECTED-TYPE, PACKAGE-ERROR-PACKAGE, STREAM-ERROR-STREAM, FILE-ERROR-PATHNAME, EXT:SIMPLE-CONDITION-FORMAT-STRING, EXT:SIMPLE-CONDITION-FORMAT-ARGUMENTS, *BREAK-ON-SIGNALS*, *DEBUGGER-HOOK*, *PRINT-READABLY*		
Release **2.10**	**1994-06-22**	
• EXT:READ-CHAR-SEQUENCE, EXT:WRITE-CHAR-SEQUENCE, EXT:READ-BYTE-SEQUENCE, EXT:WRITE-BYTE-SEQUENCE • Section 31.6, “Generic streams”		
Release **2.11**	**1994-07-04**	

- LOOP, LOOP-FINISH, MAP-INTO
- LEAST-POSITIVE-NORMALIZED-SHORT-FLOAT, LEAST-NEGATIVE-NORMALIZED-SHORT-FLOAT, LEAST-POSITIVE-NORMALIZED-SINGLE-FLOAT, LEAST-NEGATIVE-NORMALIZED-SINGLE-FLOAT, LEAST-POSITIVE-NORMALIZED-DOUBLE-FLOAT, LEAST-NEGATIVE-NORMALIZED-DOUBLE-FLOAT, LEAST-POSITIVE-NORMALIZED-LONG-FLOAT, LEAST-NEGATIVE-NORMALIZED-LONG-FLOAT

Release **2.12**	**1994-08-23**	

- generational garbage-collection
- DESTRUCTURING-BIND
- EXT:UNCOMPILE

Release **2.12.1**	**1994-09-01**	
Release **2.13**	**1994-10-26**	

- WILD-PATHNAME-P, PATHNAME-MATCH-P, TRANSLATE-PATHNAME, LOGICAL-PATHNAME, LOGICAL-PATHNAME-TRANSLATIONS, TRANSLATE-LOGICAL-PATHNAME, LOAD-LOGICAL-PATHNAME-TRANSLATIONS, COMPILE-FILE-PATHNAME

Release **2.13.1**	**1995-01-01**	
Release **2.14**	**1995-04-04**	

- **“FFI”**
- ROW-MAJOR-AREF, DELETE-PACKAGE, EXT:MUFFLE-CERRORS, EXT:APPEASE-CERRORS, EXT:EXIT-ON-ERROR

Release **2.15**	**1995-04-25**	

- New modules: **wildcard**, **regexp**
- FORMATTER, EXT:FINALIZE
- FILE-STREAM, SYNONYM-STREAM, BROADCAST-STREAM, CONCATENATED-STREAM, TWO-WAY-STREAM, ECHO-STREAM, STRING-STREAM, OPEN-STREAM-P, SYNONYM-STREAM-SYMBOL,

BROADCAST-STREAM-STREAMS, CONCATENATED-STREAM-STREAMS, TWO-WAY-STREAM-INPUT-STREAM, TWO-WAY-STREAM-OUTPUT-STREAM, ECHO-STREAM-INPUT-STREAM, ECHO-STREAM-OUTPUT-STREAM, PRINT-NOT-READABLE, PRINT-NOT-READABLE-OBJECT

Release **2.16**	**1995-06-23**	

- COMPLEMENT, WITH-STANDARD-IO-SYNTAX, DYNAMIC-EXTENT, λ, IGNORABLE, CONSTANTLY
- WITH-HASH-TABLE-ITERATOR, HASH-TABLE-REHASH-SIZE, HASH-TABLE-REHASH-THRESHOLD, HASH-TABLE-SIZE, HASH-TABLE-TEST

Release **2.17**	**1996-07-21**	

- SOCKET:SOCKET-SERVER, SOCKET:SOCKET-SERVER-CLOSE, SOCKET:SOCKET-SERVER-PORT, SOCKET:SOCKET-WAIT, SOCKET:SOCKET-ACCEPT, SOCKET:SOCKET-CONNECT, SOCKET:SOCKET-STREAM-HOST, SOCKET:SOCKET-STREAM-PORT, SOCKET:SOCKET-SERVICE-PORT, SOCKET:SOCKET-STREAM-PEER

Release **2.17**	**1996-07-22**	
Release **2.18**	**1997-05-03**	

- I18N:DEFLANGUAGE, I18N:DEFINTERNATIONAL, I18N:DEFLOCALIZED
- CUSTOM:*LOAD-COMPILING*

Release **2.19**	**1997-08-07**	

- **CLX**

Release **2.20**	**1997-09-25**	

- *READ-EVAL*
- EXT:TIMES

Release **2.20.1**	**1997-12-06**	
Release **2.21**	**1998-09-09**	

- Removed module STDWIN.
- CUSTOM:*WARN-ON-FLOATING-POINT-CONTAGION*, CUSTOM:*FLOATING-POINT-CONTAGION-ANSI*, FLOATING-POINT-INEXACT, FLOATING-POINT-INVALID-OPERATION
- EXT:PROBE-DIRECTORY, ENSURE-DIRECTORIES-EXIST
- *PRINT-RIGHT-MARGIN*, ARRAY-DISPLACEMENT,
- BOOLEAN, COPY-STRUCTURE, GENERIC-FUNCTION, STRUCTURE-OBJECT, CLASS, METHOD, SPECIAL-OPERATOR-P

Release **2.22**	**1999-01-08**	

- BASE-CHAR, EXTENDED-CHAR, BASE-STRING, SIMPLE-BASE-STRING
- GET-SETF-EXPANSION, DEFINE-SETF-EXPANDER
- PARSE-ERROR, READER-ERROR
- UNBOUND-SLOT-INSTANCE
- SOCKET:SOCKET-STREAM-LOCAL, SOCKET:SOCKET-SERVER-HOST

Release **2.23**	**1999-07-22**	

- New module: **postgresql**
- **UNICODE**, **“CHARSET”**, CUSTOM:*DEFAULT-FILE-ENCODING*, CUSTOM:*PATHNAME-ENCODING*, CUSTOM:*TERMINAL-ENCODING*, CUSTOM:*MISC-ENCODING* CUSTOM:*FOREIGN-ENCODING*, AFFI:*FOREIGN-ENCODING*
- Chapter 30, *Gray streams*
- STREAM-EXTERNAL-FORMAT
- WITH-PACKAGE-ITERATOR
- ALLOCATE-INSTANCE
- Section 31.7.1, “Weak Pointers”
- EXT:READ-INTEGER, EXT:WRITE-INTEGER
- EXT:SIMPLE-CONDITION-FORMAT-CONTROL

Release **2.24**	**2000-03-06**	

• EXT:READ-FLOAT, EXT:WRITE-FLOAT • EXT:CHAR-WIDTH, EXT:STRING-WIDTH • WITH-COMPILATION-UNIT		
Release **2.25**	**2001-03-15**	
• INSPECT, EXT:CLHS • EXT:CONVERT-STRING-FROM-BYTES, EXT:CONVERT-STRING-TO-BYTES • EXT:READ-BYTE-LOOKAHEAD, GRAY:STREAM-READ-BYTE-LOOKAHEAD, EXT:READ-BYTE-WILL-HANG-P, GRAY:STREAM-READ-BYTE-WILL-HANG-P, EXT:READ-BYTE-NO-HANG, GRAY:STREAM-READ-BYTE-NO-HANG • **Win32** improvements (unc pathnames, registry, screen)		
Release **2.25.1**	**2001-04-06**	
Release **2.26**	**2001-05-23**	
• dropped CLtL1, added #+LISP=CL to *FEATURES* • DEFINE-COMPILER-MACRO • UPGRADED-COMPLEX-PART-TYPE • EXT:RUN-SHELL-COMMAND, EXT:RUN-PROGRAM accept :WAIT • compiler checks function call signatures		
Release **2.27**	**2001-07-17**	
• (SETF EXT:GETENV) • src/install.bat		
Release **2.28**	**2002-03-03**	
• [ANSI CL standard] Pretty-Printer • MAKE-LOAD-FORM, MAKE-LOAD-FORM-SAVING-SLOTS • Section 31.7.9, “Weak Hash Tables” • EXT:FCASE • I18N:GETTEXT		
Release **2.29**	**2002-07-25**	

• Bug-fix/portability: **gcc** 3.1 etc		
Release **2.30**	**2002-09-15**	
• Do not bundle GNU libiconv, GNU gettext, GNU readline, GNU libsigsegv • `CHARSET:UCS-4` • `CUSTOM:*PARSE-NAMESTRING-DOT-FILE*` • `SOCKET:SOCKET-STREAM-SHUTDOWN` • `POSIX:STREAM-LOCK`, `POSIX:COPY-FILE`, `POSIX:DUPLICATE-HANDLE` • New module: **`oracle`**		
Release **2.31**	**2003-09-01**	
• New modules: **`fastcgi`**, **`dirkey`**, **`bindings/win32`**, **`syscalls`**, **`netica`** • Support modules on **Win32**. • **UNICODE** 3.2 • New backquote implementation. • Many [ANSI CL standard] compliance fixes. • More **“FFI”** functionality.		
Release **2.32**	**2003-12-29**	
• support **LFS** • New modules: **`berkeley-db`**, **`pcre`**		
Release **2.33**	**2004-03-17**	
• `CUSTOM:*APROPOS-MATCHER*`, `EXT:MOD-EXPT`, `EXT:ARGV` `GRAY:STREAM-POSITION` • `DEFINE-METHOD-COMBINATION`, • Portability: removed **Acorn** and **Amiga** support, fixed UNIXes.		
Release **2.33.1**	**2004-05-22**	

- Bug-fixes, portability: **gcc** 3.4

Release **2.33.2**	**2004-06-02**	

- Portability: RedHat Fedora **Linux**/x86

Release **2.34**	**2005-07-20**	

- Chapter 29, *Meta-Object Protocol*
- Section 31.7, “Weak Objects”
- Section 11.4, “Package Case-Sensitivity”
- `EXT:SET-GLOBAL-HANDLER`, `EXT:WITHOUT-GLOBAL-HANDLERS`
- Portability: removed **DOS** and **OS/2** support.
- New modules: **`matlab`**, **`rawsock`**, **`zlib`**, **`i18n`**, **`pari`**.

Release **2.35**	**2005-08-29**	

- `EXT:COMPILED-FILE-P`
- `EXT:CHAR-INVERTCASE`, `EXT:STRING-INVERTCASE`, `EXT:NSTRING-INVERTCASE`
- `POSIX:STREAM-OPTIONS`
- Close all file descriptors before `exec`.

Release **2.36**	**2005-12-04**	

- `EXT:OPEN-HTTP`, `EXT:NOTSPECIAL`, `FFI:DEF-C-CONST`, `BASE64`
- `modules/new-clx/demos/koch.lisp`
- `src/spvw_sigterm.d`

Release **2.37**	**2006-01-02**	

- `SOCKET:SOCKET-SERVER` accepts `:INTERFACE` and `:BACKLOG`.
- Fixed `(SETF EXT:GETENV)`.

Release **2.38**	**2006-01-24**	

• EXT:SAVEINITMEM creates standalone executables.		
Release **2.39**	**2006-07-16**	
• Reliable stack overflow detection and recovery.		
Release **2.40**	**2006-09-23**	
• Keep doc string and lambda list in the closure object.		
Release **2.41**	**2006-10-13**	
• FFI:DEFAULT-FOREIGN-LIBRARY • New module: **libsvm**		
Release **2.42**	**2007-10-16**	
• Section 8.2, "The structure Meta-Object Protocol." • EXT:RENAME-DIR • Many additions to modules/new-clx/demos/ • New modules: **gtk2**, **gdbm**		
Release **2.43**	**2007-11-18**	
• The **CLISP** build process is now in compliance with the GNU standards. • Use **gnulib-tool** to sync with gnulib.		

Abstract

This document describes the GNU **CLISP** - an implementation of the [ANSI CL standard].

See the section called "Bugs" for instructions on how to report bugs (both in these notes and in **CLISP** itself).

See Q: A.1.1.5 for information on **CLISP** support.

Table of Contents

List of Figures

List of Tables

List of Examples

Overview

These notes discuss the **CLISP** implementation of **Common Lisp** by Bruno Haible and Michael Stoll. The current maintainers are Bruno Haible and Sam Steingold.

This implementation is mostly conforming to the [ANSI CL standard] available on-line as the [Common Lisp HyperSpec] (but the printed ANSI document remains the authoritative source of information). [ANSI CL standard] supersedes the earlier specifications [CLtL1] and [CLtL2].

The first part of these notes, Part I, "Chapters or the Common Lisp HyperSpec", is indexed in parallel to the [Common Lisp HyperSpec] and documents how **CLISP** implements the standard [ANSI CL standard].

The second part, Part II, "Common Portable Extensions", documents the common extensions to the [ANSI CL standard], specifically Meta-Object Protocol and **"GRAY"** `STREAM`s.

The third part, Part III, “Extensions Specific to **CLISP**”, documents the **CLISP**-specific extensions, e.g., Section 32.5, “Socket Streams”.

The fourth part, Part IV, “Internals of the **CLISP** Implementation”, is intended mostly for developers as it documents the **CLISP** internals, e.g., garbage-collection, adding new built-ins, and the bytecodes generated by the compiler (i.e., what is printed by `DISASSEMBLE`).

Conventions

The following is the mark-up notations used in this document:

Table 1. Mark-up conventions

Object Kind	Example
Function	`CAR`
Variable	`CUSTOM:*LOAD-PATHS*`
Formal Argument	*x*
Keyword	`:EOF`
Number	`0`
Character	#\Newline
Class, type	`REGEXP:MATCH`
Format instruction	`~A`
Standard lambda list keyword	`&KEY`
Declaration	`FTYPE`
Package	**“COMMON-LISP-USER”**
Real file	`config.lisp`
Abstract file	`#P".c"`

Object Kind	Example
Code (you are likely to type it)	`(CONS 1 2)`
Data (**CLISP** is likely to print it)	`#(1 2 3)`

Program listing	`(defun cycle-length (n &OPTIONAL (len 1) (top 0)`
	`  (cond ((= n 1) (values len top))`
	`        ((evenp n) (cycle-length (ash n -1) (1+`
	`        (t (let ((next (1+ (* 3 n))))`
	`             (cycle-length next (1+ len) (max to`

Bytecode instruction	`(STOREV` *`k m`*`)`
First mention of an entity	***firstterm***
External module	**`libsvm`**, **`bindings/glibc`**

Part I. Chapters or the [Common Lisp HyperSpec]

Table of Contents

Chapter 1. Introduction [CLHS-1]

Table of Contents

1.1. Special Symbols [CLHS-1.4.1.3]

The ***final delimiter*** of an interactive stream:

UNIX
: type **Control**+D at the beginning of a line

Win32
: type **Control**+Z, followed by **Return**

This final delimiter is never actually seen by programs; no need to test for #\^D or #\^Z - use `READ-CHAR-NO-HANG` to check for `end-of-stream`. Calling `CLEAR-INPUT` on the stream removes the `end-of-stream` state, thus making it available for further input.

A newline character can be entered by the user by pressing the **Newline** key or, on the numeric keypad, the **Enter** key.

1.2. Error Terminology [CLHS-1.4.2]

Safety settings are ignored by the interpreted code; therefore where the standard uses the phrase “should signal an error”, an `ERROR` is `SIGNAL`ed. See Section 3.3.2, “Declaration `SAFETY`” for the safety of compiled code.

1.3. Symbols in the Package “COMMON-LISP” [CLHS-1.9]

All 978 symbols in the **“COMMON-LISP”** package specified by the [ANSI CL standard] are implemented.

Chapter 2. Syntax [CLHS-2]

Table of Contents

2.1. Standard Characters [CLHS-2.1.3]

The standard characters are #\Newline and the graphic characters with a `CODE-CHAR` between 32 and 126 (inclusive).

2.2. Reader Algorithm [CLHS-2.2]

The requirement of step 4 that a “reader macro function may return zero values or one value” **is** enforced. You can use the function `VALUES` to control the number of values returned.

2.3. Symbols as Tokens [CLHS-2.3.4]

A ***reserved token*** , i.e., a token that has potential number syntax but cannot be interpreted as a `NUMBER`, is interpreted as `SYMBOL` when being read.

2.4. Valid Patterns for Tokens [CLHS-2.3.5]

When a token with package markers is read, then no checking is done whether the package part and the symbol-name part do not have number syntax. (What's the purpose of this check?) So we consider tokens like USER:: or :1 or LISP::4711 or 21:3 as symbols.

2.5. Backquote [CLHS-2.4.6]

The backquote read macro also works when nested. Example:

```
   (EVAL ``(,#'(LAMBDA () ',a) ,#'(LAMBDA () ',b)))
≡ (EVAL `(list #'(LAMBDA () ',a) #'(LAMBDA () ',b)))
≡ (EVAL (list 'list (list 'function (list 'lambda nil (l
                     (list 'function (list 'lambda nil (l
```

2.6. Sharpsign [CLHS-2.4.8]

Reader macros are also defined for the following:

Additional reader macros

#,
: load-time evaluation, kept despite the [ANSI CL standard] issue SHARP-COMMA-CONFUSION:REMOVE.

#Y
: compiled `FUNCTION` objects and input `STREAM`'s `EXT:ENCODING`s

#""
: `PATHNAME`: `#"test.lisp"` is the value of `(PATHNAME "test.lisp")`

2.6.1. Sharpsign Backslash [CLHS-2.4.8.1]

#\Code allows input of characters of arbitrary code: e.g., #\Code231 reads as the character `(CODE-CHAR 231.)`.

2.6.2. Sharpsign Less-Than-Sign [CLHS-2.4.8.20]

This is the list of objects whose external representation cannot be meaningfully read in:

Unreadable objects

#<type ...>
: all `STRUCTURE-OBJECT`s lacking a keyword constructor

#<ARRAY *type dimensions*>
: all ARRAYs except STRINGs, if *PRINT-ARRAY* is NIL

#<SYSTEM-FUNCTION *name*>
: built-in function written in **C**

#<ADD-ON-SYSTEM-FUNCTION *name*>
: module function written in **C**

#<SPECIAL-OPERATOR *name*>
: special operator handler

#<COMPILED-CLOSURE *name*>
: compiled function, if CUSTOM:*PRINT-CLOSURE* is NIL

#<CLOSURE *name* ...>
: interpreted function

#<FRAME-POINTER #x...>
: pointer to a stack frame

#<DISABLED POINTER>
: frame pointer which has become invalid on exit from the corresponding BLOCK or TAGBODY

#<...STREAM...>
: STREAM

#<PACKAGE *name*>
: PACKAGE

#<HASH-TABLE #x...>
: HASH-TABLE, if *PRINT-ARRAY* is NIL

#<READTABLE #x...>
: READTABLE

#<SYMBOL-MACRO *form*>
: SYMBOL-MACRO handler

#<MACRO *function*>
: macro expander (defined by DEFMACRO and friends)

#<FFI:FOREIGN-POINTER #x...>
: foreign pointer (Platform Dependent: **UNIX**, **Win32** platforms only.)

#<FFI:FOREIGN-ADDRESS #x...>
: foreign address (Platform Dependent: **UNIX**, **Win32** platforms only.)

#<FFI:FOREIGN-VARIABLE *name* #x...>
: foreign variable (Platform Dependent: **UNIX**, **Win32** platforms only.)

#<FFI:FOREIGN-FUNCTION *name* #x...>
: foreign function (Platform Dependent: **UNIX**, **Win32** platforms only.)

#<UNBOUND>
: “value” of an unbound symbol, an unsupplied optional or keyword argument

#<SPECIAL REFERENCE>
: environment marker for variables declared `SPECIAL`

#<DOT>
: internal `READ` result for "."

#<END OF FILE>
: internal `READ` result, when the `end-of-stream` is reached

#<READ-LABEL ...>
: intermediate `READ` result for `#n#`

#<ADDRESS #x...>
: machine address, should not occur

#<SYSTEM-POINTER #x...>
: should not occur

Chapter 3. Evaluation and Compilation [CLHS-3]

Table of Contents

All the functions built by `FUNCTION`, `COMPILE` and the like are atoms. There are built-in functions written in **C**, compiled functions (both of type `COMPILED-FUNCTION`) and interpreted functions (of type `FUNCTION`).

3.1. Evaluation [CLHS-3.1]

3.1.1. Introduction to Environments [CLHS-3.1.1]

Macro `EXT:THE-ENVIRONMENT`. As in Scheme, the macro `(EXT:THE-ENVIRONMENT)` returns the current lexical environment. This works only in interpreted code and is not compilable!

Function `(EXT:EVAL-ENV` *`form`* `&OPTIONAL` *`environment`*`).` evaluates a form in a given lexical environment, just as if the form had been a part of the program that the *`environment`* came from.

3.1.2. Dynamic Variables [CLHS-3.1.2.1.1.2]

“Undefined variables”, i.e. variables which are referenced outside any lexical binding for a variable of the same name and which are not declared `SPECIAL`, are treated like dynamic variables in the global environment. The compiler `SIGNAL`s a `WARNING` when it encounters an undefined variable.

3.1.3. Conses as Forms [CLHS-3.1.2.1.2]

Lists of the form `((SETF symbol) ...)` are also treated as function forms. This makes the syntax `(function-name arguments ...)` consistent with the syntax `(FUNCALL #'function-name arguments ...)`. It implements the item 7 of the [ANSI CL standard] issue FUNCTION-NAME:LARGE and the definition of function forms, and is consistent with the use of function names elsewhere in **Common Lisp**.

3.2. Compilation [CLHS-3.2]

3.2.1. Compiler Terminology [CLHS-3.2.1]

CLISP compiles to platform-independent bytecode.

3.2.2. Compiler Macros [CLHS-3.2.2.1]

Compiler macros are expanded in the compiled code only, and ignored by the interpreter.

3.2.3. Semantic Constraints [CLHS-3.2.2.3]

Non-conforming code that does not follow the rule

> “Special proclamations for dynamic variables must be made in the compilation environment.”

can produce quite unexpected results, e.g., observable differences between *compiled* and *interpreted* programs:

```
(defun adder-c (value) (declare ((COMPILE)) (lambda (x)
⇒ ADDER-C  ; compiled function; value is lexical
(defun adder-i (value) (lambda (x) (+ x value)))
⇒ ADDER-I  ; interpreted function; value is lexical
(setq add-c-10 (adder-c 10))
⇒ ADD-C-10 ; compiled function
(setq add-i-10 (adder-i 10))
⇒ ADD-I-10 ; interpreted function
(funcall add-c-10 32)
⇒ 42 ; as expected
(funcall add-i-10 32)
⇒ 42 ; as expected
(defvar value 12)
⇒ VALUE ; affects ADDER-I and ADD-I-10 but not ADDER-C ar
(funcall add-c-10 32)
⇒ 42 ; as before
(funcall add-i-10 32)
⇒ 44 ; value is now dynamic!
```

Non-conformance. The code shown above has a `SPECIAL` proclamation (by `DEFVAR`) for the variable *`value`* in the execution environment (before the last two `FUNCALL`s) but not in the compilation environment: at the moment the `ADDER-I` function is defined, *`value`* is not known to be a `SPECIAL` variable. Therefore the code is not conforming.

Rationale

The function `ADD-C-10` was compiled **before** *`value`* was declared `SPECIAL`, so the symbol *`value`* was eliminated from its code and the `SPECIAL` declaration did not affect the return value (i.e., `(funcall add-c-10 32)` always returned 42).

On the opposite, function `ADDER-I` was **not** compiled, so `ADD-I-10` was *interpreted*. Whenever `ADD-I-10` is executed, its definition is interpreted all over again. Before `DEFVAR`, *`value`* is evaluated as a lexical (because is is **not** declared `SPECIAL` yet), but after `DEFVAR`, we see a globally `SPECIAL` symbol *`value`* which can have only a global `SYMBOL-VALUE` (not a local binding), and thus we are compelled to evaluate it to 12.

This behavior was implemented intentionally to ease interactive development, because *usually* the `ADDER-I` above would be followed by a (forgotten) `DEFVAR`.

When a user compiles a program, the compiler is allowed to remember the information whether a variable was `SPECIAL` or not, because that allows the compiler to generate more efficient code, but in interpreted code, when the user changes the state of a variable, he does **not** want to re -evaluate all `DEFUN`s that use the variable.

[ANSI CL standard] gives the implementation freedom regarding interpreted evaluation, how much it wants to remember / cache, and how much it wants to evaluate according the current environment, if the environment has changed. **CLISP** implements *ad-hoc look-up* for variables, but not for macros.

3.2.4. Definition of Similarity [CLHS-3.2.4.2.2]

Hash tables are externalizable objects.

3.3. Declarations [CLHS-3.3]

The declarations `(TYPE` ***`type variable`*** `...)`, `(FTYPE` ***`type function`*** `...)`, are ignored by both the interpreter and the compiler.

3.3.1. Declaration `SPECIAL`

Declaration `EXT:NOTSPECIAL`. Declarations `(PROCLAIM '(SPECIAL` ***`variable`***`))` and `DEFCONSTANT` are undone by the `(PROCLAIM '(EXT:NOTSPECIAL` ***`variable`***`))` declaration. This declaration can be used only in global `PROCLAIM` and `DECLAIM` forms, not in local `DECLARE` forms. Of course, you cannot expect miracles: functions compiled before the EXT:NOTSPECIAL proclamation was issued will still be treating *variable* as special even after the EXT:NOTSPECIAL proclamation.

Function EXT:SPECIAL-VARIABLE-P. You can use the function (EXT:SPECIAL-VARIABLE-P *symbol* &OPTIONAL *environment*) to check whether the symbol is a special variable. *environment* of NIL or omitted means use the global environment. You can also obtain the current lexical environment using the macro EXT:THE-ENVIRONMENT (interpreted code only). This function will always return T for global special variables and constant variables.

3.3.2. Declaration SAFETY

Declaration (OPTIMIZE (SAFETY 3)) results in “safe” compiled code: function calls are never eliminated. This guarantees the semantics described in [ANSI CL standard] Section 3.5.

3.3.3. Declaration (COMPILE)

The declaration (COMPILE) has the effect that the current form is compiled prior to execution. Examples:

```
(LOCALLY (DECLARE (compile)) form)
```

executes a compiled version of *form*.

```
(LET ((x 0))
  (FLET ((inc () (DECLARE (compile)) (INCF x))
         (dec () (DECF x)))
    (VALUES #'inc #'dec)))
```

returns two functions. The first is compiled and increments *x*, the second is interpreted (slower) and decrements the same *x*.

The type assertion (THE *value-type form*) enforces a type check in interpreted code. No type check is done in compiled code. See also the EXT:ETHE macro.

3.3.4. Declaration SPACE

The declaration determines what metadata is recorded in the function object:

SPACE >= 2
: documentation string is discarded

SPACE >= 3
: the original lambda list is also discarded (most information is still available, see `DESCRIBE`, but the names of the positional arguments are not).

3.4. Lambda Lists [CLHS-3.4]

3.4.1. Boa Lambda Lists [CLHS-3.4.6]

The initial value of an `&AUX` variable in a boa lambda list is the value of the corresponding slot's initial form.

3.5. The Evaluation and Compilation Dictionary [CLHS-3.8]

3.5.1. Function CONSTANTP

Function `CONSTANTP` fully complies with [ANSI CL standard]. Additionally, some non-trivial forms are identified as constants, e.g., `(CONSTANTP '(+ 1 2 3))` returns `T`.

Warning

Since `DEFCONSTANT` initial value forms are not evaluated at compile time, `CONSTANTP` will not report `T` of their name within the same compilation unit for the null lexical environment. This is consistent and matches questionable code using the pattern `(if (CONSTANTP form) (EVAL form))`. Use `EVAL-WHEN` if you need recognition and the value during compile-time.

3.5.2. Macro `EVAL-WHEN`

`EVAL-WHEN` also accepts the situations `(NOT EVAL)` and `(NOT COMPILE)`.

Warning

The situations `EVAL`, `LOAD` and `COMPILE` are deprecated by the [ANSI CL standard], and they are **not** equivalent to the new standard situations `:EXECUTE`, `:LOAD-TOPLEVEL` and `:COMPILE-TOPLEVEL` in that they ignore the top-level form versus non-top-level form distinction.

Chapter 4. Types and Classes [CLHS-4]

Table of Contents

4.1. Types [CLHS-4.2]

4.1.1. Type Specifiers [CLHS-4.2.3]

The general form of the COMPLEX type specifier is (COMPLEX *type-of-real-part type-of-imaginary-part*). The type specifier (COMPLEX *type*) is equivalent to (COMPLEX *type type*).

DEFTYPE lambda lists are subject to destructuring (nested lambda lists are allowed, as in DEFMACRO) and may contain a &WHOLE marker, but not an &ENVIRONMENT marker.

Function (EXT:TYPE-EXPAND *typespec* &OPTIONAL *once-p*). If *typespec* is a user-defined type, this will expand it recursively until it is no longer a user-defined type (unless *once-p* is supplied and non-NIL). Two values are returned - the expansion and an indicator (T or NIL) of whether the original *typespec* was a user-defined type.

The possible results of TYPE-OF

- CONS
- SYMBOL, NULL, BOOLEAN, KEYWORD
- BIT, (INTEGER 0 #.MOST-POSITIVE-FIXNUM), (INTEGER #.MOST-NEGATIVE-FIXNUM (0)), (INTEGER (#.MOST-POSITIVE-FIXNUM)), (INTEGER * (#.MOST-NEGATIVE-FIXNUM))
- RATIONAL, SHORT-FLOAT, SINGLE-FLOAT, DOUBLE-FLOAT, LONG-FLOAT, COMPLEX
- CHARACTER, BASE-CHAR, STANDARD-CHAR
- (ARRAY *element-type dimensions*), (SIMPLE-ARRAY *element-type dimensions*)
- (VECTOR T *size*), (SIMPLE-VECTOR *size*)
- (STRING *size*), (SIMPLE-STRING *size*)
- (BASE-STRING *size*), (SIMPLE-BASE-STRING *size*)
- (BIT-VECTOR *size*), (SIMPLE-BIT-VECTOR *size*)

- `FUNCTION`, `COMPILED-FUNCTION`, `STANDARD-GENERIC-FUNCTION`
- `STREAM`, `FILE-STREAM`, `SYNONYM-STREAM`, `BROADCAST-STREAM`, `CONCATENATED-STREAM`, `TWO-WAY-STREAM`, `ECHO-STREAM`, `STRING-STREAM`
- `PACKAGE`, `HASH-TABLE`, `READTABLE`, `PATHNAME`, `LOGICAL-PATHNAME`, `RANDOM-STATE`, BYTE
- `SPECIAL-OPERATOR`, `LOAD-TIME-EVAL`, `SYMBOL-MACRO`, `GLOBAL-SYMBOL-MACRO`, `EXT:ENCODING`, `FFI:FOREIGN-POINTER`, `FFI:FOREIGN-ADDRESS`, `FFI:FOREIGN-VARIABLE`, `FFI:FOREIGN-FUNCTION`
- `EXT:WEAK-POINTER`, `EXT:WEAK-LIST`, `EXT:WEAK-AND-RELATION`, `EXT:WEAK-OR-RELATION`, `EXT:WEAK-MAPPING`, `EXT:WEAK-AND-MAPPING`, `EXT:WEAK-OR-MAPPING`, `EXT:WEAK-ALIST`, `READ-LABEL`, `FRAME-POINTER`, `SYSTEM-INTERNAL`
- `ADDRESS` (should not occur)
- any other `SYMBOL` (structure types or **CLOS** classes)
- a class object (**CLOS** classes without a proper name)

4.2. Classes [CLHS-4.3]

The **CLOS** symbols are `EXPORT`ed from the package **“CLOS”**. **“COMMON-LISP”** uses (as in `USE-PACKAGE`) **“CLOS”** and `EXT:RE-EXPORT`s the [ANSI CL standard] standard exported symbols (the **CLISP** extensions, e.g., those described in Chapter 29, *Meta-Object Protocol*, are **not** `EXT:RE-EXPORT`ed). Since the default `:USE` argument to `MAKE-PACKAGE` is **“COMMON-LISP”**, the standard **CLOS** symbols are normally visible in all user-defined packages. If you do not want them (for example, if you want to use the PCL implementation of **CLOS** instead of the native one), do the following:

```
(DEFPACKAGE "CL-NO-CLOS" (:use "CL"))
(DO-EXTERNAL-SYMBOLS (symbol "COMMON-LISP")
  (SHADOW symbol "CL-NO-CLOS"))
(DO-SYMBOLS (symbol "CL-NO-CLOS")
  (EXPORT symbol "CL-NO-CLOS"))
(IN-PACKAGE "CL-NO-CLOS")
(LOAD "pcl") ; or whatever
(DEFPACKAGE "MY-USER" (:use "CL-NO-CLOS"))
(IN-PACKAGE "MY-USER")
;; your code which uses PCL goes here
```

4.3. Deviations from [ANSI CL standard]

DEFCLASS supports the option :METACLASS STRUCTURE-CLASS. This option is necessary in order to define a subclass of a DEFSTRUCT-defined structure type using DEFCLASS instead of DEFSTRUCT.

When CALL-NEXT-METHOD is called with arguments, the rule that the ordered set of applicable methods must be the same as for the original arguments is enforced by the implementation only in interpreted code.

CLOS:GENERIC-FLET and ***CLOS:GENERIC-LABELS*** are implemented as macros, not as special operators (as permitted by Section 3.1.2.1.2.2). They are not imported into the packages **"COMMON-LISP-USER"** and **"COMMON-LISP"** because of the [ANSI CL standard] issue GENERIC-FLET-POORLY-DESIGNED:DELETE.

PRINT-OBJECT is only called on objects of type STANDARD-OBJECT and STRUCTURE-OBJECT. It is not called on other objects, like CONSes and NUMBERs, due to the performance concerns.

4.4. Standard Metaclasses [CLHS-4.3.1.1]

Among those classes listed in Figure 4-8, only the following are instances of BUILT-IN-CLASS:

- T
- CHARACTER
- NUMBER, COMPLEX, REAL, FLOAT, RATIONAL, RATIO, INTEGER
- SEQUENCE
- ARRAY, VECTOR, BIT-VECTOR, STRING
- LIST, CONS
- SYMBOL, NULL
- FUNCTION, GENERIC-FUNCTION, STANDARD-GENERIC-FUNCTION
- HASH-TABLE
- PACKAGE
- PATHNAME, LOGICAL-PATHNAME
- RANDOM-STATE
- READTABLE

- STREAM, BROADCAST-STREAM, CONCATENATED-STREAM, ECHO-STREAM, STRING-STREAM, FILE-STREAM, SYNONYM-STREAM, TWO-WAY-STREAM

4.5. Defining Classes [CLHS-4.3.2]

DEFCLASS supports the :METACLASS option. Possible values are STANDARD-CLASS (the default), STRUCTURE-CLASS (which creates structure classes, like DEFSTRUCT does), and user-defined meta-classes (see Section 29.3.6.7, "Generic Function CLOS:VALIDATE-SUPERCLASS").

It is **not** required that the superclasses of a class are defined before the DEFCLASS form for the class is evaluated. Use Meta-Object Protocol generic functions CLOS:CLASS-FINALIZED-P to check whether the class has been finalized and thus its instances can be created, and CLOS:FINALIZE-INHERITANCE to force class finalization.

See also Section 29.3.1, "Macro DEFCLASS".

4.6. Redefining Classes [CLHS-4.3.6]

Trivial changes, e.g., those that can occur when doubly loading the same code, do not require updating the instances. These are the changes that do not modify the set of local slots accessible in instances, e.g., changes to slot options :INITFORM, :DOCUMENTATION, and changes to class options :DEFAULT-INITARGS, :DOCUMENTATION.

The instances are updated when they are first accessed, **not** at the time when the class is redefined or MAKE-INSTANCES-OBSOLETE is called. When the class has been redefined several times since the instance was last accessed, UPDATE-INSTANCE-FOR-REDEFINED-CLASS is still called just once.

4.7. The Types and Classes Dictionary [CLHS-4.4]

4.7.1. Function COERCE

FIXNUM is not a character designator in [ANSI CL standard], although CODE-CHAR provides an obvious venue to COERCE a FIXNUM to a CHARACTER. When CUSTOM:*COERCE-FIXNUM-CHAR-ANSI* is NIL, **CLISP** COERCEs FIXNUMs to CHARACTERs via CODE-CHAR. When CUSTOM:*COERCE-FIXNUM-CHAR-ANSI* is non-NIL, FIXNUMs cannot be COERCEd to CHARACTERs.

Chapter 5. Data and Control Flow [CLHS-5]

Table of Contents

5.1. The Data and Control Flow Dictionary [CLHS-5.3]

Function `FUNCTION-LAMBDA-EXPRESSION`. The *`name`* of a `FFI:FOREIGN-FUNCTION` is a *string* (the name of the underlying **C** function), not a lisp function name.

Macro `DESTRUCTURING-BIND`. This macro does not perform full error checking.

Macros `PROG1`, `PROG2`, `AND`, `OR`, `PSETQ`, `WHEN`, `UNLESS`, `COND`, `CASE`, `MULTIPLE-VALUE-LIST`, `MULTIPLE-VALUE-BIND`, `MULTIPLE-VALUE-SETQ`. These macros are implemented as special operators (as permitted by Section 3.1.2.1.2.2) and, as such, are rather efficient.

5.1.1. Macro DEFCONSTANT

The initial value is **not** evaluated at compile time, just like with `DEFVAR` and `DEFPARAMETER`. Use `EVAL-WHEN` if you need the value at compile time.

If the variable is already bound to a value which is not `EQL` to the new value, a `WARNING` is issued.

constant variables may not be bound dynamically or lexically.

5.1.2. Macro EXT:FCASE

This macro allows specifying the test for CASE, e.g.,

```
(fcase string= (subseq foo 0 (position #\Space foo))
  ("first" 1)
  (("second" "two") 2)
  (("true" "yes") t)
  (otherwise nil))
```

is the same as

```
(let ((var (subseq foo 0 (position #\Space foo))))
  (cond ((string= var "first") 1)
        ((or (string= var "second") (string= var "two")) :
        ((or (string= var "true") (string= var "yes")) t)
        (t nil)))
```

If you use a built-in HASH-TABLE test (see Section 18.1.3, “Function HASH-TABLE-TEST”) as the test (e.g., EQUAL instead of STRING= above, but not a test defined using EXT:DEFINE-HASH-TABLE-TEST), the compiler will be able to optimize the EXT:FCASE form better than the corresponding COND form.

5.1.3. Function EXT:XOR

This function checks that exactly one of its arguments is non-NIL and, if this is the case, returns its value and index in the argument list as multiple values, otherwise returns NIL.

5.1.4. Function EQ

EQ compares CHARACTERs and FIXNUMs as EQL does. No unnecessary copies are made of CHARACTERs and NUMBERs. Nevertheless, one should use EQL as it is more portable across **Common Lisp** implementations.

`(LET ((x y)) (EQ x x))` always returns T, regardless of *y*.

See also Equality of foreign values..

5.1.5. Function SYMBOL-FUNCTION

(SETF (SYMBOL-FUNCTION *symbol*) *object*) requires *object* to be either a function, a SYMBOL-FUNCTION return value, or a lambda expression. The lambda expression is thereby immediately converted to a FUNCTION.

5.1.6. Macro SETF

Additional places:

FUNCALL
: (SETF (FUNCALL #'*symbol* ...) *object*) and (SETF (FUNCALL '*symbol* ...) *object*) are equivalent to (SETF (*symbol* ...) *object*).

PROGN
: (SETF (PROGN *form* ... *place*) *object*)

LOCALLY
: (SETF (LOCALLY *declaration* ... *form* ... *place*) *object*)

IF
: (SETF (IF *condition* $place_1$ $place_2$) *object*)

GET-DISPATCH-MACRO-CHARACTER
: (SETF (GET-DISPATCH-MACRO-CHARACTER ...) ...) calls SET-DISPATCH-MACRO-CHARACTER.

EXT:LONG-FLOAT-DIGITS:
: (SETF (EXT:LONG-FLOAT-DIGITS) *digits*) sets the default mantissa length of LONG-FLOATs to *digits* bits.

VALUES-LIST
: (SETF (VALUES-LIST *list*) *form*) is equivalent to (VALUES-LIST (SETF *list* (MULTIPLE-VALUE-LIST *form*))).

Note

Note that this place is restricted: it can only be used in SETF, EXT:LETF, EXT:LETF*, not in other positions.

&KEY markers in DEFSETF lambda lists are supported, but the corresponding keywords must appear literally in the program text.

(GET-SETF-EXPANSION *form* &OPTIONAL *environment*), (EXT:GET-SETF-METHOD *form* &OPTIONAL *environment*), and (EXT:GET-SETF-METHOD-MULTIPLE-VALUE *form* &OPTIONAL *environment*) receive as optional argument *environment* the environment necessary for macro expansions. In DEFINE-SETF-EXPANDER and EXT:DEFINE-SETF-METHOD lambda lists, one can specify &ENVIRONMENT and a variable, which will be bound to the environment. This environment should be passed to all calls of GET-SETF-EXPANSION, EXT:GET-SETF-METHOD and EXT:GET-SETF-METHOD-MULTIPLE-VALUE. If this is done, even local macros will be interpreted as places correctly.

An attempt to modify read-only data SIGNALs an ERROR. Program text and quoted constants loaded from files are considered read-only data. This check is only performed for strings, not for conses, other kinds of arrays, and user-defined data types.

See also Section 31.11.2, "Macros EXT:LETF & EXT:LETF*".

5.1.7. Special Operator FUNCTION

(FUNCTION *symbol*) returns the local function definition established by FLET or LABELS, if it exists, otherwise the global function definition.

(SPECIAL-OPERATOR-P *symbol*) returns NIL or T. If it returns T, then (SYMBOL-FUNCTION *symbol*) returns the (useless) special operator handler.

5.1.8. Macro DEFINE-SYMBOL-MACRO

The macro DEFINE-SYMBOL-MACRO establishes SYMBOL-MACROs with global scope (as opposed to SYMBOL-MACROs defined with SYMBOL-MACROLET, which have local scope).

The function ***EXT:SYMBOL-MACRO-EXPAND*** tests for a SYMBOL-MACRO: If *symbol* is defined as a SYMBOL-MACRO in the global environment, (EXT:SYMBOL-MACRO-EXPAND *symbol*) returns two values, T and the expansion; otherwise it returns NIL.

EXT:SYMBOL-MACRO-EXPAND is a special case of MACROEXPAND-1. MACROEXPAND-1 can also test whether a symbol is defined as a SYMBOL-MACRO in lexical environments other than the global environment.

5.1.9. Macro LAMBDA

Constant LAMBDA-LIST-KEYWORDS. (&OPTIONAL &REST &KEY &ALLOW-OTHER-KEYS &AUX &BODY &WHOLE &ENVIRONMENT)

Table 5.1. Function call limits

CALL-ARGUMENTS-LIMIT	2^{12}=4096
MULTIPLE-VALUES-LIMIT	2^{7}=128
LAMBDA-PARAMETERS-LIMIT	2^{12}=4096

5.1.10. Macros DEFUN & DEFMACRO

DEFUN and DEFMACRO are allowed in non-toplevel positions. As an example, consider the old ([CLtL1]) definition of GENSYM:

```
(let ((gensym-prefix "G")
      (gensym-count 1))
  (defun gensym (&optional (x nil s))
    (when s
      (cond ((stringp x) (setq gensym-prefix x))
            ((integerp x)
             (if (minusp x)
               (error "~S: index ~S is negative" 'gensym :
               (setq gensym-count x)))
            (t (error "~S: argument ~S of wrong type" 'ge
    (prog1
      (make-symbol
        (concatenate 'string
          gensym-prefix
```

```
           (write-to-string gensym-count :base 10 :radix n:
       (incf gensym-count))))
```

Function EXT:ARGLIST. Function (EXT:ARGLIST *name*) returns the lambda list of the function or macro that *name* names and SIGNALs an ERROR if *name* is not FBOUNDP. It also SIGNALs an ERROR when the macro lambda list is not available due to the compiler optimization settings (see Section 3.3.4, “Declaration SPACE”).

Variable CUSTOM:*SUPPRESS-CHECK-REDEFINITION*. When *CUSTOM:*SUPPRESS-CHECK-REDEFINITION** is NIL, **CLISP** issues a WARNING when a function (macro, variable, class, etc) is redefined in a different file than its original definition. It is **not** a good idea to set this variable to T.

Variable CUSTOM:*DEFUN-ACCEPT-SPECIALIZED-LAMBDA-LIST*. When *CUSTOM:*DEFUN-ACCEPT-SPECIALIZED-LAMBDA-LIST** is non-NIL, DEFUN accepts specialized lambda lists, converting type-parameter associations to type declarations:

```
(defun f ((x list) (y integer)) ...)
```

is equivalent to

```
(defun f (x y) (declare (type list x) (type integer y)) .
```

This extension is disabled by -ansi and by setting CUSTOM:*ANSI* to T, but can be re-enabled by setting CUSTOM:*DEFUN-ACCEPT-SPECIALIZED-LAMBDA-LIST* explicitly.

Chapter 6. Iteration [CLHS-6]

Table of Contents

6.1. The LOOP Facility [CLHS-6.1]

6.1.1. Iteration variables in the loop epilogue

The standard is unambiguous in that the iteration variables do still exist in the FINALLY clause, but **not** as to what values these variables might have. Therefore the code which relies on the values of such variables, e.g.,

```
(loop for x on y finally (return x))
```

is inherently non-portable across **Common Lisp** implementations, and should be avoided.

6.1.2. Backward Compatibility

There have been some tightening in the LOOP syntax between [CLtL2] and [ANSI CL standard], e.g., the following form is legal in the former but not the latter:

```
(loop initially for i from 1 to 5 do (print i) finally re
```

When ***CUSTOM:*LOOP-ANSI**** is NIL, such forms are still accepted in **CLISP** but elicit a warning at macro-expansion time. When CUSTOM:*LOOP-ANSI* is non-NIL, an ERROR is SIGNALed.

6.2. The Iteration Dictionary [CLHS-6.2]

The macros DOLIST and DOTIMES establish a single binding for the iteration variable and assign it on each iteration.

Chapter 7. Objects [CLHS-7]

Table of Contents

7.1. Standard Method Combination [CLHS-7.6.6.2]

Generic function **`CLOS:NO-PRIMARY-METHOD`** (similar to `NO-APPLICABLE-METHOD`) is called when there is an applicable method but no applicable *primary* method.

The default methods for `CLOS:NO-PRIMARY-METHOD`, `NO-APPLICABLE-METHOD` and `NO-NEXT-METHOD` `SIGNAL` an `ERROR` of type **`CLOS:METHOD-CALL-ERROR`** . You can find out more information about the error using functions `CLOS:METHOD-CALL-ERROR-GENERIC-FUNCTION`, `CLOS:METHOD-CALL-ERROR-ARGUMENT-LIST`, and (only for `NO-NEXT-METHOD`) `CLOS:METHOD-CALL-ERROR-METHOD`. Moreover, when the generic function has only one *dispatching argument*, (i.e., such an argument that not all the corresponding parameter specializers are `T`), an `ERROR` of type **`CLOS:METHOD-CALL-TYPE-ERROR`** is `SIGNAL`ed, additionally making `TYPE-ERROR-DATUM` and `TYPE-ERROR-EXPECTED-TYPE` available.

Chapter 8. Structures [CLHS-8]

Table of Contents

8.1. The options for `DEFSTRUCT`.

8.1.1. The :PRINT-FUNCTION option.

The :PRINT-FUNCTION option should contain a lambda expression (LAMBDA (object stream depth) (declare (ignore depth)) ...) This lambda expression names a FUNCTION whose task is to output the external representation of the STRUCTURE-OBJECT *object* onto the STREAM *stream*. This may be done by outputting text onto the stream using WRITE-CHAR, WRITE-STRING, WRITE, PRIN1, PRINC, PRINT, PPRINT, FORMAT and the like. The following rules must be obeyed:

- The value of *PRINT-ESCAPE* must be respected.
- The treatment of *PRINT-PRETTY* is up to you.
- The value of *PRINT-CIRCLE* need not be respected. This is managed by the system. (But the print-circle mechanism handles only those objects that are direct or indirect components of the structure.)
- The value of *PRINT-LEVEL* is respected by WRITE, PRIN1, PRINC, PRINT, PPRINT, FORMAT instructions ~A, ~S, ~W, and FORMAT instructions ~R, ~D, ~B, ~O, ~X, ~F, ~E, ~G, ~$ with not-numerical arguments. Therefore the print-level mechanism works automatically if only these functions are used for outputting objects and if they are not called on objects with nesting level > 1. (The print-level mechanism does not recognize how many parentheses you have output. It only counts how many times it was called recursively.)
- The value of *PRINT-LENGTH* must be respected, especially if you are outputting an arbitrary number of components.
- The value of *PRINT-READABLY* must be respected. Remember that the values of *PRINT-ESCAPE*, *PRINT-LEVEL*, *PRINT-LENGTH* are ignored if *PRINT-READABLY* is true. The value of *PRINT-READABLY* is respected by PRINT-UNREADABLE-OBJECT, WRITE, PRIN1, PRINC, PRINT, PPRINT, FORMAT instructions ~A, ~S, ~W, and FORMAT instructions ~R, ~D, ~B, ~O, ~X, ~F, ~E, ~G, ~$ with not-numerical arguments. Therefore *PRINT-READABLY* will be respected automatically if only these functions are used for printing objects.
- You need not worry about the values of *PRINT-BASE*, *PRINT-RADIX*, *PRINT-CASE*, *PRINT-GENSYM*, *PRINT-ARRAY*,

CUSTOM:*PRINT-CLOSURE*, CUSTOM:*PRINT-RPARS*, CUSTOM:*PRINT-INDENT-LISTS*.

8.1.2. The :INHERIT option

The :INHERIT option is exactly like :INCLUDE except that it does not create new accessors for the inherited slots (this is a **CLISP** extension).

8.2. The structure Meta-Object Protocol.

The following functions accept a structure *name* as the only argument. If DEFSTRUCT was given the :TYPE option (i.e., DEFSTRUCT did **not** define a new type), then (FIND-CLASS *name*) fails (and the regular **CLOS** Meta-Object Protocol is not applicable), but these functions still work.

EXT:STRUCTURE-SLOTS
: Return the LIST of effective slot definition metaobjects.

EXT:STRUCTURE-DIRECT-SLOTS
: Return the LIST of direct slot definition metaobjects.

EXT:STRUCTURE-KEYWORD-CONSTRUCTOR
: Return the name (a SYMBOL) of the keyword constructor function for the structure, or NIL if the structure has no keyword constructor.

EXT:STRUCTURE-BOA-CONSTRUCTORS
: Return the LIST of names (SYMBOLs) of BOA constructors for the structure.

EXT:STRUCTURE-COPIER
: Return the name (a SYMBOL) of the copier for the structure.

EXT:STRUCTURE-PREDICATE
: Return the name (a SYMBOL) of the predicate for the structure.

Chapter 9. Conditions [CLHS-9]

Table of Contents

When an error occurred, you are in a break loop. You can evaluate forms as usual. The **help** command (or help key if there is one) lists the available debugging commands.

Macro `EXT:MUFFLE-CERRORS`. The macro `(EXT:MUFFLE-CERRORS {form}*)` executes the *`form`*s; when a continuable `ERROR` occurs whose `CONTINUE` `RESTART` can be invoked non-interactively (this includes all continuable `ERROR`s signaled by the function `CERROR`), no message is printed, instead, the `CONTINUE` `RESTART` is invoked.

Macro `EXT:APPEASE-CERRORS`. The macro `(EXT:APPEASE-CERRORS {form}*)` executes the *`form`*s; when a continuable `ERROR` occurs whose `CONTINUE` `RESTART` can be invoked non-interactively (this includes all continuable `ERROR`s `SIGNAL`ed by the function `CERROR`), it is reported as a `WARNING`, and the `CONTINUE` `RESTART` is invoked.

Macro `EXT:ABORT-ON-ERROR`. The macro `(EXT:ABORT-ON-ERROR {form}*)` executes the *`form`*s; when an `ERROR` occurs, or when a **Control**+C interrupt occurs, the error message is printed and the `ABORT` `RESTART` is invoked.

Macro `EXT:EXIT-ON-ERROR`. The macro `(EXT:EXIT-ON-ERROR {form}*)` executes the *`form`*s; when an `ERROR` occurs, or when a **Control**+C interrupt occurs, the error message is printed and **CLISP** terminates with an error status.

Variable `CUSTOM:*REPORT-ERROR-PRINT-BACKTRACE*`. When this variable is non-`NIL` the error message printed by `EXT:ABORT-ON-ERROR` and `EXT:EXIT-ON-ERROR` includes the backtrace (stack).

Function `EXT:SET-GLOBAL-HANDLER`. The function `(EXT:SET-GLOBAL-HANDLER condition handler)` establishes a global handler for the *`condition`*. The *`handler`* should be `FUNCALL`able (a `SYMBOL` or a `FUNCTION`). If it returns, the next applicable handler is invoked, so if you do not want to land in the debugger, it should **not** return. E.g., the option `-on-error abort` and the macro `EXT:ABORT-ON-ERROR` are implemented by installing the following handler:

```
(defun sys::abortonerror (condition)
  (sys::report-error condition)
  (INVOKE-RESTART (FIND-RESTART 'ABORT condition)))
```

When *handler* is NIL, the handler for *condition* is removed and returned. When *condition* is also NIL, all global handlers are removed and returned as a LIST, which can then be passed to EXT:SET-GLOBAL-HANDLER as the first argument and the handlers re-established.

Macro EXT:WITHOUT-GLOBAL-HANDLERS. The macro (EXT:WITHOUT-GLOBAL-HANDLERS &BODY *body*) removes all global handlers, executes *body*, and then restores the handlers.

Macro EXT:WITH-RESTARTS. The macro EXT:WITH-RESTARTS is like RESTART-CASE, except that the forms are specified after the restart clauses instead of before them, and the restarts created are not implicitly associated with any CONDITION. (EXT:WITH-RESTARTS ({*restart-clause*}*) {*form*}*) is therefore equivalent to (RESTART-CASE (PROGN {*form*}*) {*restart-clause*}*).

9.1. Embedded Newlines in Condition Reports [CLHS-9.1.3.1.3]

The error message prefix for the first line is "*** - ". All subsequent lines are indented by 6 characters. Long lines are broken on whitespace (see Section 30.2, "Class EXT:FILL-STREAM").

9.2. The Conditions Dictionary [CLHS-9.2]

Macro RESTART-CASE. In (RESTART-CASE *form* {*restart-clause*}*), the argument list can also be specified after the keyword/value pairs instead of before them, i.e., each *restart-clause* can be either (restart-name EXT:*ARGS* {*keyword-value-pair*}* {*form*}*) or (*restart-name* {*keyword-value-pair*}* EXT:*ARGS* {*form*}*).

Function COMPUTE-RESTARTS. COMPUTE-RESTARTS and FIND-RESTART behave as specified in [ANSI CL standard]: If the optional *condition* argument is non-NIL, only RESTARTs associated with that CONDITION and RESTARTs associated with no CONDITION at all are considered. Therefore the effect of associating a restart to a condition is not to activate it, but to hide it from other conditions. This makes the syntax-dependent implicit association performed by RESTART-CASE nearly obsolete.

Chapter 10. Symbols [CLHS-10]

No notes.

Chapter 11. Packages [CLHS-11]

Table of Contents

The [ANSI CL standard] packages present in CLISP

“COMMON-LISP”
: with the nicknames **“CL”** and **“LISP”**

“COMMON-LISP-USER”
: with the nicknames **“CL-USER”** and **“USER”**

“KEYWORD”
: with no nicknames

11.1. Constraints on the "COMMON-LISP" Package for Conforming Programs - package locking [CLHS-11.1.2.1.2]

Function `EXT:PACKAGE-LOCK`. Packages can be "locked". When a package is locked, attempts to change its symbol table or redefine functions which its symbols name result in a continuable `ERROR` (continuing overrides locking for this operation). When `CUSTOM:*SUPPRESS-CHECK-REDEFINITION*` is `T` (**not** a good idea!), the `ERROR` is not `SIGNAL`ed for redefine operations. Function `(EXT:PACKAGE-LOCK package)` returns the generalized boolean indicating whether the *package* is locked. A package (or a list thereof) can be locked using `(SETF (EXT:PACKAGE-LOCK package-or-list) T)`. **CLISP** locks its system packages (specified in the variable `CUSTOM:*SYSTEM-PACKAGE-LIST*`).

Macro `EXT:WITHOUT-PACKAGE-LOCK`. If you want to evaluate some forms with certain packages unlocked, you can use ***`EXT:WITHOUT-PACKAGE-LOCK`*** :

```
(EXT:WITHOUT-PACKAGE-LOCK ("COMMON-LISP" "EXT" "CLOS")
  (defun restart () ...))
```

or

```
(EXT:WITHOUT-PACKAGE-LOCK ("COMMON-LISP") (trace read-lin
```

`(EXT:WITHOUT-PACKAGE-LOCK () ...)` temporarily unlocks all packages in `CUSTOM:*SYSTEM-PACKAGE-LIST*`.

Variable `CUSTOM:*SYSTEM-PACKAGE-LIST*`. This variable specifies the default packages to be locked by `EXT:SAVEINITMEM` and unlocked by `EXT:WITHOUT-PACKAGE-LOCK` as a list of package names. You may add names to this list, e.g., a module will add its package, but you should **not** remove **CLISP** internal packages from this list.

Discussion - see also the USENET posting by Steven M. Haflich. This should prevent you from accidentally hosing yourself with

```
(DEFSTRUCT instance ...)
```

and allow enforcing modularity. Note that you will also get the continuable `ERROR` when you try to assign (with `SETQ`, `PSETQ`, etc.) a value to an internal special variable living in a locked package and not accessible in your current `*PACKAGE*`, but only in the interpreted code and during compilation. There is no check for package locks in compiled code because of the performance considerations.

11.2. The "COMMON-LISP-USER" Package [CLHS-11.1.2.2]

The **"COMMON-LISP-USER"** package uses the **"COMMON-LISP"** and **"EXT"** packages.

11.3. Implementation-Defined Packages [CLHS-11.1.2.4]

The following additional packages exist:

Implementation-Defined Packages

"CLOS"
: `EXPORT`s all **CLOS**-specific symbols, including some additional symbols.

"SYSTEM"
: has the nicknames **"SYS"** and **"COMPILER"**, and has no `EXPORT`ed symbols. It defines many system internals.

"EXT"
: is the umbrella package for all extensions: it imports and `EXT:RE-EXPORT`s all the external symbols in all **CLISP** extensions, so a simple `(USE-PACKAGE "EXT")` is enough to make all the extensions available in the current package. This package uses packages (in addition to **"COMMON-LISP"**): **"LDAP"**, **"POSIX"**, **"SOCKET"**, **"GSTREAM"**, **"GRAY"**, **"I18N"**, **"CUSTOM"**.

"CHARSET"
: defines and `EXPORT`s some character sets, for use with `EXT:MAKE-ENCODING` and as `:EXTERNAL-FORMAT` argument.

“FFI”
implements the foreign function interface. Some platforms only.

“SCREEN”
defines an API for random screen access. Some platforms only.

“CS-COMMON-LISP”
“CS-COMMON-LISP-USER”
case-sensitive versions of **“COMMON-LISP”** and **“COMMON-LISP-USER”**. See Section 11.4, “Package Case-Sensitivity”.

All pre-existing packages except **“COMMON-LISP-USER”** belong to the implementation, in the sense that the programs that do not follow Section 11.1.2.1.2 ("Constraints on the **“COMMON-LISP”** Package for Conforming Programs") cause undefined behavior.

11.4. Package Case-Sensitivity

CLISP supports programs written with case sensitive symbols. For example, with case sensitive symbols, the symbols `cdr` (the function equivalent to `REST`) and the symbol `CDR` (a user-defined type denoting a Call Data Record) are different and unrelated.

There are some incompatibilities between programs assuming case sensitive symbols and programs assuming the [ANSI CL standard] case insensitive symbols. For example, `(eq 'KB 'Kb)` evaluates to false in a case sensitive world and to true in a case insensitive world. However, unlike some commercial **Common Lisp** implementations, **CLISP** allows both kinds of programs to coexist in the same process and interoperate with each other. Example:

OLD.lisp

```
(IN-PACKAGE "OLD")
(DEFUN FOO () ...)
```

modern.lisp

```
(in-package "NEW")
(defun bar () (old:foo))
(symbol-name 'bar) ; ⇒ "bar"
```

This is achieved through specification of the symbol case policy at the package level. A ***modern package*** is one that is declared to be both case-sensitive and case-inverted and which use the symbols from the **“CS-COMMON-LISP”** package.

A ***case-sensitive package*** is one whose `DEFPACKAGE` declaration (or `MAKE-PACKAGE` creation form) has the option `(:CASE-SENSITIVE T)`. In a case-sensitive package, the reader does **not** uppercase the symbol name before calling `INTERN`. Similarly, the printer, when printing the `SYMBOL-NAME` part of a `SYMBOL` (i.e. the part after the package markers), behaves as if the readtable's case were set to `:PRESERVE`. See also Section 11.5.5, “Function `EXT:PACKAGE-CASE-SENSITIVE-P`”.

A ***case-inverted package*** is one whose `DEFPACKAGE` declaration (or `MAKE-PACKAGE` creation form) has the option `(:CASE-INVERTED T)`. In the context of a case-inverted package, symbol names are case-inverted: upper case characters are mapped to lower case, lower case characters are mapped to upper case, and other characters are left untouched. Every symbol thus conceptually has two symbol names: an old-world symbol name and a modern-world symbol name, which is the case-inverted old-world name. The first symbol name is returned by the function `SYMBOL-NAME`, the modern one by the function `cs-cl:symbol-name`. The internal functions for creating or looking up symbols in a package, which traditionally took a string argument, now conceptually take two string arguments: old-style-string and inverted-string. Actually, a function like `INTERN` takes the old-style-string as argument and computes the inverted-string from it; whereas the function `cs-cl:intern` takes the inverted-string as argument and computes the old-style-string from it. See also Section 11.5.4, “Function `EXT:PACKAGE-CASE-INVERTED-P`”.

For a few built-in functions, a variant for the case-inverted world is defined in the **“CS-COMMON-LISP”** package, which has the nickname **“CS-CL”**:

`cs-cl:symbol-name`
: returns the case-inverted symbol name.

`cs-cl:intern`
`cs-cl:find-symbol`
: work consistently with `cs-cl:symbol-name`.

`cs-cl:shadow`
`cs-cl:find-all-symbols`
`cs-cl:string=`
`cs-cl:string/=`
`cs-cl:string<`
`cs-cl:string>`
`cs-cl:string<=`
`cs-cl:string>=`
`cs-cl:string-trim`
`cs-cl:string-left-trim`
`cs-cl:string-right-trim`
: convert a `SYMBOL` to a `STRING` and therefore exist in a variant that uses `cs-cl:symbol-name` instead of `SYMBOL-NAME`.

`cs-cl:make-package`
: creates a case-inverted `PACKAGE`.

11.4.1. User Package for the Case-sensitive World

A package **"CS-COMMON-LISP-USER"** is provided for the user to modify and work in. It plays the same role as **"COMMON-LISP-USER"**, but for the case-sensitive world.

11.4.2. Package Names

The handling of package names is unchanged. Package names are still usually uppercase. The package names are also subject to `(READTABLE-CASE *READTABLE*)`.

11.4.3. Gensyms and Keywords

Note that gensyms and keywords are still treated traditionally: even in a case-sensitive package, `(EQ #:FooBar #:foobar)` and `(EQ ':KeyWord ':keyword)` evaluate to true. We believe this has limited negative impact for the moment, but can be changed a few years from now.

11.4.4. Migration Tips

The following practices will pose no problems when migrating to a modern case-sensitive world:

- Using [ANSI CL standard] symbols in lowercase.
- Macros that create symbols by suffixing or prefixing given symbols.
- Comparing symbol names as in `(string= (symbol-name x) (symbol-name y))`.

The following practices will not work in a case-sensitive world or can give problems:

- Accessing the same symbol in both upper- and lowercase from the same source file.
- Macros that create symbols in other packages than the original symbols.
- Comparing symbol-name return values with `EQ`.
- Comparing `(SYMBOL-NAME x)` with `(cs-cl:symbol-name y)`.

11.4.5. Using case-sensitive packages by default

CLISP supports a command-line option `-modern` that sets the `*PACKAGE*` initially to the **“CS-COMMON-LISP-USER”** package, and `*PRINT-BASE*` to `:DOWNCASE`.

For packages to be located in the “modern” (case-sensitive) world, you need to augment their `DEFPACKAGE` declaration by adding the option `(:MODERN T)`.

11.5. The Packages Dictionary [CLHS-11.2]

11.5.1. Function MAKE-PACKAGE

The default value of the :USE argument is (**“COMMON-LISP”**).

MAKE-PACKAGE accepts additional keyword arguments :CASE-SENSITIVE and :CASE-INVERTED (but **not** :MODERN!)

11.5.2. Macro DEFPACKAGE

DEFPACKAGE accepts additional options :CASE-SENSITIVE, :CASE-INVERTED, and :MODERN.

When the package being defined already exists, it is modified as follows (and in this order):

:CASE-SENSITIVE
: adjusted with (SETF EXT:PACKAGE-CASE-SENSITIVE-P) (with a warning)

:CASE-INVERTED
: adjusted with (SETF EXT:PACKAGE-CASE-INVERTED-P) (with a warning)

:MODERN
: if **“COMMON-LISP”** is being used, it is un-used and **“CS-COMMON-LISP”** is used instead; also, **“CS-COMMON-LISP”** is used instead of **“COMMON-LISP”** throughout the DEFPACKAGE form, e.g.,

```
(DEFPACKAGE "FOO"
  (:MODERN T)
  (:USE "COMMON-LISP" "EXT"))
```

is equivalent to

```
(DEFPACKAGE "FOO"
  (:CASE-SENSITIVE T)
```

```
    (:CASE-INVERTED T)
    (:USE "CS-COMMON-LISP" "EXT"))
```

(:MODERN NIL) reverts the effects of (:MODERN T).

:NICKNAMES
: adjusted with RENAME-PACKAGE

:DOCUMENTATION
: reset to the new value with (SETF DOCUMENTATION)

:SHADOW
: adjusted with SHADOW

:SHADOWING-IMPORT-FROM
: adjusted with SHADOWING-IMPORT

:USE
: adjusted with USE-PACKAGE and UNUSE-PACKAGE

:IMPORT-FROM
: adjusted with IMPORT

:INTERN
: adjusted with INTERN (but **not** UNINTERN)

:EXPORT
: adjusted with INTERN and EXPORT (but **not** UNEXPORT)

:SIZE
: ignored

11.5.3. Function EXT:RE-EXPORT

The function (EXT:RE-EXPORT *FROM-PACK* *TO-PACK*) re-EXPORTs all external SYMBOLs from *FROM-PACK* also from *TO-PACK*, provided it already uses *FROM-PACK*; and SIGNALs an ERROR otherwise.

11.5.4. Function EXT:PACKAGE-CASE-INVERTED-P

Returns T if the argument is a case-inverted package. This function is SETFable, although it is probably not a good idea to change the case-inverted status of an existing package.

11.5.5. Function EXT:PACKAGE-CASE-SENSITIVE-P

Returns `T` if the argument is a case-sensitive package. This function is `SETF`able, although it is probably not a good idea to change the case-sensitive status of an existing package.

Chapter 12. Numbers [CLHS-12]

Table of Contents

12.1. Numeric Types

The type NUMBER is the disjoint union of the types REAL and COMPLEX ("exhaustive partition")

The type REAL is the disjoint union of the types RATIONAL and FLOAT.

The type RATIONAL is the disjoint union of the types INTEGER and RATIO.

The type INTEGER is the disjoint union of the types FIXNUM and BIGNUM.

The type FLOAT is the disjoint union of the types SHORT-FLOAT, SINGLE-FLOAT, DOUBLE-FLOAT and LONG-FLOAT.

12.2. Number Concepts [CLHS-12.1]

12.2.1. Byte Operations on Integers [CLHS-12.1.1.3.2]

Byte specifiers are objects of built-in type BYTE, not INTEGERs.

12.2.2. Rule of Float Substitutability [CLHS-12.1.3.3]

When a mathematical function may return an exact (RATIONAL) or inexact (FLOAT) result, it always returns the exact result.

12.2.3. Floating-point Computations [CLHS-12.1.4]

There are four floating point types: SHORT-FLOAT, SINGLE-FLOAT, DOUBLE-FLOAT and LONG-FLOAT:

type	sign	mantissa	exponent	comment
SHORT-FLOAT	1 bit	16+1 bits	8 bits	immediate
SINGLE-FLOAT	1 bit	23+1 bits	8 bits	IEEE 754
DOUBLE-FLOAT	1 bit	52+1 bits	11 bits	IEEE 754
LONG-FLOAT	1 bit	>=64 bits	32 bits	variable length

The single and double float formats are those of the IEEE 754 “Standard for Binary Floating-Point Arithmetic”, except that **CLISP** does not support features like `±0`, `±inf`, `NaN`, gradual underflow, etc. **Common Lisp** does not make use of these features, so, to reduce portability problems, **CLISP** by design returns the same floating point results on all platforms (**CLISP** has a floating-point emulation built in for platforms that do not support IEEE 754). Note that

- When you got a `NaN` in your program, your program is broken, so you will spend time determining where the `NaN` came from. It is better to SIGNAL an ERROR in this case.
- When you got unnormalized floats in your program, your results will have a greatly reduced accuracy anyway. Since **CLISP** has the means to cope with this - LONG-FLOATs of variable precision - it does not need unnormalized floats.

This is why *FEATURES* does not contain the :IEEE-FLOATING-POINT keyword.

Arbitrary Precision Floats. LONG-FLOATs have variable mantissa length, which is a multiple of 16 (or 32, depending on the word size of the processor). The default length used when LONG-FLOATs are READ is given by the place (EXT:LONG-FLOAT-DIGITS). It can be set by (SETF (EXT:LONG-FLOAT-DIGITS) *n*), where *n* is a positive INTEGER. E.g., (SETF (EXT:LONG-FLOAT-DIGITS) 3322) sets the default precision of LONG-FLOATs to about 1000 decimal digits.

12.2.3.1. Rule of Float Precision Contagion [CLHS-12.1.4.4]

The floating point contagion is controlled by the variable CUSTOM:*FLOATING-POINT-CONTAGION-ANSI*. When it is non-NIL, contagion is done as per the [ANSI CL standard]: SHORT-FLOAT → SINGLE-FLOAT → DOUBLE-FLOAT → LONG-FLOAT.

Rationale:

See it pragmatically: save what you can and let others worry about the rest.

Brief:

Common Lisp knows the number's precision, not accuracy, so preserving the precision can be accomplished reliably, while anything relating to the accuracy is just a speculation - only the user (programmer) knows what it is in each case.

Detailed:

A computer float is an approximation of a real number. One can think of it as a random variable with the mean equal to itself and standard deviation equal to half the last significant digit. E.g., 1.5 is actually 1.5±0.05. Consider adding 1.5 and 1.75. [ANSI CL standard] requires that (+ 1.5 1.75) return 3.25, while traditional **CLISP** would return 3.3. The implied random variables are: 3.25±0.005 and 3.3±0.05. Note that the traditional **CLISP** way **does** lie about the mean: the mean **is** 3.25 and nothing else, while the standard way **could** be lying about the deviation (accuracy): if the implied accuracy of 1.5 (0.05) is its actual accuracy, then the accuracy of the result cannot be smaller that that. Therefore, since

Common Lisp has no way of knowing the actual accuracy, [ANSI CL standard] (and all the other standard engineering programming languages, like **C**, **Fortran** etc) decides that keeping the accuracy correct is the business of the programmer, while the language should preserve what it can - the precision.

Experience:

Rounding errors accumulate, and if a computation is conducted with insufficient precision, an outright incorrect result can be returned. (E.g., `E(x`^{2}`) - E(x)`^{2} can be negative!) The user should not mix floats of different precision (that's what `CUSTOM:*WARN-ON-FLOATING-POINT-CONTAGION*` is for), but one should not be penalized for this too harshly.

When `CUSTOM:*FLOATING-POINT-CONTAGION-ANSI*` is `NIL`, the traditional **CLISP** method is used, namely the result of an arithmetic operation whose arguments are of different float types is rounded to the float format of the shortest (least precise) of the arguments: `RATIONAL` → `LONG-FLOAT` → `DOUBLE-FLOAT` → `SINGLE-FLOAT` → `SHORT-FLOAT` (in contrast to 12.1.4.4 Rule of Float Precision Contagion!)

Rationale:

See it mathematically. Add intervals: `{1.0 ± 1e-8} + {1.0 ± 1e-16} = {2.0 ± 1e-8}`. So, if we add `1.0s0` and `1.0d0`, we should get `2.0s0`.

Brief:

Do not suggest accuracy of a result by giving it a precision that is greater than its accuracy.

Example:

`(- (+ 1.7 PI) PI)` should not return `1.700000726342836417234L0`, it should return `1.7f0` (or `1.700001f0` if there were rounding errors).

Experience:

If in a computation using thousands of `SHORT-FLOAT`s, a `LONG-FLOAT` (like `PI`) happens to be used, the long precision should not propagate throughout all the intermediate values. Otherwise, the long result would look precise, but its accuracy is only that of a `SHORT-FLOAT`; furthermore much computation time would be lost by calculating with `LONG-FLOAT`s when only `SHORT-FLOAT`s would be needed.

Variable CUSTOM:*WARN-ON-FLOATING-POINT-CONTAGION*

If the variable `CUSTOM:*WARN-ON-FLOATING-POINT-CONTAGION*` is non-`NIL`, a `WARNING` is emitted for every coercion involving different floating-point types. As explained above, float precision contagion is not a good idea. You can avoid the contagion by doing all your computations with the same floating-point type (and using `FLOAT` to convert all constants, e.g., `PI`, to your preferred type).

This variable helps you eliminate all occurrences of float precision contagion: set it to `T` to have **CLISP** `SIGNAL` a `WARNING` on float precision contagion; set it to `ERROR` to have **CLISP** `SIGNAL` an `ERROR` on float precision contagion, so that you can look at the stack backtrace.

12.2.3.2. Rule of Float and Rational Contagion [CLHS-12.1.4.1]

The contagion between floating point and rational numbers is controlled by the variable `CUSTOM:*FLOATING-POINT-RATIONAL-CONTAGION-ANSI*`. When it is non-`NIL`, contagion is done as per the [ANSI CL standard]: `RATIONAL` → `FLOAT`.

When `CUSTOM:*FLOATING-POINT-RATIONAL-CONTAGION-ANSI*` is `NIL`, the traditional **CLISP** method is used, namely if the result is mathematically an exact rational number, this rational number is returned (in contrast to 12.1.4.1 Rule of Float and Rational Contagion!)

`CUSTOM:*FLOATING-POINT-RATIONAL-CONTAGION-ANSI*` has an effect only in those few cases when the mathematical result is exact although one of the arguments is a floating-point number, such as `(* 0 1.618)`, `(/ 0 1.618)`, `(ATAN 0 1.0)`, `(EXPT 2.0 0)`, `(PHASE 2.718)`.

Variable CUSTOM:*WARN-ON-FLOATING-POINT-RATIONAL-CONTAGION*

If the variable `CUSTOM:*WARN-ON-FLOATING-POINT-RATIONAL-CONTAGION*` is non-`NIL`, a `WARNING` is emitted for every avoidable coercion from a rational number to a floating-point number. You can avoid such coercions by calling `FLOAT` to convert the particular rational numbers to your preferred floating-point type.

This variable helps you eliminate all occurrences of avoidable coercions to a floating-point number when a rational number result would be possible: set it to `T` to have **CLISP** `SIGNAL` a `WARNING` in such situations; set it to `ERROR` to have **CLISP** `SIGNAL` an `ERROR` in such situations, so that you can look at the stack backtrace.

Variable CUSTOM:*PHASE-ANSI*

A similar variable, `CUSTOM:*PHASE-ANSI*`, controls the return value of `PHASE` when the argument is an exact nonnegative `REAL`. Namely, if `CUSTOM:*PHASE-ANSI*` is non-`NIL`, it returns a floating-point zero; if `CUSTOM:*PHASE-ANSI*` is `NIL`, it returns an exact zero. Example: `(PHASE 2/3)`

12.2.4. Complex Computations [CLHS-12.1.5]

Complex numbers can have a real part and an imaginary part of different types. For example, `(SQRT -9.0)` evaluates to the number `#C(0 3.0)`, which has a real part of exactly `0`, not only `0.0` (which would mean “approximately 0”).

The type specifier for this is `(COMPLEX INTEGER SINGLE-FLOAT)`, and `(COMPLEX type-of-real-part type-of-imaginary-part)` in general.

The type specifier `(COMPLEX type)` is equivalent to `(COMPLEX type type)`.

12.2.5. Rule of Canonical Representation for Complex Rationals [CLHS-12.1.5.3]

Complex numbers can have a real part and an imaginary part of different types. If the imaginary part is `EQL` to `0`, the number is automatically converted to a real number.

This has the advantage that `(LET ((x (SQRT -9.0))) (* x x))` - instead of evaluating to `#C(-9.0 0.0)`, with *x* = `#C(0.0 3.0)` - evaluates to `#C(-9.0 0)` = `-9.0`, with *x* = `#C(0 3.0)`.

12.3. The Numbers Dictionary [CLHS-12.2]

12.3.1. Random Numbers

To ease reproducibility, the variable `*RANDOM-STATE*` is initialized to the same value on each invocation, so that

```
$ clisp -norc -x '(RANDOM 1s0)'
```

will always print the same number.

If you want a new random state on each invocation, you can arrange for that by using init function:

```
$ clisp -norc -x '(EXT:SAVEINITMEM "foo" :init-function (
$ clisp -norc -M foo.mem -x '(RANDOM 1s0)'
```

or by placing `(SETQ *RANDOM-STATE* (MAKE-RANDOM-STATE T))` into your RC file.

12.3.2. Additional Integer Functions

Function `EXT:!` `(EXT:! n)` returns the factorial of *n*, *n* being a nonnegative `INTEGER`.

Function `EXT:EXQUO`. `(EXT:EXQUO x y)` returns the integer quotient `x/y` of two integers *x*,*y*, and `SIGNAL`s an `ERROR` when the quotient is not integer. (This is more efficient than `/`.)

Function `EXT:XGCD`. `(EXT:XGCD` x_1 `...` x_n`)` returns the values *l*, k_1, ..., k_n, where *l* is the greatest common divisor of the integers x_1, ..., x_n, and k_1, ..., k_n are the integer coefficients such that

```
l = (GCD x1 ... xn)
  = (+ (* k1 x1) ... (* kn xn))
```

Function `EXT:MOD-EXPT`. `(EXT:MOD-EXPT k l m)` is equivalent to `(MOD (EXPT k l) m)` except it is more efficient for very large arguments.

12.3.3. Floating Point Arithmetics

Function `EXPT`. `(EXPT base exponent)` is not very precise if *`exponent`* has a large absolute value.

Function `LOG`. `(LOG number base)` `SIGNAL`s an `ERROR` if `base = 1`.

Constant `PI`. The value of `PI` is a `LONG-FLOAT` with the precision given by `(EXT:LONG-FLOAT-DIGITS)`. When this precision is changed, the value of `PI` is automatically recomputed. Therefore `PI` is **not** a constant variable.

Function `UPGRADED-COMPLEX-PART-TYPE`. When the argument is not a recognizable subtype or `REAL`, `UPGRADED-COMPLEX-PART-TYPE` `SIGNAL`s an `ERROR`, otherwise it returns its argument (even though a

COMPLEX number in **CLISP** can always have REALPART and IMAGPART of any type) because it allows the most precise type inference.

Variable CUSTOM:*DEFAULT-FLOAT-FORMAT*. When rational numbers are to be converted to floats (due to FLOAT, COERCE, SQRT or a transcendental function), the result type is given by the variable CUSTOM:*DEFAULT-FLOAT-FORMAT*.

Macro EXT:WITHOUT-FLOATING-POINT-UNDERFLOW. The macro (EXT:WITHOUT-FLOATING-POINT-UNDERFLOW {*form*}*) executes the *form*s, with errors of type FLOATING-POINT-UNDERFLOW inhibited. Floating point operations will silently return zero instead of SIGNALing an ERROR of type FLOATING-POINT-UNDERFLOW.

Condition FLOATING-POINT-INVALID-OPERATION. This CONDITION is never SIGNALed by **CLISP**.

Condition FLOATING-POINT-INEXACT. This CONDITION is never SIGNALed by **CLISP**.

12.3.4. Float Decoding [CLHS]

FLOAT-RADIX always returns 2.

(FLOAT-DIGITS *number* *digits*) coerces *number* (a REAL) to a floating point number with at least *digits* mantissa digits. The following always evaluates to T:

```
(>= (FLOAT-DIGITS (FLOAT-DIGITS number digits)) digits)
```

12.3.5. Boolean Operations [CLHS]

Table 12.1. Boolean operations

constant	value
BOOLE-CLR	0
BOOLE-SET	15
BOOLE-1	10

constant	value
BOOLE-2	12
BOOLE-C1	5
BOOLE-C2	3
BOOLE-AND	8
BOOLE-IOR	14
BOOLE-XOR	6
BOOLE-EQV	9
BOOLE-NAND	7
BOOLE-NOR	1
BOOLE-ANDC1	4
BOOLE-ANDC2	2
BOOLE-ORC1	13
BOOLE-ORC2	11

12.3.6. Fixnum Limits [CLHS]

Table 12.2. Fixnum limits

CPU type	32-bit CPU	64-bit CPU
MOST-POSITIVE-FIXNUM	2^{24}-1 = 16777215	2^{48}-1 = 281474976710655
MOST-NEGATIVE-FIXNUM	-2^{24} = -16777216	-2^{48} = -281474976710656

12.3.7. Bignum Limits [CLHS]

BIGNUMs are limited in size. Their maximum size is $32*(2^{16}-2)=2097088$ bits. The largest representable BIGNUM is therefore $2^{2097088}-1$.

12.3.8. Float Limits [CLHS]

Together with PI, the other LONG-FLOAT constants

LEAST-NEGATIVE-LONG-FLOAT	LONG-FLOAT-EPSILON
LEAST-NEGATIVE-NORMALIZED-LONG-FLOAT	LONG-FLOAT-NEGATIVE-EPSILON
LEAST-POSITIVE-LONG-FLOAT	MOST-NEGATIVE-LONG-FLOAT
LEAST-POSITIVE-NORMALIZED-LONG-FLOAT	MOST-POSITIVE-LONG-FLOAT

are recomputed whenever (EXT:LONG-FLOAT-DIGITS) is SETFed. They are **not** constant variables.

Chapter 13. Characters [CLHS-13]

Table of Contents

The characters are ordered according to a superset of the ASCII character set.

Platform Dependent: Only in CLISP built with compile-time flag UNICODE

More precisely, **CLISP** uses the 21-bit **UNICODE** 3.2 character set (ISO 10646, also known as UCS-4).

Platform Dependent: UNIX (except NeXTstep), Win32 platforms only, and only in CLISP built without compile-time flag UNICODE.

More precisely, **CLISP** uses the ISO Latin-1 (ISO 8859-1) character set:

	#x0	**#x1**	**#x2**	**#x3**	**#x4**	**#x5**	**#x6**	**#x7**	**#x8**	**#x9**	**#xA**	**#xB**	**#xC**	**#xD**
#x00	**	**	**	**	**	**	**	**	**	**	**	**	**	**
#x10	**	**	**	**	**	**	**	**	**	**	**	**	**	**
#x20		!	"	#	$	%	&	'	(	)	*	+	,	-
#x30	0	1	2	3	4	5	6	7	8	9	:	;	<	=
#x40	@	A	B	C	D	E	F	G	H	I	J	K	L	M
#x50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]
#x60	`	a	b	c	d	e	f	g	h	i	j	k	l	m
#x70	p	q	r	s	t	u	v	w	x	y	z	{	|	}
#x80														
#x90														
#xA0		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	
#xB0	°	±	^{2}	^{3}	´	µ	¶	·	¸	^{1}	º	»	¼	½
#xC0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í
#xD0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý
#xE0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í
#xF0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý

Here ** are control characters, not graphic characters. (The characters left blank here cannot be represented in this character set).

Platform Dependent: NeXTstep platforms only, and only in CLISP built without compile-time flag UNICODE.

More precisely, CLISP uses the NeXTstep character set:

	#x0	#x1	#x2	#x3	#x4	#x5	#x6	#x7	#x8	#x9	#xA	#xB	#xC	#xD
#x00	**	**	**	**	**	**	**	**	**	**	**	**	**	**
#x10	**	**	**	**	**	**	**	**	**	**	**	**	**	**
#x20		!	"	#	$	%	&	'	(	)	*	+	,	-
#x30	0	1	2	3	4	5	6	7	8	9	:	;	<	=
#x40	@	A	B	C	D	E	F	G	H	I	J	K	L	M
#x50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]
#x60	`	a	b	c	d	e	f	g	h	i	j	k	l	m
#x70	p	q	r	s	t	u	v	w	x	y	z	{	|	}
#x80		À	Á	Â	Ã	Ä	Å	Ç	È	É	Ê	Ë	Ì	Í
#x90	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	Ù	Ú	Û	Ü	Ý	Þ	µ
#xA0	©	¡	¢	£	⁄	¥	ƒ	§	¤	’	“	«	‹	›
#xB0	®	–	†	‡	·	¦	¶	•	‚	„	”	»	…	‰
#xC0	¹	`	´	ˆ	˜	¯	˘	˙	¨	²	˚	¸	³	˝
#xD0	—	±	¼	½	¾	à	á	â	ã	ä	å	ç	è	é
#xE0	ì	Æ	í	ª	î	ï	ð	ñ	Ł	Ø	Œ	º	ò	ó
#xF0	ö	æ	ù	ú	û	ı	ü	ý	ł	ø	œ	ß	þ	ÿ

Here ** are control characters, not graphic characters. (The characters left blank here cannot be represented in this character set).

Table 13.1. Standard characters

character	code
#\Space	#x20
#\Newline	#x0A

Table 13.2. Semi-standard characters

character	code
#\Backspace	#x08
#\Tab	#x09
#\Linefeed	#x0A
#\Page	#x0C

character	code
#\Return	#x0D

#\Newline is the line terminator.

Table 13.3. Additional Named Characters

character	code
#\Null	#x00
#\Bell	#x07
#\Escape	#x1B

Table 13.4. Additional syntax for characters with code from #x00 to #x1F:

character	code
#\^@	#x00
#\^A … #\^Z	#x01 … #x1A
#\^[	#x1B
#\^\	#x1C
#\^]	#x1D
#\^^	#x1E
#\^_	#x1F

See also Section 2.6.1, “Sharpsign Backslash [CLHS-2.4.8.1]”.

13.1. Character Scripts [CLHS-13.1.2.1]

The only defined character script is the type `CHARACTER` itself.

13.2. Character Attributes [CLHS-13.1.3]

Characters have no implementation-defined or [CLtL1] font and bit attributes. All characters are simple characters.

13.2.1. Input Characters

For backward compatibility, there is a class `SYS::INPUT-CHARACTER` representing either a character with font and bits, or a keystroke. The following functions work with objects of types `CHARACTER` and `SYS::INPUT-CHARACTER`. Note that `EQL` or `EQUAL` are equivalent to `EQ` on objects of type `SYS::INPUT-CHARACTER`.

`EXT:CHAR-FONT-LIMIT` = **16**

The system uses only font 0.

`EXT:CHAR-BITS-LIMIT` = **16**

Character bits:

key	value
`:CONTROL`	`EXT:CHAR-CONTROL-BIT`
`:META`	`EXT:CHAR-META-BIT`
`:SUPER`	`EXT:CHAR-SUPER-BIT`
`:HYPER`	`EXT:CHAR-HYPER-BIT`

`(EXT:CHAR-FONT` *`object`*`)`

returns the font of a `CHARACTER` or `SYS::INPUT-CHARACTER`.

`(EXT:CHAR-BITS` *`object`*`)`

returns the bits of a `CHARACTER` or `SYS::INPUT-CHARACTER`.

`(EXT:MAKE-CHAR` *`char`* `[`*`bits`* `[`*`font`*`]])`

returns a new `SYS::INPUT-CHARACTER`, or `NIL` if such a character cannot be created.

`(EXT:CHAR-BIT` *`object name`*`)`

returns `T` if the named bit is set in *`object`*, else `NIL`.

`(EXT:SET-CHAR-BIT` *`object name new-value`*`)`

returns a new `SYS::INPUT-CHARACTER` with the named bit set or unset, depending on the `BOOLEAN` *`new-value`*.

Warning

`SYS::INPUT-CHARACTER` is **not** a subtype of `CHARACTER`.

Platform Dependent: UNIX, Win32 platforms only.
The system itself uses this `SYS::INPUT-CHARACTER` type only to mention special keys and **Control**/**Alternate**/**Shift** key status on return from `(READ-CHAR EXT:*KEYBOARD-INPUT*)`.

13.3. Graphic Characters [CLHS-13.1.4.1]

The graphic characters are those **UNICODE** characters which are defined by the **UNICODE** standard, excluding the ranges U0000 … U001F and U007F … U009F.

13.4. Alphabetic Characters [CLHS-13.1.4.2]

The alphabetic characters are those **UNICODE** characters which are defined as letters by the **UNICODE** standard, e.g., the **ASCII** characters

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz

and the international alphabetic characters from the character set:

ÇüéâäàåçêëèïîìÄÅÉæÆôöòûùÿÖÜßáíóúñÑªºãõØøÀÃÕ etc.

13.5. Characters With Case [CLHS-13.1.4.3]

13.5.1. Function EXT:CHAR-INVERTCASE

`(EXT:CHAR-INVERTCASE char)` returns the corresponding character in the other case for `CHAR`, i.e., `CHAR-UPCASE` for a lowercase character and `CHAR-DOWNCASE` for an uppercase character; for a character that does not have a case attribute, the argument is returned. See also `EXT:STRING-INVERTCASE` and `EXT:NSTRING-INVERTCASE`.

13.5.2. Case of Implementation-Defined Characters [CLHS-13.1.4.3.4]

The characters with case are those **UNICODE** characters *c*, for which the upper case mapping *uc* and the lower case mapping *lc* have the following properties:

- *uc* and *lc* are different
- *c* is one of *uc* and *lc*
- the upper case mapping of *uc* and of *lc* is *uc*
- the lower case mapping of *uc* and of *lc* is *lc*

The titlecase property of **UNICODE** characters has no equivalent in **Common Lisp**.

13.6. Numeric Characters [CLHS-13.1.4.4]

The numeric characters are those **UNICODE** characters which are defined as digits by the **UNICODE** standard.

13.7. Ordering of Characters [CLHS-13.1.6]

The characters are ordered according to their **UNICODE** code.

The functions `CHAR-EQUAL` `CHAR-NOT-EQUAL`, `CHAR-LESSP`, `CHAR-GREATERP`, `CHAR-NOT-GREATERP`, `CHAR-NOT-LESSP` ignore bits and font attributes of their arguments.

13.8. Treatment of Newline during Input and Output [CLHS-13.1.8]

Newlines are written according to the stream's `EXT:ENCODING`, see the function `STREAM-EXTERNAL-FORMAT` and the description of `EXT:ENCODING`s, in particular, line terminators. The default behavior is as follows:

Platform Dependent: Win32 platform only.

When writing to a file, #\Newline is converted to CR/LF. (This is the usual convention on **DOS**.) For example, #\Return+#\Newline is written as CR/CR/LF.

When reading from a file, CR/LF is converted to #\Newline (the usual convention on **DOS**), and CR not followed by LF is converted to #\Newline as well (the usual conversion on MacOS, also used by some programs on **Win32**). If you do not want this, i.e., if you really want to distinguish LF, CR and CR/LF, you have to resort to binary input (function `READ-BYTE`).

Justification. Unicode Newline Guidelines say: “Even if you know which characters represents NLF on your particular platform, on input and in interpretation, treat CR, LF, CRLF, and NEL the same. Only on output do you need to distinguish between them.”

Rationale. In **CLISP**, #\Newline is identical to #\Linefeed (which is specifically permitted by [ANSI CL standard] in section 13.1.7 “Character Names”). Consider a file containing exactly this string: `(CONCATENATE 'STRING "foo" (STRING #\Linefeed) "bar" (STRING #\Return) (STRING #\Linefeed))` Suppose we open it with `(OPEN "foo" :EXTERNAL-FORMAT :DOS)`. What should `READ-LINE` return? Right now, it returns "foo" (the second `READ-LINE` returns "bar" and reaches `end-of-stream`). If our i/o were “faithful”, `READ-LINE` would have returned the string `(CONCATENATE 'STRING "foo" (STRING #\Linefeed) "bar")`, i.e., a string with an embedded #\Newline between "foo" and "bar" (because a single #\Linefeed is not a #\Newline in the specified `:EXTERNAL-FORMAT`, it will not make `READ-LINE` return, but it **is** a **CLISP** #\Newline!) Even though the specification for `READ-LINE` does not explicitly forbids newlines inside the returned string, such behavior would be quite surprising, to say the least. Moreover, this line (with an embedded #\Newline) would be written as two lines (when writing to a STREAM with :EXTERNAL-FORMAT of :DOS), because the embedded #\Newline would be written as CR+LF.

13.9. Character Encodings [CLHS-13.1.9]

The integer returned by CHAR-INT is the same as the character's code (CHAR-CODE).

13.10. Documentation of Implementation-Defined Scripts [CLHS-13.1.10]

See Section 31.5, “Encodings”.

13.11. The Characters Dictionary [CLHS-13.2]

13.11.1. Function CHAR-CODE

CHAR-CODE takes values from 0 (inclusive) to CHAR-CODE-LIMIT (exclusive), i.e., the implementation supports exactly CHAR-CODE-LIMIT characters.

Table 13.5. Number of characters

binaries built	without UNICODE support	with UNICODE support
CHAR-CODE-LIMIT	$2^8 = 256$	$17 * 2^{16} = 1114112$

13.11.2. Type BASE-CHAR

The types EXT:STRING-CHAR and BASE-CHAR are equivalent to CHARACTER. EXT:STRING-CHAR used to be available as STRING-CHAR prior to removal from [ANSI CL standard] by CHARACTER-PROPOSAL:2.

13.11.3. Function EXT:CHAR-WIDTH

(EXT:CHAR-WIDTH *char*) returns the number of screen columns occupied by *char*. This is 0 for non-spacing characters (such as control characters and many combining characters), 2 for double-width East Asian characters, and 1 for all other characters. See also function EXT:STRING-WIDTH.

13.12. Platform-Dependent Characters

The characters that are not graphic chars and the space character have names:

Table 13.6. Additional characters (Platform Dependent: Win32 platform only.)

code	char	
(CODE-CHAR #x00)	#\Null	
(CODE-CHAR #x07)	#\Bell	
(CODE-CHAR #x08)	#\Backspace	
(CODE-CHAR #x09)	#\Tab	
(CODE-CHAR #x0A)	#\Newline	#\Linefeed
(CODE-CHAR #x0B)	#\Code11	
(CODE-CHAR #x0C)	#\Page	
(CODE-CHAR #x0D)	#\Return	
(CODE-CHAR #x1A)	#\Code26	
(CODE-CHAR #x1B)	#\Escape	#\Esc
(CODE-CHAR #x20)	#\Space	

code	char	
(CODE-CHAR #x7F)	#\Rubout	

Table 13.7. Additional characters (Platform Dependent: UNIX platform only.)

code	char		
(CODE-CHAR #x00)	#\Null	#\Nul	
(CODE-CHAR #x01)	#\Soh		
(CODE-CHAR #x02)	#\Stx		
(CODE-CHAR #x03)	#\Etx		
(CODE-CHAR #x04)	#\Eot		
(CODE-CHAR #x05)	#\Enq		
(CODE-CHAR #x06)	#\Ack		
(CODE-CHAR #x07)	#\Bell	#\Bel	
(CODE-CHAR #x08)	#\Backspace	#\Bs	
(CODE-CHAR #x09)	#\Tab	#\Ht	
(CODE-CHAR #x0A)	#\Newline	#\Nl	#\Linefeed
(CODE-CHAR #x0B)	#\Vt		
(CODE-CHAR #x0C)	#\Page	#\Np	
(CODE-CHAR #x0D)	#\Return	#\Cr	
(CODE-CHAR #x0E)	#\So		
(CODE-CHAR #x0F)	#\Si		
(CODE-CHAR #x10)	#\Dle		
(CODE-CHAR #x11)	#\Dc1		
(CODE-CHAR #x12)	#\Dc2		
(CODE-CHAR #x13)	#\Dc3		
(CODE-CHAR #x14)	#\Dc4		
(CODE-CHAR #x15)	#\Nak		
(CODE-CHAR #x16)	#\Syn		
(CODE-CHAR #x17)	#\Etb		
(CODE-CHAR #x18)	#\Can		
(CODE-CHAR #x19)	#\Em		

code	char		
(CODE-CHAR #x1A)	#\Sub		
(CODE-CHAR #x1B)	#\Escape	#\Esc	
(CODE-CHAR #x1C)	#\Fs		
(CODE-CHAR #x1D)	#\Gs		
(CODE-CHAR #x1E)	#\Rs		
(CODE-CHAR #x1F)	#\Us		
(CODE-CHAR #x20)	#\Space	#\Sp	
(CODE-CHAR #x7F)	#\Rubout	#\Delete	#\Del

13.13. Obsolete Constants

Table 13.8. Character bit constants (obsolete)

constant	value
EXT:CHAR-CONTROL-BIT	1
EXT:CHAR-META-BIT	2
EXT:CHAR-SUPER-BIT	4
EXT:CHAR-HYPER-BIT	8

Chapter 14. Conses [CLHS-14]

Table of Contents

14.1. The Conses Dictionary [CLHS-14.2]

14.1.1. Mapping Functions

Function EXT:MAPCAP. The function EXT:MAPCAP is like MAPCAN, except that it concatenates the resulting lists with APPEND instead of NCONC:

```
(EXT:MAPCAP function x1 ... xn) ≡
(APPLY #'APPEND (MAPCAR function x1 ... xn))
```

(Actually a bit more efficient that this would have been.)

Function EXT:MAPLAP. The function EXT:MAPLAP is like MAPCON, except that it concatenates the resulting lists with APPEND instead of NCONC:

```
(EXT:MAPLAP function x1 ... xn) ≡
(APPLY #'APPEND (MAPLIST function x1 ... xn))
```

(Actually a bit more efficient that this would have been.)

Chapter 15. Arrays [CLHS-15]

Table of Contents

Function MAKE-ARRAY. MAKE-ARRAY can return specialized arrays for the ARRAY-ELEMENT-TYPEs (UNSIGNED-BYTE 2), (UNSIGNED-BYTE 4), (UNSIGNED-BYTE 8), (UNSIGNED-BYTE 16), (UNSIGNED-BYTE 32), and, of course, the required specializations NIL, BIT and CHARACTER.

15.1. Array Elements [CLHS-15.1.1]

Table 15.1. Array limits

CPU type	32-bit CPU	64-bit CPU
ARRAY-RANK-LIMIT	$2^{12} = 4096$	
ARRAY-DIMENSION-LIMIT	$2^{24}-1 = 16777215$	$2^{32}-1 = 4294967295$

CPU type	32-bit CPU	64-bit CPU
ARRAY-TOTAL-SIZE-LIMIT	2^{24}-1 = 16777215	2^{32}-1 = 4294967295

15.2. The Arrays Dictionary [CLHS-15.2]

Function ADJUST-ARRAY for displaced arrays. An array to which another array is displaced should not be shrunk (using ADJUST-ARRAY) in such a way that the other array points into void space. This cannot be checked at the time ADJUST-ARRAY is called!

Chapter 16. Strings [CLHS-16]

Table of Contents

16.1. The Strings Dictionary [CLHS-16.2]

16.1.1. String Comparison

String comparison (STRING< and friends) is based on the function CHAR<= (see Section 13.7, “Ordering of Characters [CLHS-13.1.6]”). Therefore diphthongs do **not** obey the usual national rules. Example: `o < oe < z < ö`.

16.1.2. Function EXT:STRING-WIDTH

(EXT:STRING-WIDTH *string* &KEY *start* *end*) returns the number of screen columns occupied by *string*. This is computed as the sum of all EXT:CHAR-WIDTHs of all of the *string*'s characters:

```
(REDUCE #'+ string :KEY #'EXT:CHAR-WIDTH)
```

16.1.3. Functions EXT:STRING-INVERTCASE and EXT:NSTRING-INVERTCASE

(EXT:STRING-INVERTCASE *string* &KEY *start* *end*) and (EXT:NSTRING-INVERTCASE *string* &KEY *start* *end*) are similar to STRING-UPCASE et al: they use EXT:CHAR-INVERTCASE to invert the case of each characters in the argument string region.

Chapter 17. Sequences [CLHS-17]

Table of Contents

17.1. The Sequences Dictionary [CLHS-17.3]

17.1.1. Additional Macros

17.1.1.1. Macro EXT:DOSEQ

For iteration through a sequence, a macro EXT:DOSEQ, similar to DOLIST, may be used instead of MAP:

```
(EXT:DOSEQ (variable seqform [resultform])
  {declaration}*
  {tag|form}*)
```

EXT:DOSEQ forms are iteration forms.

17.1.2. Functions NREVERSE & NRECONC

Function NREVERSE. The result of NREVERSE is always EQ to the argument. NREVERSE on a VECTOR swaps pairs of elements. NREVERSE on a LIST swaps the first and the last element and reverses the list chaining between them.

Function NRECONC. The result of NRECONC is EQ to the first argument unless it is NIL, in which case the result is EQ to the second argument.

17.1.3. Functions REMOVE & DELETE

REMOVE, REMOVE-IF, REMOVE-IF-NOT, REMOVE-DUPLICATES return their argument unchanged, if no element has to be removed.

DELETE, DELETE-IF, DELETE-IF-NOT, DELETE-DUPLICATES destructively modify their argument: If the argument is a LIST, the CDR parts are modified. If the argument is a VECTOR with fill pointer, the fill pointer is lowered and the remaining elements are compacted below the new fill pointer.

Variable CUSTOM:*SEQUENCE-COUNT-ANSI*. Contrary to the [ANSI CL standard] issue RANGE-OF-COUNT-KEYWORD:NIL-OR-INTEGER, negative :COUNT keyword arguments are not allowed unless you set CUSTOM:*SEQUENCE-COUNT-ANSI* to a non-NIL value, in which case “using a negative integer value is functionally equivalent to using a value of zero”, as per the [ANSI CL standard] issue.

17.1.4. Functions SORT & STABLE-SORT

SORT and STABLE-SORT accept two additional keyword arguments :START and :END:

```
(SORT sequence predicate &KEY :KEY :START :END)
(STABLE-SORT sequence predicate &KEY :KEY :START :END)
```

SORT and STABLE-SORT are identical. They implement the mergesort algorithm. Worst case complexity: O(n*log(n)) comparisons, where *n* is the LENGTH of the subsequence bounded by the :START and :END arguments.

Chapter 18. Hash Tables [CLHS-18]

Table of Contents

18.1. The Hash Tables Dictionary [CLHS-18.2]

18.1.1. Function MAKE-HASH-TABLE

MAKE-HASH-TABLE accepts two additional keyword arguments :INITIAL-CONTENTS and :WEAK:

```
(MAKE-HASH-TABLE &KEY :TEST :INITIAL-CONTENTS :SIZE
                 :REHASH-SIZE :REHASH-THRESHOLD
                 :WARN-IF-NEEDS-REHASH-AFTER-GC :WEAK)
```

The :TEST argument can be, other than one of the symbols EQ, EQL, EQUAL, EQUALP, one of the symbols EXT:FASTHASH-EQ and EXT:STABLEHASH-EQ. Both of these tests use EQ as the comparison function; they differ in their performance characteristics.

EXT:FASTHASH-EQ
: This uses the fastest possible hash function. Its drawback is that its hash codes become invalid at every garbage-collection (except if all keys are immediate objects), thus requiring a reorganization of the hash table at the first access after each garbage-collection. Especially when generational garbage-collection is used, which leads to frequent small garbage-collections, large hash table with this test can lead to scalability problems.

EXT:STABLEHASH-EQ
: This uses a slower hash function that has the property that its hash codes for instances of the classes SYMBOL, EXT:STANDARD-STABLEHASH (subclass of STANDARD-OBJECT) and EXT:STRUCTURE

-STABLEHASH (subclass of STRUCTURE-OBJECT) are stable across GCs. This test can thus avoid the scalability problems if all keys, other than immediate objects, are SYMBOL, EXT:STANDARD-STABLEHASH or EXT:STRUCTURE-STABLEHASH instances.

One can recommend to use EXT:FASTHASH-EQ for short-lived hash tables. For tables with a longer lifespan which can be big or accessed frequently, it is recommended to use EXT:STABLEHASH-EQ, and to modify the objects that are used as its keys to become instances of EXT:STANDARD-STABLEHASH or EXT:STRUCTURE-STABLEHASH.

When the symbol EQ or the function #'eq is used as a :TEST argument, the value of the variable CUSTOM:*EQ-HASHFUNCTION* is used instead. This value must be one of EXT:FASTHASH-EQ, EXT:STABLEHASH-EQ.

Similarly, the :TEST argument can also be one of the symbols EXT:FASTHASH-EQL, EXT:STABLEHASH-EQL, EXT:FASTHASH-EQUAL, EXT:STABLEHASH-EQUAL. The same remarks apply as for EXT:FASTHASH-EQ and EXT:STABLEHASH-EQ. When the symbol EQL or the function #'eql is used as a :TEST argument, the value of the variable CUSTOM:*EQL-HASHFUNCTION* is used instead; this value must be one of EXT:FASTHASH-EQL, EXT:STABLEHASH-EQL. Similarly, when the symbol EQUAL or the function #'equal is used as a :TEST argument, the value of the variable CUSTOM:*EQUAL-HASHFUNCTION* is used instead; this value must be one of EXT:FASTHASH-EQUAL, EXT:STABLEHASH-EQUAL.

The :WARN-IF-NEEDS-REHASH-AFTER-GC argument, if true, causes a WARNING to be SIGNALed when an object is stored into the table which will force table reorganizations at the first access of the table after each garbage-collection. This keyword argument can be used to check whether EXT:STABLEHASH-EQ should be preferred over EXT:FASTHASH-EQ for a particular table. Use HASH-TABLE-WARN-IF-NEEDS-REHASH-AFTER-GC to check and SETF this parameter after the table has been created.

The :INITIAL-CONTENTS argument is an association list that is used to initialize the new hash table.

The :REHASH-THRESHOLD argument is ignored.

The :WEAK argument can take the following values:

NIL (default)
:KEY
:VALUE
:KEY-AND-VALUE
:KEY-OR-VALUE

and specifies whether the HASH-TABLE is *weak*: if the key, value, either or both are not accessible for the garbage-collection purposes, i.e., if they are only accessible via weak HASH-TABLEs and EXT:WEAK-POINTERs, it is garbage-collected and removed from the weak HASH-TABLE.

The SETFable predicate EXT:HASH-TABLE-WEAK-P checks whether the HASH-TABLE is weak.

Note that the only test that makes sense for weak hash tables are EQ and its variants EXT:FASTHASH-EQ and EXT:STABLEHASH-EQ.

Just like all other weak objects, weak HASH-TABLEs cannot be printed readably.

See also Section 31.7.9, “Weak Hash Tables”.

18.1.1.1. Interaction between HASH-TABLEs and garbage-collection

When a hash table contains keys to be compared by identity - such as NUMBERs in HASH-TABLEs with the HASH-TABLE-TEST EQ; or CONSes in tables which test with EQ or EQL; or VECTORs in tables which test with EQ, EQL or EQUAL; or STANDARD-OBJECT or STRUCTURE-OBJECT instances in tables which test with EQ, EQL, EQUAL or EQUALP; - the hash code will in general depend on the object's address in memory. Therefore it will in general be invalidated after a garbage-collection, and the hash table's internal structure must be recomputed at the next table access.

While :WARN-IF-NEEDS-REHASH-AFTER-GC can help checking the efficiency of a particular HASH-TABLE, the variable ***CUSTOM:*WARN-ON-***

HASHTABLE-NEEDING-REHASH-AFTER-GC* achieves the same effect for all `HASH-TABLE`s in the system at once: when `CUSTOM:*WARN-ON-HASHTABLE-NEEDING-REHASH-AFTER-GC*` is true and a `HASH-TABLE` needs to be rehashed after a garbage-collection, a warning is issued that shows the inefficient `HASH-TABLE`.

What can be done to avoid the inefficiencies detected by these warnings?

1. In many cases you can solve the problem by using the `STABLEHASH` variant of the hash test.
2. In other cases, namely `STANDARD-OBJECT` or `STRUCTURE-OBJECT` instances, you can solve the problem by making the key object classes inherit from `EXT:STANDARD-STABLEHASH` or `EXT:STRUCTURE-STABLEHASH`, respectively.
3. In the remaining cases, you should store a hash key inside the object, of which you can guarantee uniqueness through your application (for example the ID of an object in a database, or the serial number of an object), and use this key as hash key instead of the original object.

18.1.2. Macro EXT:DEFINE-HASH-TABLE-TEST

You can define a new hash table test using the macro `EXT:DEFINE-HASH-TABLE-TEST`: `(EXT:DEFINE-HASH-TABLE-TEST test-name test-function hash-function)`, after which *`test-name`* can be passed as the `:TEST` argument to `MAKE-HASH-TABLE`. E.g.:

```
(EXT:DEFINE-HASH-TABLE-TEST string STRING= SXHASH)
 (MAKE-HASH-TABLE :test 'string)
```

(which is not too useful because it is equivalent to an `EQUAL` `HASH-TABLE` but less efficient).

The fundamental requirement is that the *`test-function`* and *`hash-function`* are consistent:

```
(FUNCALL test-function x y) ⇒
(= (FUNCALL hash-function x) (FUNCALL hash-function y))
```

This means that the following definition:

```
(EXT:DEFINE-HASH-TABLE-TEST number = SXHASH) ; broken!
```

is **not** correct because `(= 1 1d0)` is `T` but `(= (SXHASH 1) (SXHASH 1d0))` is `NIL`. The correct way is, e.g.:

```
(EXT:DEFINE-HASH-TABLE-TEST number = (LAMBDA (x) (SXHASH
```

(note that `(COERCE x SHORT-FLOAT)` does **not** cons up fresh objects while `(COERCE x DOUBLE-FLOAT)` does).

18.1.3. Function HASH-TABLE-TEST

Function `HASH-TABLE-TEST` returns either one of `EXT:FASTHASH-EQ`, `EXT:STABLEHASH-EQ`, `EXT:FASTHASH-EQL`, `EXT:STABLEHASH-EQL`, `EXT:FASTHASH-EQUAL`, `EXT:STABLEHASH-EQUAL`, `EQUALP` (but not `EQ`, `EQL` nor `EQUAL` anymore), or, for `HASH-TABLE`s created with a user-defined `HASH-TABLE-TEST` (see macro `EXT:DEFINE-HASH-TABLE-TEST`), a `CONS` cell `(`*`test-function . hash-function`*`)`.

18.1.4. Macro EXT:DOHASH

For iteration through a `HASH-TABLE`, a macro `EXT:DOHASH`, similar to `DOLIST`, can be used instead of `MAPHASH`:

```
(EXT:DOHASH (key-var value-var hash-table-form [resultfor
  {declaration}*
  {tag|form}*)
```

`EXT:DOHASH` forms are iteration forms.

Chapter 19. Filenames [CLHS-19]

Table of Contents

For most operations, pathnames denoting files and pathnames denoting directories cannot be used interchangeably.

Platform Dependent: UNIX platform only.

For example, `#P"foo/bar"` denotes the file `#P"bar"` in the directory `#P"foo"`, while `#P"foo/bar/"` denotes the subdirectory `#P"bar"` of the directory `#P"foo"`.

Platform Dependent: Win32 platform only.

For example, `#P"foo\\bar"` denotes the file `#P"bar"` in the directory `#P"foo"`, while `#P"foo\\bar\\"` denotes the subdirectory `#P"bar"` of the directory `#P"foo"`.

Platform Dependent: Win32 and Cygwin platforms only.

User variable ***`CUSTOM:*DEVICE-PREFIX*`*** controls translation between **Cygwin** pathnames (e.g., `#P"/cygdrive/c/gnu/clisp/"`) and native **Win32** pathnames (e.g., `#P"C:\\gnu\\clisp\\"`) When it is set to `NIL`, no translations occur and the **Cygwin** port will not understand the native paths and the native **Win32** port will not understand the **Cygwin** paths. When its value is a string, it is used by `PARSE-NAMESTRING` to translate into the appropriate platform-specific representation, so that on **Cygwin**, `(PARSE-NAMESTRING "c:/gnu/clisp/")` returns `#P"/cygdrive/c/gnu/clisp/"`, while on **Win32** `(PARSE-NAMESTRING "/cygdrive/c/gnu/clisp/")` returns `#P"C:/gnu/clisp/"`. The initial value is `"cygdrive"`, you should edit `config.lisp` to change it.

This is especially important for the directory-handling functions.

Table 19.1. The minimum filename syntax that may be used portably

pathname	meaning
"xxx"	for a file with name *xxx*
"xxx.yy"	for a file with name *xxx* and type *yy*
".yy"	for a pathname with type *yy* and no name or with name *.yy* and no type, depending on the value of CUSTOM:*PARSE-NAMESTRING-DOT-FILE*.

Hereby *xxx* denotes 1 to 8 characters, and *yy* denotes 1 to 3 characters, each of which being either an alphanumeric character or the underscore #_. Other properties of pathname syntax vary between operating systems.

19.1. Pathname Components [CLHS-19.2.1]

When a pathname is to be fully specified (no wildcards), that means that no :WILD, :WILD-INFERIORS is allowed, no wildcard characters are allowed in the strings, and *name* EQ NIL may not be allowed either.

19.1.1. Directory canonicalization

As permitted by the MAKE-PATHNAME specification, the PATHNAME directory component is canonicalized when the pathname is constructed:

1. "" and "." are removed
2. "..", "*", and "**" are converted to :UP, :WILD and :WILD-INFERIORS, respectively
3. patterns foo/../ are collapsed

19.1.2. Platform-specific issues

Platform Dependent: UNIX platform only.

Pathname components

host
: always NIL

device
: always NIL

directory = (startpoint . subdirs)

element	values	meaning
startpoint	:RELATIVE | :ABSOLUTE	
subdirs	() | (*subdir* . *subdirs*)	
subdir	:WILD-INFERIORS	** or ..., all subdirectories
subdir	SIMPLE-STRING, may contain wildcard characters "?" and "*" (may also be specified as :WILD)	

name
type
: NIL or SIMPLE-STRING, may contain wildcard characters "?" and "*" (may also be specified as :WILD)

version
: NIL or :WILD or :NEWEST (after merging the defaults)

A **UNIX** filename is split into name and type.

External notation:	"server:sub1.typ/sub2.typ/name.typ"
using defaults:	"sub1.typ/sub2.typ/name.typ"
or	"name.typ"
or	"sub1.typ/**/sub3.typ/x*.lisp"
or similar.	

Platform Dependent: Win32 platform only.

Pathname components

host
: NIL or SIMPLE-STRING, wildcard characters may occur but do not act as wildcards

device
: NIL or :WILD or A|...|Z

directory = (startpoint . subdirs)

element	values	meaning
startpoint	:RELATIVE | :ABSOLUTE	
subdirs	() | (*subdir* . *subdirs*)	
subdir	:WILD-INFERIORS	** or ..., all subdirectories
subdir	SIMPLE-STRING, may contain wildcard characters "?" and "*" (may also be specified as :WILD)	

name
type

NIL or SIMPLE-STRING, may contain wildcard characters "?" and "*" (may also be specified as :WILD)

version

NIL or :WILD or :NEWEST (after merging the defaults)

If *host* is non-NIL, *device* must be NIL.
A **Win32** filename is split into name and type.

External notation:	"A:\sub1.typ\sub2.typ\name.typ"
using defaults:	"\sub1.typ\sub2.typ\name.typ"
or	"name.typ"
or	"*:\sub1.typ**\sub3.typ\x*.lisp"
or similar.	

Instead of "\" one may use "/", as usual for DOS calls.
If *host* is non-NIL and the *directory*'s *startpoint* is not :ABSOLUTE, (PARSE-NAMESTRING (NAMESTRING *pathname*)) will not be the same as *pathname*.

Platform Dependent: UNIX, Win32 platforms only.

The wildcard characters: "*" matches any sequence of characters, "?" matches any one character.

Name/type namestring split.

Platform Dependent: UNIX, Win32 platforms only.

A filename is split into name and type according to the following rule:

- if there is no `"."` in the filename, then the *`name`* is everything, *`type`* is `NIL`;
- if there is a `"."`, then *`name`* is the part before and *`type`* the part after the last dot.
- if the only `"."` is the first character, then the behavior depends on the value of the user variable ***`CUSTOM:*PARSE-NAMESTRING-DOT-FILE*`*** which can be either

`:TYPE`
: `NIL` *`name`*, everything after the `"."` is the *`type`*; or

`:NAME`
: `NIL` *`type`*, everything is the *`name`*

Note

Due to this name/type splitting rule, there are pathnames that cannot result from `PARSE-NAMESTRING`. To get a pathname whose type contains a dot or whose name contains a dot and whose type is `NIL`, `MAKE-PATHNAME` must be used. Example: `(MAKE-PATHNAME :NAME "foo.bar")`.

19.2. :UNSPECIFIC as a Component Value [CLHS-19.2.2.2.3]

The symbol `:UNSPECIFIC` is not permitted as a pathname component for any slot of any pathname. It is also illegal to pass it as an argument to `MAKE-PATHNAME`, although it **is** a legal argument (treated as `NIL`) to `USER-HOMEDIR-PATHNAME`.

The only use for `:UNSPECIFIC` is that it is returned by `PATHNAME-DEVICE` for `LOGICAL-PATHNAME`s, as required by [CLHS-19.3.2.1] Unspecific Components of a Logical Pathname.

19.3. External notation

External notation of pathnames (cf. PARSE-NAMESTRING and NAMESTRING), of course without spaces, [,],{,}:

Platform Dependent: UNIX platform only.

["/"]	"/" denotes absolute pathnames
{ *name* "/" }	each *name* is a subdirectory
[*name* ["." *type*]]	filename with type (extension)

Name and type may be STRINGs of any LENGTH (consisting of printing CHARACTERs, except "/").

Platform Dependent: Win32 platform only.

[[*drivespec*] :]	a letter "*"|a|...|z|A|...|Z
{ *name* [. *type*] \ }	each *name* is a subdirectory, "\" may be replaced by "/"
[*name* [. *type*]]	filename with type (extension)

Name and type may be STRINGs of any LENGTH (consisting of printing CHARACTERs, except "/", "\", ":").

19.4. Logical Pathnames [CLHS-19.3]

No notes.

19.5. The Filenames Dictionary [CLHS-19.4]

Pathname Designators. When CUSTOM:*PARSE-NAMESTRING-ANSI* is NIL, SYMBOL is also treated as a pathname designator, namely its SYMBOL-NAME is converted to the operating system's preferred pathname case.

Function PATHNAME-MATCH-P. PATHNAME-MATCH-P does not interpret missing components as wild.

19.5.1. Function TRANSLATE-PATHNAME

TRANSLATE-PATHNAME accepts three additional keyword arguments: `(TRANSLATE-PATHNAME source from-wildname to-wildname &KEY :ALL :MERGE :ABSOLUTE)`

If `:ALL` is specified and non-NIL, a list of all resulting pathnames, corresponding to all matches of `(PATHNAME-MATCH-P source from-wildname)`, is returned.

If `:MERGE` is specified and NIL, unspecified pieces of *to-pathname* are not replaced by corresponding pieces of *source*.

If `:ABSOLUTE` is specified and non-NIL, the returned pathnames are converted to absolute by merging in the current process' directory, therefore rendering pathnames suitable for the OS and external programs. So, to pass a pathname to an external program, you do `(NAMESTRING (TRANSLATE-PATHNAME pathname #P"" #P"" :ABSOLUTE T))` or `(NAMESTRING (EXT:ABSOLUTE-PATHNAME pathname))`.

19.5.2. Function TRANSLATE-LOGICAL-PATHNAME

TRANSLATE-LOGICAL-PATHNAME accepts an additional keyword argument `:ABSOLUTE`, similar to Section 19.5.1, “Function TRANSLATE-PATHNAME”.

19.5.3. Function PARSE-NAMESTRING

`(PARSE-NAMESTRING string &OPTIONAL host defaults &KEY start end junk-allowed)` returns a logical pathname only if *host* is a logical host or *host* is NIL and *defaults* is a LOGICAL-PATHNAME. To construct a logical pathname from a string, the function LOGICAL-PATHNAME can be used.

The [ANSI CL standard] behavior of recognizing logical pathnames when the *string* begins with some alphanumeric characters followed by a colon (#\:) can be very confusing (cf. `"c:/autoexec.bat"`, `"home:.clisprc"` and `"prep:/pub/gnu"`) and therefore is disabled by default. To enable the [ANSI CL standard] behavior, you should set CUSTOM:*PARSE-NAMESTRING-ANSI* to non-NIL. Note that this also disables treating SYMBOLs as pathname designators.

19.5.4. Function MERGE-PATHNAMES

`(MERGE-PATHNAMES pathname [default-pathname])` returns a logical pathname only if *default-pathname* is a LOGICAL-PATHNAME. To construct a logical pathname from a STRING, the function LOGICAL-PATHNAME can be used.

When both *pathname* and *default-pathname* are relative pathnames, the behavior depends on CUSTOM:*MERGE-PATHNAMES-ANSI*: when it is NIL, then **CLISP** retains its traditional behavior: `(MERGE-PATHNAMES #P"x/" #P"y/")` evaluates to `#P"x/"`

Rationale. MERGE-PATHNAMES is used to specify default components for pathnames, so there is some analogy between `(MERGE-PATHNAMES a b)` and `(OR a b)`. Obviously, putting in the same default a second time should do the same as putting it in once: `(OR a b b)` is the same as `(OR a b)`, so `(MERGE-PATHNAMES (MERGE-PATHNAMES a b) b)` should be the same as `(MERGE-PATHNAMES a b)`.

(This question actually does matter because in **Common Lisp** there is no distinction between “pathnames with defaults merged-in” and “pathnames with defaults not yet applied”.)

Now, `(MERGE-PATHNAMES (MERGE-PATHNAMES #P"x/" #P"y/") #P"y/")` and `(MERGE-PATHNAMES #P"x/" #P"y/")` are `EQUAL` in **CLISP** (when `CUSTOM:*MERGE-PATHNAMES-ANSI*` is `NIL`), but not in implementations that strictly follow the [ANSI CL standard]. In fact, the above *twice-default* = *once-default* rule holds for all pathnames in **CLISP**.

Conversely, when `CUSTOM:*MERGE-PATHNAMES-ANSI*` is non-`NIL`, the normal [ANSI CL standard] behavior is exhibited: `(MERGE-PATHNAMES #P"x/" #P"y/")` evaluates to `#P"y/x/"`.

Rationale. "merge" is *merge* and not *or*.

19.5.5. Function `LOAD-LOGICAL-PATHNAME-TRANSLATIONS`

When the *`host`* argument to `LOAD-LOGICAL-PATHNAME-TRANSLATIONS` is not a defined logical host yet, we proceed as follows:

1. If both environment variables `LOGICAL_HOST_`*`host`*`_FROM` and `LOGICAL_HOST_`*`host`*`_TO` exist, then their values define the map of the *`host`*.
2. If the environment variable `LOGICAL_HOST_`*`host`* exists, its value is read from, and the result is passed to `(SETF LOGICAL-PATHNAME-TRANSLATIONS)`.
3. Variable `CUSTOM:*LOAD-LOGICAL-PATHNAME-TRANSLATIONS-DATABASE*` is consulted. Its value should be a list of files and/or directories, which are searched for in the `CUSTOM:*LOAD-PATHS*`, just like for `LOAD`. When the element is a file, it is `READ` from, Allegro CL-style, odd objects being host names and even object being their `LOGICAL-PATHNAME-TRANSLATIONS`. When the element is a directory, a file, named *`host`* or *`host`*`.host`, in that directory, is `READ` from once, CMUCL-style, the object read being the `LOGICAL-PATHNAME-TRANSLATIONS` of the *`host`*.

19.5.6. Function EXT:ABSOLUTE-PATHNAME

(EXT:ABSOLUTE-PATHNAME *pathname*) converts the *pathname* to a physical pathname, then - if its directory component is not absolute - converts it to an absolute pathname, by merging in the current process' directory. This is like TRUENAME, except that it does not verify that a file named by the *pathname* exists, not even that its directory exists. It does no filesystem accesses, except to determine the current directory. This function is useful when you want to save a pathname over time, or pass a pathname to an external program.

Chapter 20. Files [CLHS-20]

Table of Contents

20.1. The Files Dictionary [CLHS-20.2]

Directory is not a file

CLISP has traditionally taken the view that a directory is a separate object and not a special kind of file, so whenever the standard says that a function operates on *files* without specifically mentioning that it also works on *directories*, **CLISP** SIGNALs an ERROR when passed a directory.

CLISP provides separate directory functions, such as EXT:DELETE-DIR, EXT:RENAME-DIR et al.

Function PROBE-FILE

PROBE-FILE cannot be used to check whether a directory exists. Use functions EXT:PROBE-DIRECTORY or DIRECTORY for this.

Function FILE-AUTHOR

FILE-AUTHOR always returns NIL, because the operating systems CLISP is ported to do not store a file's author in the file system. Some operating systems, such as UNIX, have the notion of a file's *owner*, and some other Common Lisp implementations return the user name of the file owner. CLISP does not do this, because *owner* and *author* are not the same; in particular, authorship is preserved by copying, while ownership is not.

Use OS:FILE-OWNER to find the owner of the file. See also OS:FILE-PROPERTIES (Platform Dependent: Win32 platform only.).

Function EXT:PROBE-DIRECTORY

(EXT:PROBE-DIRECTORY *pathname*) tests whether *pathname* exists and is a directory. It will, unlike PROBE-FILE or TRUENAME, not SIGNAL an ERROR if the parent directory of *pathname* does not exist.

Function DELETE-FILE

(DELETE-FILE *pathname*) deletes the pathname *pathname*, not its TRUENAME, and returns the absolute pathname it actually removed or NIL if *pathname* did not exist. When *pathname* points to a file which is currently open in CLISP, an ERROR is SIGNALed. To remove a directory, use EXT:DELETE-DIR instead.

Function RENAME-FILE

This function cannot operate on directories, use EXT:RENAME-DIR to rename a directory.

Function DIRECTORY

(DIRECTORY &OPTIONAL *pathname* &KEY :FULL :CIRCLE :IF-DOES-NOT-EXIST) can run in two modes:

- If *`pathname`* contains no name or type component, a list of all matching directories is produced. E.g., `(DIRECTORY "/etc/*/")` lists all subdirectories in the directory `#P"/etc/"`.
- Otherwise a list of all matching files is returned. E.g., `(DIRECTORY "/etc/*")` lists all regular files in the directory `#P"/etc/"`. If the `:FULL` argument is non-`NIL`, additional information is returned: for each matching file you get a `LIST` of at least four elements (*`file-pathname file-truename file-write-date-as-decoded-time file-length`*).

If you want **all** the files **and** subdirectories in the current directory, you should use `(NCONC (DIRECTORY "*/") (DIRECTORY "*"))`. If you want all the files and subdirectories in all the subdirectories under the current directory (similar to the **ls -R** **UNIX** command), use `(NCONC (DIRECTORY "**/") (DIRECTORY "**/*"))`.

Platform Dependent: UNIX platform only.

If the `:CIRCLE` argument is non-`NIL`, the function avoids endless loops that may result from symbolic links.

The argument `:IF-DOES-NOT-EXIST` controls the treatment of links pointing to non-existent files and can take the following values:

`:DISCARD` (default)
: discard the bad directory entries

`:ERROR`
: an `ERROR` is `SIGNAL`ed on bad directory entries (this corresponds to the default behavior of `DIRECTORY` in **CMU CL**)

`:KEEP`
: keep bad directory entries in the returned list (this roughly corresponds to the `(DIRECTORY ... :TRUNAMEP NIL)` call in **CMU CL**)

`:IGNORE`
: Similar to `:DISCARD`, but also do not signal an error when a directory is unaccessible (contrary to the [ANSI CL standard] specification).

Function `EXT:DIR`

`(EXT:DIR &OPTIONAL pathname)` is like `DIRECTORY`, but displays the pathnames instead of returning them. `(EXT:DIR)` shows the contents of the current directory.

Function `EXT:CD`

`(EXT:CD pathname)` sets the current working directory, `(EXT:CD)` returns it.

Platform Dependent: UNIX platform only.

`(EXT:CD [pathname])` manages the current directory.

Platform Dependent: Win32 platform only.

`(EXT:CD [pathname])` manages the current device and the current directory.

Function `EXT:DEFAULT-DIRECTORY`

`(EXT:DEFAULT-DIRECTORY)` is equivalent to `(EXT:CD)`. `(SETF (EXT:DEFAULT-DIRECTORY) pathname)` is equivalent to `(EXT:CD pathname)`, except for the return value.

Function `EXT:MAKE-DIR`

`(EXT:MAKE-DIR directory-pathname)` creates a new subdirectory.

Function `EXT:DELETE-DIR`

`(EXT:DELETE-DIR directory-pathname)` removes an (empty) subdirectory.

Function EXT:RENAME-DIR

(EXT:RENAME-DIR *old-directory-pathname new-directory-pathname*) renames a subdirectory to a new name.

Chapter 21. Streams [CLHS-21]

Table of Contents

21.1. Interactive Streams [CLHS-21.1.1.1.3]

Interactive streams are those whose next input might depend on a prompt one might output.

21.2. Terminal interaction

See also Section 32.1, “Random Screen Access”.

21.2.1. Command line editing with GNU readline

Platform Dependent: Only in CLISP linked against the GNU readline library.

Input through `*TERMINAL-IO*` uses the GNU readline library. Arrow keys can be used to move within the input history. The TAB key completes the `SYMBOL` name or `PATHNAME` that is being typed. See readline user manual for general details and TAB key for **CLISP**-specific extensions.

Warning

The GNU readline library is **not** used (even when **CLISP** is linked against it) if the `stdin` and `stdout` do not both refer to the same terminal. This is determined by the function `stdio_same_tty_p` in file `src/stream.d`. In some exotic cases, e.g., when running under **gdb** in an **rxvt** window under **Cygwin**, this may be determined incorrectly.

See also Section 33.4, “Advanced Readline and History Functionality”.

Linking against GNU readline. For **CLISP** to use GNU readline it has to be detected by the `configure` process.

- If you run it as **./configure --with-readline**, it will fail if it cannot find a modern working GNU readline installation.
- If you use the option `--without-readline`, it will not even try to find GNU readline.
- The default behavior (`--with-readline=default`) is to use GNU readline if it is found and link **CLISP** without it otherwise.

You can find out whether GNU readline has been detected by running

```
$ grep HAVE_READLINE config.h
```

in your build directory.

21.2.2. Macro EXT:WITH-KEYBOARD

Platform Dependent: UNIX, Win32 platforms only.

`*TERMINAL-IO*` is not the only stream that communicates directly with the user: During execution of the body of a `(EXT:WITH-KEYBOARD . *body*)` form, ***`EXT:*KEYBOARD-INPUT*`*** is the `STREAM` that reads the keystrokes from the keyboard. It returns every keystroke in detail as an `SYS::INPUT-CHARACTER` with the following slots (see Section 13.2.1, “Input Characters” for accessing them):

char
: the `CHARACTER` for standard keys (accessed with `CHARACTER`)

Note

For non-standard keys `CHARACTER` `SIGNAL`s an `ERROR`, use `EXT:CHAR-KEY`:

```
(EXT:WITH-KEYBOARD
 (LOOP :for char = (READ-CHAR EXT:*KEYBOARD-INPUT*
   :for key = (OR (EXT:CHAR-KEY char) (CHARACTER c
   :do (PRINT (LIST char key))
   :when (EQL key #\Space) :return (LIST char key)
```

`key`

the key name, for non-standard keys (accessed with `EXT:CHAR-KEY`):

Platform Dependent: UNIX, Win32 platforms only.

key	value
F1..F12	`:F1..:F12`
Insert	`:INSERT`
Delete	`:DELETE`
Home	`:HOME`
End	`:END`
Center	`:CENTER`
PgUp	`:PGUP`
PgDn	`:PGDN`
Arrow keys	`:LEFT :RIGHT :UP :DOWN`

`bits`

`:HYPER`
(Platform Dependent: **Win32** platform only.) if a non-standard key. These keys are: *[Win32]:* Function keys, cursor keypads, numeric keypad.

`:SUPER`
(Platform Dependent: **Win32** platform only.) if pressed together with **Shift** key(s) and if the keystroke would have been different without **Shift**.

`:CONTROL`
if pressed together with the **Control** key.

`:META`
(Platform Dependent: **Win32** platform only.) if pressed together with the **Alternate** key.

`font`

Always `0`.

This keyboard input is not echoed on the screen. During execution of a (EXT:WITH-KEYBOARD . *body*) form, no input from *TERMINAL-IO* or any synonymous stream should be requested.

Warning

Since SYS::INPUT-CHARACTER is **not** a subtype of CHARACTER, READ-LINE on EXT:*KEYBOARD-INPUT* is illegal.

21.3. The Streams Dictionary [CLHS-21.2]

21.3.1. Function STREAM-ELEMENT-TYPE

STREAM-ELEMENT-TYPE is SETFable. The STREAM-ELEMENT-TYPE of STREAMs created by the functions OPEN, EXT:MAKE-PIPE-INPUT-STREAM EXT:MAKE-PIPE-OUTPUT-STREAM, EXT:MAKE-PIPE-IO-STREAM, SOCKET:SOCKET-ACCEPT, SOCKET:SOCKET-CONNECT can be modified, if the old and the new STREAM-ELEMENT-TYPEs are either

- both equivalent to CHARACTER or (UNSIGNED-BYTE 8) or (SIGNED-BYTE 8); or
- both equivalent to (UNSIGNED-BYTE *n*) or (SIGNED-BYTE *n*), with the same *n*.

Functions STREAM-ELEMENT-TYPE and (SETF STREAM-ELEMENT-TYPE) are GENERIC-FUNCTIONs, see Chapter 30, *Gray streams*.

21.3.1.1. Binary input from *STANDARD-INPUT*

Note that you cannot change STREAM-ELEMENT-TYPE for some built-in streams, such as terminal streams, which is normally the value of *TERMINAL-IO*. Since *STANDARD-INPUT* normally is a SYNONYM-STREAM pointing to *TERMINAL-IO*, you cannot use READ-BYTE on it.

Since CGI (Common Gateway Interface) provides the form data for **METHOD="POST"** on the stdin, and the server will **not** send you an end-of-stream on the end of the data, you will need to use (EXT:GETENV "CONTENT_LENGTH") to determine how much data you should read from stdin. **CLISP** will detect that stdin is not a terminal and create a regular FILE-STREAM which can be passed to (SETF STREAM-ELEMENT-TYPE). To test this functionality interactively, you will need to open the standard input in the binary mode:

```
(let ((buf (MAKE-ARRAY (PARSE-INTEGER (EXT:GETENV "CONTEN'
                       :element-type '(UNSIGNED-BYTE 8)))
  (WITH-OPEN-STREAM (in (EXT:MAKE-STREAM :INPUT :ELEMENT-'
    (READ-SEQUENCE buf in))
  buf)
```

21.3.2. Function EXT:MAKE-STREAM

Function `EXT:MAKE-STREAM` creates a Lisp stream out of an OS file descriptor: `(EXT:MAKE-STREAM object &KEY :DIRECTION :ELEMENT-TYPE :EXTERNAL-FORMAT :BUFFERED)`

object designates an OS handle (a file descriptor), and should be one of the following:

number
: denotes the file descriptor of this value

:INPUT
: denotes **CLISP** process `*STANDARD-INPUT*`

:OUTPUT
: denotes **CLISP** process `*STANDARD-OUTPUT*`

:ERROR
: denotes **CLISP** process `*ERROR-OUTPUT*`

STREAM
: denotes the handle of this stream, which should be a `FILE-STREAM` or a `SOCKET:SOCKET-STREAM`

Beware of buffering!

When there are several Lisp `STREAM`s backed by the same OS file descriptor, the behavior may be highly confusing when some of the Lisp streams are `:BUFFERED`. Use `FORCE-OUTPUT` for output `STREAM`s, and bulk input for input `STREAM`s.

The handle is duplicated (with `dup`), so it is safe to `CLOSE` a `STREAM` returned by `EXT:MAKE-STREAM`.

21.3.3. Binary input, READ-BYTE, EXT:READ-INTEGER & EXT:READ-FLOAT

The function `(EXT:READ-INTEGER stream element-type &OPTIONAL ENDIANNESS eof-error-p eof-value)` reads a multibyte `INTEGER` from *`stream`*, which should be a `STREAM` with `STREAM-ELEMENT-TYPE` `(UNSIGNED-BYTE 8)`. *`element-type`* should be type equivalent to `(UNSIGNED-BYTE n)`, where *n* is a multiple of 8.

`(EXT:READ-INTEGER stream element-type)` is like `(READ-BYTE stream)` if *`stream`*'s `STREAM-ELEMENT-TYPE` were set to *`element-type`*, except that *`stream`*'s `FILE-POSITION` will increase by $^{n}/_{8}$ instead of 1.

Together with `(SETF STREAM-ELEMENT-TYPE)`, this function permits mixed character/binary input from a stream.

The function `(EXT:READ-FLOAT stream element-type &OPTIONAL ENDIANNESS eof-error-p eof-value)` reads a floating-point number in IEEE 754 binary representation from *`stream`*, which should be a `STREAM` with `STREAM-ELEMENT-TYPE` `(UNSIGNED-BYTE 8)`. *`element-type`* should be type equivalent to `SINGLE-FLOAT` or `DOUBLE-FLOAT`.

Endianness. ***`ENDIANNESS`*** can be `:LITTLE` or `:BIG`. The default is `:LITTLE`, which corresponds to the `READ-BYTE` behavior in **CLISP**.

21.3.4. Binary output, `WRITE-BYTE`, `EXT:WRITE-INTEGER` & `EXT:WRITE-FLOAT`

The function `(EXT:WRITE-INTEGER integer stream element-type &OPTIONAL ENDIANNESS)` writes a multi-byte `INTEGER` to *`stream`*, which should be a `STREAM` with `STREAM-ELEMENT-TYPE` `(UNSIGNED-BYTE 8)`. *`element-type`* should be type equivalent to `(UNSIGNED-BYTE n)`, where *n* is a multiple of 8.

`(EXT:WRITE-INTEGER integer stream element-type)` is like `(WRITE-BYTE integer stream)` if *`stream`*'s `STREAM-ELEMENT-TYPE` were set to *`element-type`*, except that *`stream`*'s `FILE-POSITION` will increase by $^{n}/_{8}$ instead of 1.

Together with `(SETF STREAM-ELEMENT-TYPE)`, this function permits mixed character/binary output to a `STREAM`.

The function `(EXT:WRITE-FLOAT` ***`float stream element-type`*** `&OPTIONAL` ***`ENDIANNESS`***`)` writes a floating-point number in IEEE 754 binary representation to *`stream`*, which should be a `STREAM` with `STREAM-ELEMENT-TYPE (UNSIGNED-BYTE 8)`. *`element-type`* should be type equivalent to `SINGLE-FLOAT` or `DOUBLE-FLOAT`.

21.3.5. Bulk Input and Output

Function `READ-SEQUENCE`. In addition to `READ-SEQUENCE`, the following two functions are provided:

`EXT:READ-BYTE-SEQUENCE` performs multiple `READ-BYTE` operations:
: `(EXT:READ-BYTE-SEQUENCE` ***`sequence stream`*** `&KEY :START :END :NO-HANG :INTERACTIVE)` fills the subsequence of *`sequence`* specified by `:START` and `:END` with `INTEGER`s consecutively read from *`stream`*. It returns the index of the first element of *`sequence`* that was not updated (= *end* or < *end* if the *`stream`* reached its end). When *`no-hang`* is non-`NIL`, it does not block: it treats input unavailability as `end-of-stream`. When *`no-hang`* is `NIL` and *`interactive`* is non-`NIL`, it can block for reading the first byte but does not block for any further bytes.
: This function is especially efficient if *`sequence`* is a `(VECTOR (UNSIGNED-BYTE 8))` and *`stream`* is a file/pipe/socket `STREAM` with `STREAM-ELEMENT-TYPE (UNSIGNED-BYTE 8)`.

`EXT:READ-CHAR-SEQUENCE` performs multiple `READ-CHAR` operations:
: `(EXT:READ-CHAR-SEQUENCE` ***`sequence stream`*** `&KEY :START :END)` fills the subsequence of *`sequence`* specified by `:START` and `:END` with characters consecutively read from *`stream`*. It returns the index of the first element of *`sequence`* that was not updated (= *end* or < *end* if the *`stream`* reached its end).
: This function is especially efficient if *`sequence`* is a `STRING` and *`stream`* is a file/pipe/socket `STREAM` with `STREAM-ELEMENT-TYPE CHARACTER` or an input `STRING-STREAM`.

Function `WRITE-SEQUENCE`. In addition to `WRITE-SEQUENCE`, the following two functions are provided:

`EXT:WRITE-BYTE-SEQUENCE` performs multiple `WRITE-BYTE` operations:

`(EXT:WRITE-BYTE-SEQUENCE` *`sequence stream`* `&KEY :START :END :NO-HANG :INTERACTIVE)` outputs the `INTEGER`s of the subsequence of *`sequence`* specified by `:START` and `:END` to *`stream`*. When *`no-hang`* is non-`NIL`, does not block. When *`no-hang`* is `NIL` and *`interactive`* is non-`NIL`, it can block for writing the first byte but does not block for any further bytes. Returns two values: *`sequence`* and the index of the first byte that was not output.
This function is especially efficient if *`sequence`* is a `(VECTOR (UNSIGNED-BYTE 8))` and *`stream`* is a file/pipe/socket `STREAM` with `STREAM-ELEMENT-TYPE (UNSIGNED-BYTE 8)`.

`EXT:WRITE-CHAR-SEQUENCE` performs multiple `WRITE-CHAR` operations:

`(EXT:WRITE-CHAR-SEQUENCE` *`sequence stream`* `&KEY :START :END)` outputs the characters of the subsequence of *`sequence`* specified by `:START` and `:END` to *`stream`*. Returns the *`sequence`* argument.
This function is especially efficient if *`sequence`* is a `STRING` and *`stream`* is a file/pipe/socket `STREAM` with `STREAM-ELEMENT-TYPE CHARACTER`.

Rationale. The rationale for `EXT:READ-CHAR-SEQUENCE`, `EXT:READ-BYTE-SEQUENCE`, `EXT:WRITE-CHAR-SEQUENCE` and `EXT:WRITE-BYTE-SEQUENCE` is that some `STREAM`s support both character and binary i/o, and when you read into a `SEQUENCE` that can hold both (e.g., `LIST` or `SIMPLE-VECTOR`) you cannot determine which kind of input to use. In such situation `READ-SEQUENCE` and `WRITE-SEQUENCE` `SIGNAL` an `ERROR`, while `EXT:READ-CHAR-SEQUENCE`, `EXT:READ-BYTE-SEQUENCE`, `EXT:WRITE-CHAR-SEQUENCE` and `EXT:WRITE-BYTE-SEQUENCE` work just fine.

21.3.6. Non-Blocking Input and Output

In addition to the standard functions `LISTEN` and `READ-CHAR-NO-HANG`, **CLISP** provides the following functionality facilitating non-blocking input and output, both binary and character.

`(EXT:READ-CHAR-WILL-HANG-P` *`stream`*`)`

`EXT:READ-CHAR-WILL-HANG-P` queries the stream's input status. It returns `NIL` if `READ-CHAR` and `PEEK-CHAR` with a *`peek-type`* of `NIL` will return immediately. Otherwise it returns `T`. (In the latter case the standard `LISTEN` function would return `NIL`.)

Note the difference with `(NOT (LISTEN stream))`: When the `end-of-stream` is reached, `LISTEN` returns `NIL`, whereas `EXT:READ-CHAR-WILL-HANG-P` returns `NIL`.

Note also that `EXT:READ-CHAR-WILL-HANG-P` is not a good way to test for `end-of-stream`: If `EXT:READ-CHAR-WILL-HANG-P` returns `T`, this does not mean that the *`stream`* will deliver more characters. It only means that it is not known at this moment whether the *`stream`* is already at `end-of-stream`, or will deliver more characters.

(`EXT:READ-BYTE-LOOKAHEAD` *stream*)

To be called only if *`stream`*'s `STREAM-ELEMENT-TYPE` is `(UNSIGNED-BYTE 8)` or `(SIGNED-BYTE 8)`. Returns `T` if `READ-BYTE` would return immediately with an `INTEGER` result.

Returns `:EOF` if the `end-of-stream` is already known to be reached. If `READ-BYTE`'s value is not available immediately, returns `NIL` instead of waiting.

(`EXT:READ-BYTE-WILL-HANG-P` *stream*)

To be called only if *`stream`*'s `STREAM-ELEMENT-TYPE` is `(UNSIGNED-BYTE 8)` or `(SIGNED-BYTE 8)`. Returns `NIL` if `READ-BYTE` will return immediately. Otherwise it returns true.

(`EXT:READ-BYTE-NO-HANG` *stream* `&OPTIONAL` *eof-error-p eof-value*)

To be called only if *`stream`*'s `STREAM-ELEMENT-TYPE` is `(UNSIGNED-BYTE 8)` or `(SIGNED-BYTE 8)`. Returns an `INTEGER` or does `end-of-stream` handling, like `READ-BYTE`, if that would return immediately. If `READ-BYTE`'s value is not available immediately, returns `NIL` instead of waiting.

`LISTEN` on binary streams

The [ANSI CL standard] specification for `LISTEN` mentions “character availability” as the criterion that determines the return value. Since a `CHARACTER` is *never* available on a binary `STREAM` (i.e., a stream with `STREAM-ELEMENT-TYPE` being a subtype of `INTEGER`), `LISTEN` returns `NIL` for such streams. (You can use `SOCKET:SOCKET-STATUS` to check binary streams). Any other behavior would be hard to make consistent: consider a bivalent stream, i.e., a `STREAM` that can be operated upon by both `READ-CHAR` and `READ-BYTE`. What should `LISTEN` return on such a stream if what is actually available on the stream at the moment is only a part of a multi-byte character? Right now one can use first `SOCKET:SOCKET-STATUS` to check if anything at all is available and then use `LISTEN` to make sure that a full `CHARACTER` is actually there.

21.3.7. Function `FILE-POSITION`

`FILE-POSITION` works on any `FILE-STREAM`.

Platform Dependent: Win32 platform only.

When a #\Newline is output to (respectively input from) a file stream, its file position is increased by 2 since #\Newline is encoded as CR/LF in the file.

21.3.8. Avoiding blank lines, `EXT:ELASTIC-NEWLINE`

The function `(EXT:ELASTIC-NEWLINE [stream])` is like `FRESH-LINE` but the other way around: It outputs a conditional newline on *`stream`*, which is canceled if the *next* output on *`stream`* happens to be a newline. More precisely, it causes a newline to be output right before the next character is written on *`stream`*, if this character is not a newline. The newline is also output if the next operation on the stream is `FRESH-LINE`, `FINISH-OUTPUT`, `FORCE-OUTPUT` or `CLOSE`.

The functionality of `EXT:ELASTIC-NEWLINE` is also available through the `FORMAT` directive ~..

A technique for avoiding unnecessary blank lines in output is to begin each chunk of output with a call to `FRESH-LINE` and to terminate it with a call to `EXT:ELASTIC-NEWLINE`.

See also `doc/Newline-Convention.txt`.

21.3.9. Function OPEN

OPEN accepts an additional keyword :BUFFERED.

The acceptable values for the arguments to the file/pipe/socket STREAM functions

:ELEMENT-TYPE
: types equivalent to CHARACTER or (UNSIGNED-BYTE *n*), (SIGNED-BYTE *n*); if the stream is to be un:BUFFERED, *n* must be a multiple of 8.
If *n* is not a multiple of 8, **CLISP** will use the specified number of bits for i/o, and write the file length (as a number of *n*-bit bytes) in the preamble.
This is done to ensure the input/output consistency: suppose you open a file with :ELEMENT-TYPE of (UNSIGNED-BYTE 3) and write 7 bytes (i.e., 21 bit) there. The underlying OS can do input/output only in whole 8-bit bytes. Thus the OS will report the size of the file as 3 (8-bit) bytes. Without the preamble **CLISP** will have no way to know how many 3-bit bytes to read from this file - 6, 7 or 8.

:EXTERNAL-FORMAT
: EXT:ENCODINGs, (constant) SYMBOLs in the **"CHARSET"** package, STRINGs (denoting iconv-based encodings), the symbol :DEFAULT, and the line terminator keywords :UNIX, :MAC, :DOS. The default encoding is CUSTOM:*DEFAULT-FILE-ENCODING*. This argument determines how the lisp CHARACTER data is converted to/from the 8-bit bytes that the underlying OS uses.

:BUFFERED
: NIL, T, or :DEFAULT. Have **CLISP** manage an internal buffer for input or output (in addition to the buffering that might be used by the underlying OS). Buffering is a known general technique to significantly speed up i/o.

- for functions that create SOCKET:SOCKET-STREAMs and pipes, :DEFAULT is equivalent to T on the input side and to NIL on the output side; it you are transmitting a lot of data then using buffering will significantly speed up your i/o;
- for functions that open files, :DEFAULT means that buffered file streams will be returned for regular files and (on **UNIX**) block-devices, and unbuffered file streams for special files.

> Note that some files, notably those on the `/proc` filesystem (on **UNIX** systems), are actually, despite their innocuous appearance, special files, so you might need to supply an explicit `:BUFFERED NIL` argument for them. Actually, **CLISP** detects that the file is a `/proc` file, so that one is covered, but there are probably more strange beasts out there!

When an already opened file is opened again, a continuable `ERROR` is `SIGNAL`ed, unless both the existing and the new `STREAM`s are read-only (i.e., `:DIRECTION` is `:INPUT` or `:INPUT-IMMUTABLE`).

21.3.10. Function `CLOSE`

Function `CLOSE` is a `GENERIC-FUNCTION`, see Chapter 30, *Gray streams*.

When the `:ABORT` argument is non-`NIL`, `CLOSE` will not `SIGNAL`s an `ERROR` even when the underlying OS call fails.

`GET-OUTPUT-STREAM-STRING` returns the same value after `CLOSE` as it would before it.

`CLOSE` on an already closed `STREAM` does nothing and returns `T`.

If you do not `CLOSE` your `STREAM` explicitly, it will be closed at garbage-collection time automatically. This is not recommended though because garbage-collection is not deterministic. Please use `WITH-OPEN-STREAM` etc.

21.3.11. Function `OPEN-STREAM-P`

Function `OPEN-STREAM-P` is a `GENERIC-FUNCTION`, see Chapter 30, *Gray streams*.

21.3.12. Class `BROADCAST-STREAM`

`INPUT-STREAM-P` and `INTERACTIVE-STREAM-P` return false for `BROADCAST-STREAM`s.

21.3.13. Functions ***EXT:MAKE-BUFFERED-INPUT-STREAM*** and ***EXT:MAKE-BUFFERED-OUTPUT-STREAM***

(EXT:MAKE-BUFFERED-OUTPUT-STREAM *function*). Returns a buffered output STREAM. *function* is a FUNCTION expecting one argument, a SIMPLE-STRING. WRITE-CHAR collects the CHARACTERs in a STRING, until a newline character is written or FORCE-OUTPUT/FINISH-OUTPUT is called. Then *function* is called with a SIMPLE-STRING as argument, that contains the characters collected so far. CLEAR-OUTPUT discards the characters collected so far.

(EXT:MAKE-BUFFERED-INPUT-STREAM *function* *mode*). Returns a buffered input STREAM. *function* is a FUNCTION of 0 arguments that returns either NIL (stands for end-of-stream) or up to three values *string*, *start*, *end*. READ-CHAR returns the CHARACTERs of the current *string* one after another, as delimited by *start* and *end*, which default to 0 and NIL, respectively. When the *string* is consumed, *function* is called again. The *string* returned by *function* should not be changed by the user. *function* should copy the *string* with COPY-SEQ or SUBSEQ before returning if the original *string* is to be modified. *mode* determines the behavior of LISTEN when the current *string* buffer is empty:

NIL
: the stream acts like a FILE-STREAM, i.e. *function* is called

T
: the stream acts like an interactive stream without end-of-stream, i.e. one can assume that further characters will always arrive, without calling *function*

FUNCTION
: this FUNCTION tells, upon call, if further non-empty *string*s are to be expected.

CLEAR-INPUT discards the rest of the current *string*, so *function* will be called upon the next READ-CHAR operation.

Chapter 22. Printer [CLHS-22]

Table of Contents

22.1. Multiple Possible Textual Representations [CLHS-22.1.1.1]

Variable `CUSTOM:*PRINT-CLOSURE*`. An additional variable `CUSTOM:*PRINT-CLOSURE*` controls whether compiled and interpreted functions (closures) are output in detailed form. If `CUSTOM:*PRINT-CLOSURE*` is non-`NIL`, compiled closures are output in #Y syntax which the reader understands. `CUSTOM:*PRINT-CLOSURE*` is initially set to `NIL`.

Variable `CUSTOM:*PRINT-RPARS*`. An additional variable `CUSTOM:*PRINT-RPARS*` controls the output of the right (closing) parentheses. If `CUSTOM:*PRINT-RPARS*` is non-`NIL`, closing parentheses which do not fit onto the same line as the the corresponding opening parenthesis are output just below their corresponding opening parenthesis, in the same column. CUSTOM:*PRINT-RPARS* is initially set to NIL.

Variable CUSTOM:*PRINT-INDENT-LISTS*. An additional variable CUSTOM:*PRINT-INDENT-LISTS* controls the indentation of lists that span more than one line. It specifies by how many characters items within the list will be indented relative to the beginning of the list. CUSTOM:*PRINT-INDENT-LISTS* is initially set to 1.

Variable CUSTOM:*PPRINT-FIRST-NEWLINE*. An additional variable CUSTOM:*PPRINT-FIRST-NEWLINE* controls pretty-printing of multi-line objects. When CUSTOM:*PPRINT-FIRST-NEWLINE* is non-NIL, and the current line already has some characters on it, and the next object will be printed on several lines, and it does not start with a #\Newline, then a #\Newline is printed before the object. CUSTOM:*PPRINT-FIRST-NEWLINE* has no effect if *PRINT-PRETTY* is NIL. CUSTOM:*PPRINT-FIRST-NEWLINE* is initially set to T.

22.2. Printing Characters [CLHS-22.1.3.2]

Characters are printed as specified in [ANSI CL standard] using #\, with one exception: when printer escaping is in effect, the space character is printed as "#\Space" when the variable CUSTOM:*PRINT-SPACE-CHAR-ANSI* is NIL. When CUSTOM:*PRINT-SPACE-CHAR-ANSI* is non-NIL, it is printed as "#\ "; this is how [ANSI CL standard] specifies it.

22.3. Package Prefixes for Symbols [CLHS-22.1.3.3.1]

Variable CUSTOM:*PRINT-SYMBOL-PACKAGE-PREFIX-SHORTEST*. When CUSTOM:*PRINT-SYMBOL-PACKAGE-PREFIX-SHORTEST* is non-NIL, the package prefix is not the PACKAGE-NAME but the shortest (nick)name as returned by EXT:PACKAGE-SHORTEST-NAME. This variable is ignored when *PRINT-READABLY* is non-NIL.

22.4. Printing Other Vectors [CLHS-22.1.3.7]

When `*PRINT-READABLY*` is true, other vectors are written as follows: if the `ARRAY-ELEMENT-TYPE` is `T`, the syntax `#(`x_0 `...` x_{n-1}`)` is used. Otherwise, the syntax `#A(`*`element-type dimensions contents`*`)` is used.

22.5. Printing Other Arrays [CLHS-22.1.3.8]

When `*PRINT-READABLY*` is true, other arrays are written as follows: if the `ARRAY-ELEMENT-TYPE` is `T`, the syntax `#`*`rank`*`A` *`contents`* is used. Otherwise, the syntax `#A(`*`element-type dimensions contents`*`)` is used.

As explicitly permitted by this section, specialized `BIT` and `CHARACTER` `ARRAY`s are printed with the innermost lists generated by the printing algorithm being instead printed using `BIT-VECTOR` and `STRING` syntax, respectively.

Variable `CUSTOM:*PRINT-EMPTY-ARRAYS-ANSI*`. Empty `ARRAY`s, i.e., arrays with no elements and zero `ARRAY-TOTAL-SIZE` (because one of its dimensions is zero) are printed with the readable syntax `#A(`*`element-type dimensions contents`*`)`, unless the variable `CUSTOM:*PRINT-EMPTY-ARRAYS-ANSI*` is non-`NIL`, in which case the arrays are printed using the [ANSI CL standard]-prescribed syntax `#`*`rank`*`A` *`contents`* which often loses the dimension information.

22.5.1. Printing Pathnames [CLHS-22.1.3.11]

Pathnames are printed as follows: If `*PRINT-ESCAPE*` is `NIL`, only the namestring is printed; otherwise it is printed with the `#P` syntax, as per the [ANSI CL standard] issue PRINT-READABLY-BEHAVIOR:CLARIFY.

But, if *PRINT-READABLY* is true, we are in trouble as #P is ambiguous (which is verboten when *PRINT-READABLY* is true), while being mandated by the [ANSI CL standard]. Therefore, in this case, **CLISP**'s behavior is determined by the value of CUSTOM:*PRINT-PATHNAMES-ANSI*: when it is NIL, we print pathnames like this: #-CLISP #P"..." #+CLISP #S(PATHNAME ...). Otherwise, when the variable CUSTOM:*PRINT-PATHNAMES-ANSI* is non-NIL, the #P notation is used as per 1.5.1.4.1 Resolution of Apparent Conflicts in Exceptional Situations.

Note

The #S notation for PATHNAMEs is used extensively in the [Common Lisp HyperSpec] (see examples for PATHNAME, PATHNAMEP, PARSE-NAMESTRING et al), but was decided against, see PATHNAME-PRINT-READ:SHARPSIGN-P.

Warning

When both *PRINT-READABLY* and CUSTOM:*PRINT-PATHNAMES-ANSI* are non-NIL and the namestring will be parsed to a dissimilar object (with the current value of CUSTOM:*PARSE-NAMESTRING-DOT-FILE*), an ERROR of type PRINT-NOT-READABLE is SIGNALed.

22.6. The Lisp Pretty Printer [CLHS-22.2]

The Lisp Pretty Printer implementation is **not** perfect yet. PPRINT-LOGICAL-BLOCK does not respect *PRINT-LINES*.

22.6.1. Pretty Print Dispatch Table [CLHS-22.2.1.4]

A pprint dispatch table is a `CONS` of a `SYMBOL` `*PRINT-PPRINT-DISPATCH*` and an association list which maps types into priorities and print functions. Their use is strongly discouraged because of the performance issues: when `*PRINT-PPRINT-DISPATCH*` is non-trivial and `*PRINT-PRETTY*` is non-`NIL`, printing of every object requires a lookup in the table, which entails many calls to `TYPEP` (which cannot be made fast enough).

22.7. Formatted Output [CLHS-22.3]

Function FORMAT

The additional `FORMAT` instruction **~!** is similar to `~/`, but avoids putting a function name into a string, thus, even if the function is not interned in the **"COMMON-LISP-USER"** package, you might not need to specify the package explicitly. `(FORMAT` *`stream`* `"~arguments!"` *`function object`*`)` is equivalent to `(FUNCALL` *`function stream object colon-modifier-p atsign-modifier-p arguments`*`)`.

The additional `FORMAT` instruction **~.** is a kind of opposite to `~&`: It outputs a conditional newline, by calling the function `EXT:ELASTIC-NEWLINE`. `~`*`n`*`.` outputs *`n-1`* newlines followed by an `EXT:ELASTIC-NEWLINE`. `~0.` does nothing.

`FORMAT` `~R` and `FORMAT` `~:R` can output only integers in the range `|n| <` 10^{66}. The output is in English, according to the American conventions, and these conventions are identical to the British conventions only in the range `|n| <` 10^{9}.

`FORMAT` `~:@C` does not output the character itself, only the instruction how to type the character.

For `FORMAT` `~E` and `FORMAT` `~G`, the value of `*READ-DEFAULT-FLOAT-FORMAT*` does not matter if `*PRINT-READABLY*` is true.

FORMAT ~T can determine the current column of any built-in stream.

22.8. The Printer Dictionary [CLHS-22.4]

22.8.1. Functions WRITE & WRITE-TO-STRING

The functions WRITE and WRITE-TO-STRING have an additional keyword argument :CLOSURE which is used to bind CUSTOM:*PRINT-CLOSURE*.

22.8.2. Macro PRINT-UNREADABLE-OBJECT

Variable CUSTOM:*PRINT-UNREADABLE-ANSI*. The macro PRINT-UNREADABLE-OBJECT, when invoked without body forms, suppresses the trailing space if only the type is to be printed, and suppresses the leading space if only the identity is to be printed. This behaviour can be turned off set setting the variable CUSTOM:*PRINT-UNREADABLE-ANSI* to a non-NIL value: in this case, a trailing or leading space are output, as prescribed by [ANSI CL standard].

22.8.3. Miscellaneous Issues

PRINT-CASE controls the output not only of symbols, but also of characters and some unreadable #< objects.

In the absence of ***SYS::WRITE-FLOAT-DECIMAL*** , floating point numbers are output in radix 2. This function is defined in floatprint.lisp and is not available if you run **CLISP** without a memory image (which you should never do anyway!)

If *PRINT-READABLY* is true, *READ-DEFAULT-FLOAT-FORMAT* has no influence on the way floating point numbers are printed.

Platform Dependent: UNIX, Win32 platforms only.

> `*PRINT-PRETTY*` is initially `NIL` but set to `T` in `config.lisp`. This makes screen output prettier.

`*PRINT-ARRAY*` is initially set to `T`.

Chapter 23. Reader [CLHS-23]

Table of Contents

23.1. Effect of Readtable Case on the Lisp Reader [CLHS-23.1.2]

When the value of `(READTABLE-CASE readtable)` is `:INVERT`, it applies to the package name and the symbol name of a symbol separately (not to the entire token at once). An alternative to the use of `READTABLE-CASE` is the use of the `:CASE-SENSITIVE` option of `MAKE-PACKAGE` and `DEFPACKAGE`.

23.2. The *`recursive-p`* argument [CLHS-23.1.3.2]

When non-`NIL` *`recursive-p`* argument is passed to a top-level `READ` call, an `ERROR` is `SIGNAL`ed.

Chapter 24. System Construction [CLHS-24]

Table of Contents

24.1. The System Construction Dictionary [CLHS-24.2]

24.1.1. Function COMPILE-FILE
24.1.2. Function COMPILE-FILE-PATHNAME
24.1.3. Function REQUIRE
24.1.4. Function LOAD
24.1.5. Variable *FEATURES*
24.1.6. Function EXT:FEATUREP [CLRFI-1]
24.1.7. Function EXT:COMPILED-FILE-P [CLRFI-2]

The compiler can be called not only by the functions COMPILE, COMPILE-FILE and DISASSEMBLE, but also by the declaration (COMPILE).

24.1.1. Function COMPILE-FILE

COMPILE-FILE compiles a file to a platform-independent bytecode:

```
(COMPILE-FILE filename &KEY :OUTPUT-FILE :LISTING :EXTERN
                            ((:WARNINGS CUSTOM:*COMPILE-W
                            ((:VERBOSE *COMPILE-VERBOSE*)
                            ((:PRINT *COMPILE-PRINT*) *CO
```

Options for COMPILE-FILE

filename
: the file to be compiled, should be a pathname designator.

:OUTPUT-FILE
: should be NIL or T or a pathname designator or an output STREAM. The default is T.

`:LISTING`
: should be `NIL` or `T` or a pathname designator or an output `STREAM`. The default is `NIL`.

`:EXTERNAL-FORMAT`
: the `EXT:ENCODING` of the *`filename`*.

`:WARNINGS`
: specifies whether warnings should also appear on the screen.

`:VERBOSE`
: specifies whether error messages should also appear on the screen.

`:PRINT`
: specifies whether an indication which forms are being compiled should appear on the screen.

The variables ***`CUSTOM:*COMPILE-WARNINGS*`***, `*COMPILE-VERBOSE*`, `*COMPILE-PRINT*` provide defaults for the `:WARNINGS`, `:VERBOSE`, `:PRINT` keyword arguments, respectively, and are bound by `COMPILE-FILE` to the values of the arguments, i.e., these arguments are recursive.

For each input file (default file type: `#P".lisp"`) the following files are generated:

File	When	Default file type	Contents
output file	only if `:OUTPUT-FILE` is not `NIL`	`#P".fas"`	can be loaded using the `LOAD` function
auxiliary output file	only if `:OUTPUT-FILE` is not `NIL`	`#P".lib"`	used by `COMPILE-FILE` when compiling a `REQUIRE` form referring to the input file
listing file	only if `:LISTING` is not `NIL`	`#P".lis"`	disassembly of the output file
C output file	only if `:OUTPUT-FILE` is not `NIL`	`#P".c"`	**"FFI"**; this file is created only if the source contains **"FFI"** forms

Warning

If you have two files in the same directory - `#P"foo.lisp"` and `#P"foo.c"`, and you compile the first file with **CLISP**, the second file will be *clobbered* if you have any **"FFI"** forms in the first one!

24.1.2. Function `COMPILE-FILE-PATHNAME`

The default for the `:OUTPUT-FILE` argument is `T`, which means `#P".fas"`.

24.1.3. Function `REQUIRE`

The function `REQUIRE` receives as the optional argument either a `PATHNAME` or a `LIST` of `PATHNAME`s: files to be `LOAD`ed if the required module is not already present.

At compile time, `(REQUIRE #P"foo")` forms are treated specially: `CUSTOM:*LOAD-PATHS*` is searched for `#P"foo.lisp"` **and** `#P"foo.lib"`. If the latest such file is a `#P".lisp"`, it is compiled; otherwise the `#P".lib"` is loaded.

The `#P".lib"` is a "header" file which contains the constant, variable, inline and macro definitions necessary for compilation of the files that `REQUIRE` this file, but not the function definitions and calls that are not necessary for that. Thus it is **not** necessary to either enclose `REQUIRE` forms in `EVAL-WHEN` or to load the required files in the makefiles: if you have two files, `#P"foo.lisp"` and `#P"bar.lisp"`, and the latter requires the former, you can write in your `Makefile`:

```
all: foo.fas bar.fas

foo.fas: foo.lisp
        clisp -c foo

bar.fas: bar.lisp foo.fas
        clisp -c bar
```

instead of the more cumbersome (and slower, since `#P".lib"`s are usually smaller and load faster that `#P".fas"`s):

```
bar.fas: bar.lisp foo.fas
        clisp -i foo -c bar
```

Thus, you do not need to `(LOAD #P"foo")` in order to `(COMPILE-FILE #P"bar.lisp")`. If memory is tight, and if `#P"foo.lisp"` contains only a few inline functions, macros, constants or variables, this is a space and time saver. If `#P"foo.lisp"` does a lot of initializations or side effects when being loaded, this is important as well.

24.1.4. Function LOAD

`LOAD` accepts four additional keyword arguments `:ECHO`, `:COMPILING`, `:EXTRA-FILE-TYPES`, and `:OBSOLETE-ACTION`.

```
(LOAD filename &KEY ((:VERBOSE *LOAD-VERBOSE*) *LOAD-VERB
                    ((:PRINT *LOAD-PRINT*) *LOAD-PRINT*)
                    ((:ECHO CUSTOM:*LOAD-ECHO*) CUSTOM:*L
                    ((:COMPILING CUSTOM:*LOAD-COMPILING*)
                    ((:OBSOLETE-ACTION CUSTOM:*LOAD-OBSOL
```

:VERBOSE
: causes `LOAD` to emit a short message that a file is being loaded. The default is `*LOAD-VERBOSE*`, which is initially `T`, but can be changed by the `-v` option.

:PRINT
: causes `LOAD` to print the value of each form. The default is `*LOAD-PRINT*`, which is initially `NIL`, but can be changed by the `-v` option.

:ECHO
: causes the input from the file to be echoed to `*STANDARD-OUTPUT*` (normally to the screen). Should there be an error in the file, you can see at one glance where it is. The default is ***`CUSTOM:*LOAD-ECHO*`***, which is initially `NIL`, but can be changed by the `-v` option.

:COMPILING
: causes each form read to be compiled on the fly. The compiled code is executed at once and - in contrast to `COMPILE-FILE` - not written to a file. The default is ***`CUSTOM:*LOAD-COMPILING*`***, which is initially `NIL`, but can be changed by the `-C` option.

:EXTRA-FILE-TYPES
: Specifies the `LIST` of additional file types considered for loading, in addition to ***`CUSTOM:*SOURCE-FILE-TYPES*`*** (which is initially

("lisp" "lsp" "cl")) and ***CUSTOM:*COMPILED-FILE-TYPES**** (which is initially ("fas")).

When *filename* does not specify a unique file (e.g., *filename* is #P"foo" and both #P"foo.lisp" and #P"foo.fas" are found in the CUSTOM:*LOAD-PATHS*), then the *newest* file is loaded.

:OBSOLETE-ACTION

Specifies the action to take when loading a #P".fas" with a different bytecode version from the one supported by this **CLISP** version. The possible actions are

:DELETE

delete the #P".fas"

:ERROR

SIGNAL an ERROR

:COMPILE

recompile the source file (if present)

NIL (default)

WARN and look for another matching file

If no file can be loaded and :IF-DOES-NOT-EXIST is non-NIL, an ERROR is SIGNALed. The default is ***CUSTOM:*LOAD-OBSOLETE-ACTION****, which is initially NIL.

The variables *LOAD-VERBOSE*, *LOAD-PRINT*, CUSTOM:*LOAD-OBSOLETE-ACTION*, CUSTOM:*LOAD-COMPILING*, and CUSTOM:*LOAD-ECHO* are bound by LOAD when it receives a corresponding keyword argument (:VERBOSE, :PRINT, :OBSOLETE-ACTION, :COMPILING, and :ECHO), i.e., these arguments are recursive, just like the arguments :WARNINGS, :VERBOSE, and :PRINT for COMPILE-FILE.

When evaluation of a read form SIGNALs an ERROR, two RESTART-s are available:

SKIP

Skip this form and read the next one.

STOP

Stop loading the file.

Variable CUSTOM:*LOAD-PATHS*. The variable CUSTOM:*LOAD-PATHS* contains a list of directories where the files are looked for - in addition to the specified or current directory - by LOAD, REQUIRE, COMPILE-FILE and LOAD-LOGICAL-PATHNAME-TRANSLATIONS.

24.1.5. Variable *FEATURES*

The variable *FEATURES* initially contains the following symbols

Default *FEATURES*

:CLISP
: the name of this implementation

:ANSI-CL
: **CLISP** purports to conform to [ANSI CL standard]

:COMMON-LISP
: required by [ANSI CL standard]

:INTERPRETER
: EVAL is implemented

:COMPILER
: COMPILE and COMPILE-FILE are implemented

:SOCKETS
: see Section 32.5, “Socket Streams”

:GENERIC-STREAMS
: see Section 31.6, “Generic streams”

:LOGICAL-PATHNAMES
: Logical Pathnames are implemented

:FFI
: if a foreign function interface (see Section 32.3, “The Foreign Function Call Facility”) is supported (Platform Dependent: Many **UNIX**, **Win32** platforms only)

:GETTEXT
: if internationalization (see Section 31.4, “Internationalization of **CLISP**”) using the GNU gettext package is supported (Platform Dependent: most **UNIX** platforms only)

:UNICODE
: if **UNICODE** (ISO 10646) characters are supported (see Section 31.5, “Encodings”)

:LOOP
: “extended” LOOP form is implemented

:CLOS
: **CLOS** is implemented

:MOP

Meta-Object Protocol is implemented

`:WIN32`
: if *`hardware`* = PC (clone) and *`operating system`* = **Win32** (Windows 95/98/Me/NT/2000/XP)

`:PC386`
: if *`hardware`* = PC (clone). It can be used as an indicator for the mainstream hardware characteristics (such as the existence of a graphics card with a non-graphics text mode, or the presence of a keyboard with arrows and Insert/Delete keys, or an ISA/VLB/PCI bus) or software characteristics (such as the **Control**+**Alternate**+Delete keyboard combination).

`:UNIX`
: if *`operating system`* = **UNIX** (in this case the *`hardware`* is irrelevant!)

`:BEOS`
: if *`operating system`* = **BeOS** (in that case `:UNIX` is also present)

`:CYGWIN`
: if **CLISP** is using the **Cygwin** **UNIX** compatibility layer on top of **Win32** (in that case `:UNIX` is also present)

`:MACOS`
: if *`operating system`* = **Mac OS X** (in that case `:UNIX` is also present)

Each module should add the appropriate keyword, e.g., `:SYSCALLS`, `:DIRKEY`, `:REGEXP`, `:PCRE`, etc.

24.1.6. Function `EXT:FEATUREP` [CLRFI-1]

`(EXT:FEATUREP` *`form`*`)` provides run-time access to the read-time conditionals `#+` and `#-`. *`form`* is a feature exression.

24.1.7. Function `EXT:COMPILED-FILE-P` [CLRFI-2]

`(EXT:COMPILED-FILE-P` *`filename`*`)` returns non-`NIL` when the file *`filename`* exists, is readable, and appears to be a **CLISP**-compiled `#P".fas"` file compatible with the currently used bytecode format.

System definition facilities (such as asdf or defsystem) can use it to determine whether the file needs to be recompiled.

Chapter 25. Environment [CLHS-25]

Table of Contents

25.1. Debugging Utilities [CLHS-25.1.2]

25.1.1. User-customizable Commands

The debugger may be invoked through the functions INVOKE-DEBUGGER, BREAK, SIGNAL, ERROR, CERROR, WARN. The stepper is invoked through the macro STEP. Debugger and stepper execute subordinate read-eval-print loop (called "break loops") which are similar to the main read-eval-print loop except for the prompt and the set of available commands. Commands must be typed literally, in any case, without surrounding quotes or whitespace. Each command has a keyword abbreviation, indicated in the second column.

Table 25.1. Commands common to the main loop, the debugger and the stepper

command	abbreviation	operation
Help	**:h**	prints a list of available commands

Table 25.2. Commands common to the debugger and the stepper

command	abbreviation	operation
Abort	**:a**	abort to the next most recent read-eval-print loop
Unwind	**:uw**	abort to the next most recent read-eval-print loop
Quit	**:q**	quit to the top read-eval-print loop

The stack is organized into frames and other stack elements. Usually every invocation of an interpreted function and every evaluation of an interpreted form corresponds to one stack frame. Special forms such as `LET`, `LET*`, `UNWIND-PROTECT` and `CATCH` produce special kinds of stack frames.

In a break loop there is a ***current stack frame***, which is initially the most recent stack frame but can be moved using the debugger commands **Up** and **Down**.

Evaluation of forms in a break loop occurs in the lexical environment of the current stack frame and *at the same time* in the dynamic environment of the debugger's caller. This means that to inspect or modify a lexical variable all you have to do is to move the current stack frame to be just below the frame that corresponds to the form or the function call that binds that variable.

There is a current ***stack mode*** which defines in how much detail the stack is shown by the stack-related debugger commands.

Table 25.3. Commands common to the debugger and the stepper

command	abbreviation	operation
Error	**:e**	print the last error object.
Inspect	**:i**	`INSPECT` the last error object.
Mode-1	**:m1**	sets the current mode to 1: all the stack elements are considered. This mode works fine for debugging compiled functions.
Mode-2	**:m2**	sets the current mode to 2: all the frames are considered.
Mode-3	**:m3**	sets the current mode to 3: only lexical frames (frames that correspond to special forms that modify the lexical environment) are considered.
Mode-4	**:m4**	sets the current mode to 4 (the default): only `EVAL` and `APPLY` frames are considered. Every evaluation of a form in the interpreter corresponds to an EVAL frame.
Mode-5	**:m5**	sets the current mode to 5: only `APPLY` frames are considered. Every invocation of an interpreted function corresponds to one `APPLY` frame.
Where	**:w**	shows the current stack frame.
Up	**:u**	goes up one frame, i.e., to the caller if in mode -5
Down	**:d**	does down one frame, i.e., to the callee if in mode-5
Top	**:t**	goes to top frame, i.e., to the top-level form if in mode-4
Bottom	**:b**	goes to bottom (most recent) frame, i.e., most probably to the form or function that caused the debugger to be entered.
Backtrace	**:bt**	lists the stack in current mode, bottom frame first, top frame last.

command	abbreviation	operation
Backtrace -1	**:bt1**	lists the stack in mode 1.
Backtrace -2	**:bt2**	lists the stack in mode 2.
Backtrace -3	**:bt3**	lists the stack in mode 3.
Backtrace -4	**:bt4**	lists the stack in mode 4.
Backtrace -5	**:bt5**	lists the stack in mode 5.
Frame-limit	**:fl**	set the frame-limit: this many frames will be printed in a backtrace at most.
Backtrace -l	**:bl**	limit of frames to print will be prompted for.

If the current stack frame is an `EVAL` or `APPLY` frame, the following commands are available as well:

Table 25.4. Commands specific to `EVAL`/`APPLY`

command	abbreviation	operation
Break+	**:br+**	sets a breakpoint in the current frame. When the corresponding form or function will be left, the debugger will be entered again, with the variable `EXT:*TRACE-VALUES*` containing a list of its values.
Break-	**:br-**	removes a breakpoint from the current frame.
Redo	**:rd**	re-evaluates the corresponding form or function call. This command can be used to restart parts of a computation without aborting it entirely.
Return	**:rt**	leaves the current frame. You will be prompted for the return values.

Table 25.5. Commands specific to the debugger

command	abbreviation	operation
Continue	**:c**	continues evaluation of the program.

Table 25.6. Commands specific to the stepper

command	abbreviation	operation
Step	**:s**	step into a form: evaluate this form in single step mode
Next	**:n**	step over a form: evaluate this form at once
Over	**:o**	step over this level: evaluate at once up to the next return
Continue	**:c**	switch off single step mode, continue evaluation

The stepper is usually used like this: If some form returns a strange value or results in an error, call `(STEP form)` and navigate using the commands **Step** and **Next** until you reach the form you regard as responsible. If you are too fast (execute **Next** once and get the error), there is no way back; you have to restart the entire stepper session. If you are too slow (stepped into a function or a form which certainly is OK), a couple of **Next** commands or one **Over** command will help.

25.1.1. User-customizable Commands

You can set ***`CUSTOM:*USER-COMMANDS*`*** to a list of `FUNCTION`s, each returning a `LIST` of ***bindings***, i.e., either a

`STRING`
: the help string printed by **Help** in addition to the standard **CLISP** help

`CONS (STRING . FUNCTION)`
: the actual binding: when the user types the string, the function is called.

E.g.,

```
(setq CUSTOM:*USER-COMMANDS*
      (list (lambda () (list (format nil "~2%User-defined
            (lambda ()
              (flet ((panic () (format t "don't panic, ~D
                (list (format nil "~%panic     :p     hit the
                      (cons "panic" #'panic)
                      (cons ":p" #'panic))))
            (lambda ()
              (let ((curses #("ouch" "yuk" "bletch")))
                (flet ((swear ()
                         (format t "~A!~%"
                                 (aref curses (random (len
                  (list (format nil "~%swear    :e     curse
                        (cons "swear" #'swear)
                        (cons ":e" #'swear)))))))
```

25.2. The Environment Dictionary [CLHS-25.2]

25.2.1. Function DISASSEMBLE

Platform Dependent: UNIX platform only.

DISASSEMBLE can disassemble to machine code, provided that GNU **gdb** is present. In that case the argument may be a EXT:SYSTEM-FUNCTION, a FFI:FOREIGN-FUNCTION, a special operator handler, a SYMBOL denoting one of these, an INTEGER (address), or a STRING.

25.2.2. Function EXT:UNCOMPILE

The function EXT:UNCOMPILE does the converse of COMPILE: (EXT:UNCOMPILE *function*) reverts a compiled *function* (name), that has been entered or loaded in the same session and then compiled, back to its interpreted form.

25.2.3. Function DOCUMENTATION

No on-line documentation is available for the system functions (yet), but see Section 25.2.4, "Function DESCRIBE".

25.2.4. Function DESCRIBE

When CUSTOM:*BROWSER* is non-NIL, and CUSTOM:CLHS-ROOT returns a valid URL, DESCRIBE on a standard **Common Lisp** symbol will point your web browser to the appropriate [Common Lisp HyperSpec] page.

Also, when CUSTOM:*BROWSER* is non-NIL, and CUSTOM:IMPNOTES-ROOT returns a valid URL, DESCRIBE on symbols and packages documented in these implementation notes will point your web browser to the appropriate page.

Function CUSTOM:CLHS-ROOT. Function CUSTOM:CLHS-ROOT is defined in config.lisp. By default it looks at (EXT:GETENV "CLHSROOT") and CUSTOM:*CLHS-ROOT-DEFAULT*, but you may redefine it in config.lisp or RC file. The return value should be a STRING terminated with a "/", e.g., http://www.lisp.org/HyperSpec/

or `/usr/doc/HyperSpec/`. If the return value is `NIL`, the feature is completely disabled.

Function `CUSTOM:IMPNOTES-ROOT`. Function `CUSTOM:IMPNOTES-ROOT` is defined in `config.lisp`. By default it looks at `(EXT:GETENV "IMPNOTES")` and `CUSTOM:*IMPNOTES-ROOT-DEFAULT*`, but you may redefine it in `config.lisp` or RC file. The return value should be a `STRING` terminated with a `"/"`, e.g., http://clisp.cons.org/impnotes/, or the path to the monolithic page, e.g., http://clisp.cons.org/impnotes.html or `/usr/doc/clisp/impnotes.html`. If the return value is `NIL`, the feature is completely disabled.

25.2.5. Macro TRACE

`(TRACE function ...)` makes the functions *`function`*, ... traced. *`function`* should be either a symbol or a list `(symbol &KEY :SUPPRESS-IF :MAX-DEPTH :STEP-IF :PRE :POST :PRE-BREAK-IF :POST-BREAK-IF :PRE-PRINT :POST-PRINT :PRINT)`, where

`:SUPPRESS-IF` ***`form`***
: no trace output as long as *`form`* is true

`:MAX-DEPTH` ***`form`***
: no trace output as long as `(> *trace-level* form)`. This is useful for tracing functions that are use by the tracer itself, such as `PRINT-OBJECT`, or otherwise when tracing would lead to an infinite recursion.

`:STEP-IF` ***`form`***
: invokes the stepper as soon as *`form`* is true

`:BINDINGS` **((*`variable`* *`form`*)...)**
: binds *`variable`*s to the result of evaluation of *`form`*s around evaluation of all of the following forms

`:PRE` ***`form`***
: evaluates *`form`* before calling the function

`:POST` ***`form`***
: evaluates *`form`* after return from the function

`:PRE-BREAK-IF` ***`form`***
: goes into the break loop before calling the function if *`form`* is true

`:POST-BREAK-IF` ***`form`***

goes into the break loop after return from the function if *`form`* is true

`:PRE-PRINT` ***`form`***
prints the values of *`form`* before calling the function

`:POST-PRINT` ***`form`***
prints the values of *`form`* after return from the function

`:PRINT` ***`form`***
prints the values of *`form`* both before calling and after return from the function

In all these forms you can access the following variables:

`EXT:*TRACE-FUNCTION*`
the function itself

`EXT:*TRACE-ARGS*`
the arguments to the function

`EXT:*TRACE-FORM*`
the function/macro call as form

`EXT:*TRACE-VALUES*`
after return from the function: the list of return values from the function call

and you can leave the function call with specified values by using `RETURN`.

`TRACE` and `UNTRACE` are also applicable to functions `(SETF` *`symbol`*`)` and to macros, but not to locally defined functions and macros.

Trace output

`TRACE` prints this line before evaluating the form: `*trace level*. Trace: *form*` and after evaluating the form it prints: `*trace level*. Trace: *function-name* ==> *result*` where “trace level” is the total nesting level.

Example

Suppose the trace level above is not enough for you to identify individual calls. You can give each call a unique id and print it:

```
(defun f0 (x)
  (cond ((zerop x) 1)
        ((zerop (random 2)) (* x (f0 (1- x))))
        (t (* x (f1 (1- x))))))
⇒ F0
(defun f1 (x)
  (cond ((zerop x) 1)
        ((zerop (random 2)) (* x (f0 (1- x))))
        (t (* x (f1 (1- x))))))
⇒ F1
(defvar *f0-call-count* 0)
⇒ *F0-CALL-COUNT*
(defvar *id0*)
⇒ *ID0*
(defvar *cc0*)
⇒ *CC0*
(defvar *f1-call-count* 0)
⇒ *F1-CALL-COUNT*
(defvar *id1*)
⇒ *ID1*
(defvar *cc1*)
⇒ *CC1*
(trace (f0 :bindings ((*cc0* (incf *f0-call-count*))
                      (*id0* (gensym "F0-")))
           :pre-print (list 'enter *id0* *cc0*)
           :post-print (list 'exit *id0* *cc0*))
       (f1 :bindings ((*cc1* (incf *f1-call-count*))
                      (*id1* (gensym "F1-")))
           :pre-print (list 'enter *id1* *cc1*)
           :post-print (list 'exit *id1* *cc1*)))
;; Tracing function F0.
;; Tracing function F1.
⇒ (F0 F1)
(f0 10)
1. Trace: (F0 '10)
(ENTER #:F0-2926 1)
2. Trace: (F1 '9)
(ENTER #:F1-2927 1)
3. Trace: (F0 '8)
(ENTER #:F0-2928 2)
4. Trace: (F1 '7)
(ENTER #:F1-2929 2)
5. Trace: (F1 '6)
(ENTER #:F1-2930 3)
6. Trace: (F1 '5)
(ENTER #:F1-2931 4)
```

```
7. Trace: (F1 '4)
(ENTER #:F1-2932 5)
8. Trace: (F0 '3)
(ENTER #:F0-2933 3)
9. Trace: (F1 '2)
(ENTER #:F1-2934 6)
10. Trace: (F0 '1)
(ENTER #:F0-2935 4)
11. Trace: (F1 '0)
(ENTER #:F1-2936 7)
(EXIT #:F1-2936 7)
11. Trace: F1 ==> 1
(EXIT #:F0-2935 4)
10. Trace: F0 ==> 1
(EXIT #:F1-2934 6)
9. Trace: F1 ==> 2
(EXIT #:F0-2933 3)
8. Trace: F0 ==> 6
(EXIT #:F1-2932 5)
7. Trace: F1 ==> 24
(EXIT #:F1-2931 4)
6. Trace: F1 ==> 120
(EXIT #:F1-2930 3)
5. Trace: F1 ==> 720
(EXIT #:F1-2929 2)
4. Trace: F1 ==> 5040
(EXIT #:F0-2928 2)
3. Trace: F0 ==> 40320
(EXIT #:F1-2927 1)
2. Trace: F1 ==> 362880
(EXIT #:F0-2926 1)
1. Trace: F0 ==> 3628800
⇒ 3628800
*f0-call-count*
⇒ 4
*f1-call-count*
⇒ 7
```

Variable CUSTOM:*TRACE-INDENT*

If you want the TRACE level to be indicated by the indentation in addition to the printed numbers, set CUSTOM:*TRACE-INDENT* to non-NIL.

Initially it is NIL since many nested traced calls will easily exhaust the available line length.

25.2.6. Function INSPECT

The function INSPECT takes a keyword argument :FRONTEND, which specifies the way **CLISP** will interact with the user, and defaults to CUSTOM:*INSPECT-FRONTEND*.

Available :FRONTENDs for INSPECT in CLISP

:TTY
: The interaction is conducted via the *TERMINAL-IO* stream. Please use the **:h** command to get the list of all available commands.

:HTTP
: A window in your Web browser (specified by the :BROWSER keyword argument) is opened and it is controlled by **CLISP** via a SOCKET:SOCKET-STREAM, using the **HTTP** protocol. You should be able to use all the standard browser features.
Since **CLISP** is not multitasking at this time, you will not be able to do anything else during an INSPECT session. Please click on the **quit** link to terminate the session.
Please be aware though, that once you terminate an INSPECT session, all links in all INSPECT windows in your browser will become obsolete and using them in a new INSPECT session will result in unpredictable behavior.

The function INSPECT also takes a keyword argument :BROWSER, which specifies the browser used by the :HTTP front-end and defaults to CUSTOM:*INSPECT-BROWSER*.

The function INSPECT binds some pretty-printer variables:

Variable	Bound to
PRINT-LENGTH	CUSTOM:*INSPECT-PRINT-LENGTH*
PRINT-LEVEL	CUSTOM:*INSPECT-PRINT-LEVEL*
PRINT-LINES	CUSTOM:*INSPECT-PRINT-LINES*

User variable ***CUSTOM:*INSPECT-LENGTH**** specifies the number of sequence elements printed in detail when a sequence is inspected.

25.2.7. Function ROOM

The function `ROOM` returns two values: the number of bytes currently occupied by Lisp objects, and the number of bytes that can be allocated before the next regular garbage-collection occurs.

The function ***EXT:GC*** starts a global garbage-collection and its return value has the same meaning as the second value of `ROOM`.

25.2.8. Macro TIME

The timing data printed by the macro `TIME` includes:

the real time (“wall” time),
the run time (processor time for this process),
the number of bytes allocated, and
the number of garbage-collections performed, if any.

The macro ***EXT:TIMES*** (mnemonic: “*TIME* and Space”) is like the macro `TIME`: `(EXT:TIMES form)` evaluates the *form*, and, as a side effect, outputs detailed information about the memory allocations caused by this evaluation. It also prints everything printed by `TIME`.

25.2.9. Function ED

The function `ED` calls the external editor specified by the value of `(EXT:GETENV "EDITOR")` or, failing that, the value of the variable ***CUSTOM:*EDITOR**** (set in `config.lisp`). If the argument is a function name which was defined in the current session (not loaded from a file), the program text to be edited is a pretty-printed version (without comments) of the text which was used to define the function.

25.2.10. Clock Time

Default Time Zone

Platform Dependent: No platform supports this currently

The variable ***CUSTOM:*DEFAULT-TIME-ZONE**** contains the default time zone used by `ENCODE-UNIVERSAL-TIME` and `DECODE-UNIVERSAL-TIME`. It is initially set to -1 (which means 1 hour east of Greenwich, i.e., Mid European Time).

The time zone in a decoded time does not necessarily have be an `INTEGER`, but (as `FLOAT` or `RATIONAL` number) it should be a multiple of `1/3600`.

Table 25.7. Time granularity

platform	UNIX	Win32
`INTERNAL-TIME-UNITS-PER-SECOND`	1,000,000	10,000,000

`GET-INTERNAL-RUN-TIME` returns the amount of run time consumed by the current **CLISP** process since its startup.

25.2.11. Machine

Platform Dependent: UNIX platform only.

The functions `SHORT-SITE-NAME`, `LONG-SITE-NAME` should be defined in a site-specific `config.lisp` file. The default implementations try to read the value of the environment variable `ORGANIZATION`, and, failing that, call `uname`.

Platform Dependent: Win32 platform only.

The functions `SHORT-SITE-NAME`, `LONG-SITE-NAME` should be defined in a site-specific `config.lisp` file. The default implementations try to read the registry.

Platform Dependent: No platform supports this currently

The functions `MACHINE-TYPE`, `MACHINE-VERSION`, `MACHINE-INSTANCE` and `SHORT-SITE-NAME`, `LONG-SITE-NAME` should be defined by every user in his user-specific `config.lisp` file.

25.2.12. Functions APROPOS & APROPOS-LIST

The search performed by APROPOS and APROPOS-LIST is case-insensitive.

Variable CUSTOM:*APROPOS-DO-MORE*. You can make APROPOS print more information about the symbols it found by setting CUSTOM:*APROPOS-DO-MORE* to a list containing some of :FUNCTION, :VARIABLE, :TYPE, and :CLASS or just set it to T to get all of the values.

Variable CUSTOM:*APROPOS-MATCHER*. You can make APROPOS and APROPOS-LIST be more flexible in their search by setting CUSTOM:*APROPOS-MATCHER* to a FUNCTION of one argument, a pattern (a STRING), returning a new FUNCTION of one argument, a SYMBOL name (also a STRING), which returns non-NIL when the symbol name matches the pattern for the purposes of APROPOS. When CUSTOM:*APROPOS-MATCHER* is NIL, SEARCH is used. Some modules come with functions which can be used for CUSTOM:*APROPOS-MATCHER*, e.g., REGEXP:REGEXP-MATCHER, WILDCARD:WILDCARD-MATCHER, PCRE:PCRE-MATCHER.

25.2.13. Function DRIBBLE

25.2.13.1. Scripting and DRIBBLE

If DRIBBLE is called with an argument, and dribbling is already enabled, a warning is printed, and the new dribbling request is ignored.

Dribbling is implemented via a kind (but **not** a recognizable subtype) of TWO-WAY-STREAM, named EXT:DRIBBLE-STREAM. If you have a *source* bidirectional STREAM *x* and you want all transactions (input and output) on *x* to be copied to the *target* output STREAM *y*, you can do

```
(DEFVAR *loggable* x)
(SETQ x (MAKE-SYNONYM-STREAM '*loggable*))
(DEFUN toggle-logging (&OPTIONAL s)
  (MULTIPLE-VALUE-BIND (so ta) (dribble-toggle *loggable*
    (WHEN (STREAMP so) (SETQ *loggable* so))
```

```
    ta))
(toggle-logging y)      ; start logging
...
(toggle-logging)        ; finish logging
...
(toggle-logging y)      ; restart logging
...
(toggle-logging)        ; finish logging
(CLOSE y)
```

(EXT:DRIBBLE-STREAM *stream*)
: When *stream* is a EXT:DRIBBLE-STREAM, returns two values: the *source* and the *target* streams. Otherwise returns NIL.

(EXT:DRIBBLE-STREAM-P *stream*)
: When *stream* is a EXT:DRIBBLE-STREAM, returns T, otherwise returns NIL.

(EXT:DRIBBLE-STREAM-SOURCE *stream*)
: When *stream* is a EXT:DRIBBLE-STREAM, returns its *source* stream, otherwise signals a TYPE-ERROR.

(EXT:DRIBBLE-STREAM-TARGET *stream*)
: When *stream* is a EXT:DRIBBLE-STREAM, returns its *target* stream, otherwise signals a TYPE-ERROR.

(EXT:MAKE-DRIBBLE-STREAM *source target*)
: Create a new EXT:DRIBBLE-STREAM.

(EXT:DRIBBLE-TOGGLE *stream* &OPTIONAL *pathname*)
: When *stream* is a EXT:DRIBBLE-STREAM and *pathname* is NIL, writes a dribble termination note to the *stream*'s *target* STREAM and returns *stream*'s *source* and *target* STREAMs; when *stream* is not a EXT:DRIBBLE-STREAM and *pathname* is non-NIL, creates a new EXT:DRIBBLE-STREAM, dribbling from *stream* to *pathname*, writes a dribble initialization note to *pathname*, and return the EXT:DRIBBLE-STREAM (the second value is the *target* STREAM); otherwise WARN that no appropriate action may be taken. *pathname* may be an open output STREAM or a pathname designator. See above for the sample usage. See also src/dribble.lisp in the **CLISP** source tree.

25.2.13.1. Scripting and DRIBBLE

DRIBBLE works by operating on *TERMINAL-IO*, thus is does **not** work when **CLISP** acts as a script interpreter (see Section 32.6.2, “Scripting with **CLISP**”).

Traditionally, **Common Lisp** implementations set *STANDARD-INPUT*, *STANDARD-OUTPUT*, and *ERROR-OUTPUT* to a SYNONYM-STREAM pointing to *TERMINAL-IO*, and **CLISP** is no exception. Thus changing *TERMINAL-IO* to a dribble stream affects all standard i/o.

On the other hand, when **CLISP** acts as a script interpreter, it adheres to the **UNIX** <stdio.h> conventions, thus *STANDARD-INPUT*, *STANDARD-OUTPUT*, and *ERROR-OUTPUT* are normal FILE-STREAMs, and thus are **not** affected by DRIBBLE (*TERMINAL-IO* - and thus (PRINT ... T) - **is** still affected). The [ANSI CL standard] explicitly permits this behavior by stating

> DRIBBLE is intended primarily for interactive debugging; its effect cannot be relied upon when used in a program.

25.2.14. Function LISP-IMPLEMENTATION-VERSION

LISP-IMPLEMENTATION-VERSION returns the numeric version (like 3.14), and the release date (like "1999-07-21"). When running on the same machine on which **CLISP** was built, it appends the binary build and memory image dump date in universal time (like 3141592654). When running on a different machine, it appends the MACHINE-INSTANCE of the machine on which it was built.

25.2.15. Function EXT:ARGV

This function will return a fresh SIMPLE-VECTOR of STRING command line arguments passed to the runtime, including those already processed by **CLISP**. Use EXT:*ARGS* instead of this function to get the arguments for your program.

Chapter 26. Glossary [CLHS-26]

No notes.

Chapter 27. Appendix [CLHS-a]

No notes.

Chapter 28. X3J13 Issue Index [CLHS-ic]

This is the list of [ANSI CL standard] issues and their current status in **CLISP**, i.e., whether **CLISP** supports code that makes use of the functionality specified by the vote.

X3J13 Issues

&ENVIRONMENT-BINDING-ORDER:FIRST
: yes

ACCESS-ERROR-NAME
: yes

ADJUST-ARRAY-DISPLACEMENT
: yes

ADJUST-ARRAY-FILL-POINTER
: yes

ADJUST-ARRAY-NOT-ADJUSTABLE:IMPLICIT-COPY
: yes

ALLOCATE-INSTANCE:ADD
: yes

ALLOW-LOCAL-INLINE:INLINE-NOTINLINE
: yes

ALLOW-OTHER-KEYS-NIL:PERMIT
: yes

AREF-1D
: yes

ARGUMENT-MISMATCH-ERROR-AGAIN:CONSISTENT
: yes

ARGUMENT-MISMATCH-ERROR-MOON:FIX
: yes

ARGUMENT-MISMATCH-ERROR:MORE-CLARIFICATIONS
yes, except for argument list checking in `CALL-NEXT-METHOD` in compiled code (items 11,12)
ARGUMENTS-UNDERSPECIFIED:SPECIFY
yes
ARRAY-DIMENSION-LIMIT-IMPLICATIONS:ALL-FIXNUM
yes
ARRAY-TYPE-ELEMENT-TYPE-SEMANTICS:UNIFY-UPGRADING
yes
ASSERT-ERROR-TYPE:ERROR
yes
ASSOC-RASSOC-IF-KEY
yes
ASSOC-RASSOC-IF-KEY:YES
yes
BOA-AUX-INITIALIZATION:ERROR-ON-READ
yes
BREAK-ON-WARNINGS-OBSOLETE:REMOVE
yes
BROADCAST-STREAM-RETURN-VALUES:CLARIFY-MINIMALLY
yes
BUTLAST-NEGATIVE:SHOULD-SIGNAL
yes
CHANGE-CLASS-INITARGS:PERMIT
yes
CHAR-NAME-CASE:X3J13-MAR-91
yes
CHARACTER-LOOSE-ENDS:FIX
yes
CHARACTER-PROPOSAL:2
yes
CHARACTER-PROPOSAL:2-1-1
yes
CHARACTER-PROPOSAL:2-1-2
yes
CHARACTER-PROPOSAL:2-2-1
yes
CHARACTER-PROPOSAL:2-3-1
yes

[CHARACTER-PROPOSAL:2-3-2]
    yes
[CHARACTER-PROPOSAL:2-3-3]
    yes
[CHARACTER-PROPOSAL:2-3-4]
    yes
[CHARACTER-PROPOSAL:2-3-5]
    yes
[CHARACTER-PROPOSAL:2-3-6]
    yes
[CHARACTER-PROPOSAL:2-4-1]
    yes
[CHARACTER-PROPOSAL:2-4-2]
    yes
[CHARACTER-PROPOSAL:2-4-3]
    yes
[CHARACTER-PROPOSAL:2-5-2]
    yes
[CHARACTER-PROPOSAL:2-5-6]
    yes
[CHARACTER-PROPOSAL:2-5-7]
    yes
[CHARACTER-PROPOSAL:2-6-1]
    yes
[CHARACTER-PROPOSAL:2-6-2]
    yes
[CHARACTER-PROPOSAL:2-6-3]
    yes
[CHARACTER-PROPOSAL:2-6-5]
    yes
[CHARACTER-VS-CHAR:LESS-INCONSISTENT-SHORT]
    yes
[CLASS-OBJECT-SPECIALIZER:AFFIRM]
    yes
[CLOS-CONDITIONS-AGAIN:ALLOW-SUBSET]
    yes
[CLOS-CONDITIONS:INTEGRATE]
    yes
[CLOS-ERROR-CHECKING-ORDER:NO-APPLICABLE-METHOD-FIRST]
    yes

CLOS-MACRO-COMPILATION:MINIMAL
yes
CLOSE-CONSTRUCTED-STREAM:ARGUMENT-STREAM-ONLY
yes
CLOSED-STREAM-OPERATIONS:ALLOW-INQUIRY
yes
COERCING-SETF-NAME-TO-FUNCTION:ALL-FUNCTION-NAMES
yes
COLON-NUMBER
yes
COMMON-FEATURES:SPECIFY
yes
COMMON-TYPE:REMOVE
yes
COMPILE-ARGUMENT-PROBLEMS-AGAIN:FIX
yes
COMPILE-FILE-HANDLING-OF-TOP-LEVEL-FORMS:CLARIFY
yes
COMPILE-FILE-OUTPUT-FILE-DEFAULTS:INPUT-FILE
yes
COMPILE-FILE-PACKAGE
yes
COMPILE-FILE-PATHNAME-ARGUMENTS:MAKE-CONSISTENT
yes
COMPILE-FILE-SYMBOL-HANDLING:NEW-REQUIRE-CONSISTENCY
yes
COMPILED-FUNCTION-REQUIREMENTS:TIGHTEN
yes
COMPILER-DIAGNOSTICS:USE-HANDLER
no
COMPILER-LET-CONFUSION:ELIMINATE
yes
COMPILER-VERBOSITY:LIKE-LOAD
yes
COMPILER-WARNING-STREAM
yes

COMPLEX-ATAN-BRANCH-CUT:TWEAK
: yes

COMPLEX-ATANH-BOGUS-FORMULA:TWEAK-MORE
: yes

COMPLEX-RATIONAL-RESULT:EXTEND
: yes

COMPUTE-APPLICABLE-METHODS:GENERIC
: yes

CONCATENATE-SEQUENCE:SIGNAL-ERROR
: yes

CONDITION-ACCESSORS-SETFABLE:NO
: yes

CONDITION-RESTARTS:BUGGY
: yes

CONDITION-RESTARTS:PERMIT-ASSOCIATION
: yes

CONDITION-SLOTS:HIDDEN
: yes

CONS-TYPE-SPECIFIER:ADD
: yes

CONSTANT-CIRCULAR-COMPILATION:YES
: yes

CONSTANT-COLLAPSING:GENERALIZE
: yes

CONSTANT-COMPILABLE-TYPES:SPECIFY
: yes

CONSTANT-FUNCTION-COMPILATION:NO
: **CLISP** can dump compiled functions defined in the global lexical environment. Interpreted functions can **not** be dumped; this should not be a problem, because an *interpreted* function in a *compiled* file usually indicate a programmer error (often an extra `QUOTE`).

CONSTANT-MODIFICATION:DISALLOW
: yes

CONSTANTP-DEFINITION:INTENTIONAL
: yes

CONSTANTP-ENVIRONMENT:ADD-ARG
: yes

CONTAGION-ON-NUMERICAL-COMPARISONS:TRANSITIVE
: yes

COPY-SYMBOL-COPY-PLIST:COPY-LIST
: yes

COPY-SYMBOL-PRINT-NAME:EQUAL
: yes

DATA-IO:ADD-SUPPORT
: yes

DATA-TYPES-HIERARCHY-UNDERSPECIFIED
: yes

DEBUGGER-HOOK-VS-BREAK:CLARIFY
: yes

DECLARATION-SCOPE:NO-HOISTING
: yes

DECLARE-ARRAY-TYPE-ELEMENT-REFERENCES:RESTRICTIVE
: yes

DECLARE-FUNCTION-AMBIGUITY:DELETE-FTYPE-ABBREVIATION
: yes

DECLARE-MACROS:FLUSH
: yes

DECLARE-TYPE-FREE:LEXICAL
: yes

DECLS-AND-DOC
: there is no writeup, but all affected operators are fully implemented as specified

DECODE-UNIVERSAL-TIME-DAYLIGHT:LIKE-ENCODE
: yes

DEFCONSTANT-SPECIAL:NO
: yes

DEFGENERIC-DECLARE:ALLOW-MULTIPLE
: yes

DEFINE-COMPILER-MACRO:X3J13-NOV89
: yes

DEFINE-CONDITION-SYNTAX:INCOMPATIBLY-MORE-LIKE-DEFCLASS+EMPHASIZE-READ-ONLY
: yes

DEFINE-METHOD-COMBINATION-BEHAVIOR:CLARIFY
: no

DEFINING-MACROS-NON-TOP-LEVEL:ALLOW
: yes

DEFMACRO-BLOCK-SCOPE:EXCLUDES-BINDINGS
: yes

DEFMACRO-LAMBDA-LIST:TIGHTEN-DESCRIPTION

yes
DEFMETHOD-DECLARATION-SCOPE:CORRESPONDS-TO-BINDINGS
yes
DEFPACKAGE:ADDITION
yes
DEFSTRUCT-CONSTRUCTOR-KEY-MIXTURE:ALLOW-KEY
yes
DEFSTRUCT-CONSTRUCTOR-OPTIONS:EXPLICIT
yes
DEFSTRUCT-CONSTRUCTOR-SLOT-VARIABLES:NOT-BOUND
yes
DEFSTRUCT-COPIER-ARGUMENT-TYPE:RESTRICT
yes
DEFSTRUCT-COPIER:ARGUMENT-TYPE
yes
DEFSTRUCT-DEFAULT-VALUE-EVALUATION:IFF-NEEDED
yes
DEFSTRUCT-INCLUDE-DEFTYPE:EXPLICITLY-UNDEFINED
yes
DEFSTRUCT-PRINT-FUNCTION-AGAIN:X3J13-MAR-93
yes
DEFSTRUCT-PRINT-FUNCTION-INHERITANCE:YES
yes
DEFSTRUCT-REDEFINITION:ERROR
yes
DEFSTRUCT-SLOTS-CONSTRAINTS-NAME:DUPLICATES-ERROR
yes
DEFSTRUCT-SLOTS-CONSTRAINTS-NUMBER
yes
DEFTYPE-DESTRUCTURING:YES
yes
DEFTYPE-KEY:ALLOW
yes
DEFVAR-DOCUMENTATION:UNEVALUATED
yes
DEFVAR-INIT-TIME:NOT-DELAYED
yes
DEFVAR-INITIALIZATION:CONSERVATIVE

yes
DEPRECATION-POSITION:LIMITED
yes
DESCRIBE-INTERACTIVE:NO
yes
DESCRIBE-UNDERSPECIFIED:DESCRIBE-OBJECT
yes
DESTRUCTIVE-OPERATIONS:SPECIFY
yes
DESTRUCTURING-BIND:NEW-MACRO
yes
DISASSEMBLE-SIDE-EFFECT:DO-NOT-INSTALL
yes
DISPLACED-ARRAY-PREDICATE:ADD
yes
DO-SYMBOLS-BLOCK-SCOPE:ENTIRE-FORM
yes
DO-SYMBOLS-DUPLICATES
yes
DOCUMENTATION-FUNCTION-BUGS:FIX
yes
DOCUMENTATION-FUNCTION-TANGLED:REQUIRE-ARGUMENT
yes
DOTIMES-IGNORE:X3J13-MAR91
yes
DOTTED-LIST-ARGUMENTS:CLARIFY
yes
DOTTED-MACRO-FORMS:ALLOW
yes
DRIBBLE-TECHNIQUE
yes
DYNAMIC-EXTENT-FUNCTION:EXTEND
yes
DYNAMIC-EXTENT:NEW-DECLARATION
yes
EQUAL-STRUCTURE:MAYBE-STATUS-QUO
yes
ERROR-TERMINOLOGY-WARNING:MIGHT
yes
EVAL-OTHER:SELF-EVALUATE

yes

EVAL-TOP-LEVEL:LOAD-LIKE-COMPILE-FILE

yes

EVAL-WHEN-NON-TOP-LEVEL:GENERALIZE-EVAL-NEW-KEYWORDS

yes

EVAL-WHEN-OBSOLETE-KEYWORDS:X3J13-MAR-1993

no

EVALHOOK-STEP-CONFUSION:FIX

yes

EVALHOOK-STEP-CONFUSION:X3J13-NOV-89

yes

EXIT-EXTENT-AND-CONDITION-SYSTEM:LIKE-DYNAMIC-BINDINGS

yes

EXIT-EXTENT:MINIMAL

yes, actually implement MEDIUM

EXPT-RATIO:P.211

yes

EXTENSIONS-POSITION:DOCUMENTATION

yes

EXTERNAL-FORMAT-FOR-EVERY-FILE-CONNECTION:MINIMUM

yes

EXTRA-RETURN-VALUES:NO

yes

FILE-OPEN-ERROR:SIGNAL-FILE-ERROR

yes

FIXNUM-NON-PORTABLE:TIGHTEN-DEFINITION

yes

FLET-DECLARATIONS

yes

FLET-DECLARATIONS:ALLOW

yes

FLET-IMPLICIT-BLOCK:YES

yes

FLOAT-UNDERFLOW:ADD-VARIABLES

yes

FLOATING-POINT-CONDITION-NAMES:X3J13-NOV-89

yes

FORMAT-ATSIGN-COLON

yes

FORMAT-COLON-UPARROW-SCOPE

yes

FORMAT-COMMA-INTERVAL

yes

FORMAT-E-EXPONENT-SIGN:FORCE-SIGN

yes

FORMAT-OP-C

yes

FORMAT-PRETTY-PRINT:YES

yes, except that `~F`, `~E`, `~G`, `~$` also bind `*PRINT-BASE*` to 10 and `*PRINT-RADIX*` to `NIL`

FORMAT-STRING-ARGUMENTS:SPECIFY

yes

FUNCTION-CALL-EVALUATION-ORDER:MORE-UNSPECIFIED

yes

FUNCTION-COMPOSITION:JAN89-X3J13

yes

FUNCTION-DEFINITION:JAN89-X3J13

yes

FUNCTION-NAME:LARGE

yes

FUNCTION-TYPE

yes

FUNCTION-TYPE-ARGUMENT-TYPE-SEMANTICS:RESTRICTIVE

yes

FUNCTION-TYPE-KEY-NAME:SPECIFY-KEYWORD

yes

FUNCTION-TYPE-REST-LIST-ELEMENT:USE-ACTUAL-ARGUMENT-TYPE

yes

FUNCTION-TYPE:X3J13-MARCH-88

yes

GENERALIZE-PRETTY-PRINTER:UNIFY

no

GENERIC-FLET-POORLY-DESIGNED:DELETE

yes

GENSYM-NAME-STICKINESS:LIKE-TEFLON

yes

GENTEMP-BAD-IDEA:DEPRECATE
: yes

GET-MACRO-CHARACTER-READTABLE:NIL-STANDARD
: yes

GET-SETF-METHOD-ENVIRONMENT:ADD-ARG
: yes

HASH-TABLE-ACCESS:X3J13-MAR-89
: yes

HASH-TABLE-KEY-MODIFICATION:SPECIFY
: yes

HASH-TABLE-PACKAGE-GENERATORS:ADD-WITH-WRAPPER
: yes

HASH-TABLE-REHASH-SIZE-INTEGER
: yes

HASH-TABLE-SIZE:INTENDED-ENTRIES
: yes

HASH-TABLE-TESTS:ADD-EQUALP
: yes

IEEE-ATAN-BRANCH-CUT:SPLIT
: yes

IGNORE-USE-TERMINOLOGY:VALUE-ONLY
: yes

IMPORT-SETF-SYMBOL-PACKAGE
: yes

IN-PACKAGE-FUNCTIONALITY:MAR89-X3J13
: yes

IN-SYNTAX:MINIMAL
: yes

INITIALIZATION-FUNCTION-KEYWORD-CHECKING
: yes

ISO-COMPATIBILITY:ADD-SUBSTRATE
: yes

JUN90-TRIVIAL-ISSUES:11
: yes

JUN90-TRIVIAL-ISSUES:14
: yes

JUN90-TRIVIAL-ISSUES:24
: yes

JUN90-TRIVIAL-ISSUES:25
: yes

JUN90-TRIVIAL-ISSUES:27
yes for `THE`, no for `APPLY` (spec not clear)
JUN90-TRIVIAL-ISSUES:3
yes
JUN90-TRIVIAL-ISSUES:4
yes
JUN90-TRIVIAL-ISSUES:5
yes
JUN90-TRIVIAL-ISSUES:9
yes
KEYWORD-ARGUMENT-NAME-PACKAGE:ANY
yes
LAST-N
yes
LCM-NO-ARGUMENTS:1
yes
LEXICAL-CONSTRUCT-GLOBAL-DEFINITION:UNDEFINED
yes
LISP-PACKAGE-NAME:COMMON-LISP
yes
LISP-SYMBOL-REDEFINITION-AGAIN:MORE-FIXES
yes
LISP-SYMBOL-REDEFINITION:MAR89-X3J13
yes
LOAD-OBJECTS:MAKE-LOAD-FORM
yes
LOAD-TIME-EVAL:R2-NEW-SPECIAL-FORM**
obsolete
LOAD-TIME-EVAL:R3-NEW-SPECIAL-FORM**
yes
LOAD-TRUENAME:NEW-PATHNAME-VARIABLES
yes
LOCALLY-TOP-LEVEL:SPECIAL-FORM
yes
LOOP-AND-DISCREPANCY:NO-REITERATION
yes
LOOP-FOR-AS-ON-TYPO:FIX-TYPO
yes
LOOP-INITFORM-ENVIRONMENT:PARTIAL-INTERLEAVING-VAGUE
no

LOOP-MISCELLANEOUS-REPAIRS:FIX
: yes

LOOP-NAMED-BLOCK-NIL:OVERRIDE
: yes

LOOP-PRESENT-SYMBOLS-TYPO:FLUSH-WRONG-WORDS
: yes

LOOP-SYNTAX-OVERHAUL:REPAIR
: yes

MACRO-AS-FUNCTION:DISALLOW
: yes

MACRO-DECLARATIONS:MAKE-EXPLICIT
: yes

MACRO-ENVIRONMENT-EXTENT:DYNAMIC
: yes

MACRO-FUNCTION-ENVIRONMENT
: obsolete

MACRO-FUNCTION-ENVIRONMENT:YES
: yes

MACRO-SUBFORMS-TOP-LEVEL-P:ADD-CONSTRAINTS
: no

MACROEXPAND-HOOK-DEFAULT:EXPLICITLY-VAGUE
: yes

MACROEXPAND-HOOK-INITIAL-VALUE:IMPLEMENTATION-DEPENDENT
: yes

MACROEXPAND-RETURN-VALUE:TRUE
: yes

MAKE-LOAD-FORM-CONFUSION:REWRITE
: yes

MAKE-LOAD-FORM-SAVING-SLOTS:NO-INITFORMS
: yes

MAKE-PACKAGE-USE-DEFAULT:IMPLEMENTATION-DEPENDENT
: yes

MAP-INTO:ADD-FUNCTION
: yes

MAPPING-DESTRUCTIVE-INTERACTION:EXPLICITLY-VAGUE
: yes

METACLASS-OF-SYSTEM-CLASS:UNSPECIFIED
: yes

METHOD-COMBINATION-ARGUMENTS:CLARIFY
no
METHOD-INITFORM:FORBID-CALL-NEXT-METHOD
no
MUFFLE-WARNING-CONDITION-ARGUMENT
yes
MULTIPLE-VALUE-SETQ-ORDER:LIKE-SETF-OF-VALUES
yes
MULTIPLE-VALUES-LIMIT-ON-VARIABLES:UNDEFINED
yes
NINTERSECTION-DESTRUCTION
yes
NINTERSECTION-DESTRUCTION:REVERT
yes
NOT-AND-NULL-RETURN-VALUE:X3J13-MAR-93
yes
NTH-VALUE:ADD
yes
OPTIMIZE-DEBUG-INFO:NEW-QUALITY
yes
PACKAGE-CLUTTER:REDUCE
yes
PACKAGE-DELETION:NEW-FUNCTION
yes
PACKAGE-FUNCTION-CONSISTENCY:MORE-PERMISSIVE
yes
PARSE-ERROR-STREAM:SPLIT-TYPES
yes
PATHNAME-COMPONENT-CASE:KEYWORD-ARGUMENT
yes
PATHNAME-COMPONENT-VALUE:SPECIFY
no
PATHNAME-HOST-PARSING:RECOGNIZE-LOGICAL-HOST-NAMES
yes when `CUSTOM:*PARSE-NAMESTRING-ANSI*` is non-`NIL`
PATHNAME-LOGICAL:ADD
yes
PATHNAME-PRINT-READ:SHARPSIGN-P
yes
PATHNAME-STREAM
yes

PATHNAME-STREAM:FILES-OR-SYNONYM
: yes

PATHNAME-SUBDIRECTORY-LIST:NEW-REPRESENTATION
: yes

PATHNAME-SYMBOL
: yes when `CUSTOM:*PARSE-NAMESTRING-ANSI*` is non-`NIL`

PATHNAME-SYNTAX-ERROR-TIME:EXPLICITLY-VAGUE
: yes

PATHNAME-UNSPECIFIC-COMPONENT:NEW-TOKEN
: yes

PATHNAME-WILD:NEW-FUNCTIONS
: yes

PEEK-CHAR-READ-CHAR-ECHO:FIRST-READ-CHAR
: yes

PLIST-DUPLICATES:ALLOW
: yes

PRETTY-PRINT-INTERFACE
: yes

PRINC-READABLY:X3J13-DEC-91
: yes

PRINT-CASE-BEHAVIOR:CLARIFY
: yes

PRINT-CASE-PRINT-ESCAPE-INTERACTION:VERTICAL-BAR-RULE-NO-UPCASE
: yes

PRINT-CIRCLE-SHARED:RESPECT-PRINT-CIRCLE
: yes

PRINT-CIRCLE-STRUCTURE:USER-FUNCTIONS-WORK
: yes

PRINT-READABLY-BEHAVIOR:CLARIFY
: yes

PRINTER-WHITESPACE:JUST-ONE-SPACE
: yes

PROCLAIM-ETC-IN-COMPILE-FILE:NEW-MACRO
: yes

PUSH-EVALUATION-ORDER:FIRST-ITEM
: yes

PUSH-EVALUATION-ORDER:ITEM-FIRST
: yes

PUSHNEW-STORE-REQUIRED:UNSPECIFIED
: yes

QUOTE-SEMANTICS:NO-COPYING
: yes

RANGE-OF-COUNT-KEYWORD:NIL-OR-INTEGER
: yes, when `CUSTOM:*SEQUENCE-COUNT-ANSI*` is non-`NIL`; otherwise negative `:COUNT` values are not allowed.

RANGE-OF-START-AND-END-PARAMETERS:INTEGER-AND-INTEGER-NIL
: yes

READ-AND-WRITE-BYTES:NEW-FUNCTIONS
: yes

READ-CASE-SENSITIVITY:READTABLE-KEYWORDS
: yes

READ-MODIFY-WRITE-EVALUATION-ORDER:DELAYED-ACCESS-STORES
: no

READ-SUPPRESS-CONFUSING:GENERALIZE
: yes, except that `READ-DELIMITED-LIST` still constructs a `LIST`

READER-ERROR:NEW-TYPE
: yes

REAL-NUMBER-TYPE:X3J13-MAR-89
: yes

RECURSIVE-DEFTYPE:EXPLICITLY-VAGUE
: yes

REDUCE-ARGUMENT-EXTRACTION
: yes

REMF-DESTRUCTION-UNSPECIFIED:X3J13-MAR-89
: yes

REQUIRE-PATHNAME-DEFAULTS-AGAIN:X3J13-DEC-91
: yes

REQUIRE-PATHNAME-DEFAULTS-YET-AGAIN:RESTORE-ARGUMENT
: yes

REQUIRE-PATHNAME-DEFAULTS:ELIMINATE
: superseded by REQUIRE-PATHNAME-DEFAULTS-AGAIN:X3J13-DEC-91

REST-LIST-ALLOCATION:MAY-SHARE
: yes

RESULT-LISTS-SHARED:SPECIFY
: yes

RETURN-VALUES-UNSPECIFIED:SPECIFY
: yes

ROOM-DEFAULT-ARGUMENT:NEW-VALUE
yes
SELF-MODIFYING-CODE:FORBID
yes
SEQUENCE-TYPE-LENGTH:MUST-MATCH
yes
SETF-APPLY-EXPANSION:IGNORE-EXPANDER
no
SETF-FIND-CLASS:ALLOW-NIL
yes
SETF-FUNCTIONS-AGAIN:MINIMAL-CHANGES
yes
SETF-GET-DEFAULT:EVALUATED-BUT-IGNORED
yes
SETF-MACRO-EXPANSION:LAST
yes
SETF-METHOD-VS-SETF-METHOD:RENAME-OLD-TERMS
yes
SETF-MULTIPLE-STORE-VARIABLES:ALLOW
yes
SETF-OF-APPLY:ONLY-AREF-AND-FRIENDS
yes
SETF-OF-VALUES:ADD
yes
SETF-SUB-METHODS:DELAYED-ACCESS-STORES
yes
SHADOW-ALREADY-PRESENT
yes
SHADOW-ALREADY-PRESENT:WORKS
yes
SHARP-COMMA-CONFUSION:REMOVE
no
SHARP-O-FOOBAR:CONSEQUENCES-UNDEFINED
yes
SHARP-STAR-DELIMITER:NORMAL-DELIMITER
yes
SHARPSIGN-PLUS-MINUS-PACKAGE:KEYWORD
yes
SLOT-MISSING-VALUES:SPECIFY
yes
SLOT-VALUE-METACLASSES:LESS-MINIMAL

yes
SPECIAL-FORM-P-MISNOMER:RENAME
yes
SPECIAL-TYPE-SHADOWING:CLARIFY
yes
STANDARD-INPUT-INITIAL-BINDING:DEFINED-CONTRACTS
yes
STANDARD-REPERTOIRE-GRATUITOUS:RENAME
yes
STEP-ENVIRONMENT:CURRENT
yes
STEP-MINIMAL:PERMIT-PROGN
yes
STREAM-ACCESS:ADD-TYPES-ACCESSORS
yes
STREAM-CAPABILITIES:INTERACTIVE-STREAM-P
yes
STRING-COERCION:MAKE-CONSISTENT
yes
STRING-OUTPUT-STREAM-BASHING:UNDEFINED
yes
STRUCTURE-READ-PRINT-SYNTAX:KEYWORDS
yes
SUBSEQ-OUT-OF-BOUNDS
yes
SUBSEQ-OUT-OF-BOUNDS:IS-AN-ERROR
yes
SUBSETTING-POSITION:NONE
yes
SUBTYPEP-ENVIRONMENT:ADD-ARG
yes
SUBTYPEP-TOO-VAGUE:CLARIFY-MORE
yes
SXHASH-DEFINITION:SIMILAR-FOR-SXHASH
yes
SYMBOL-MACROLET-DECLARE:ALLOW
yes
SYMBOL-MACROLET-SEMANTICS:SPECIAL-FORM
yes
SYMBOL-MACROLET-TYPE-DECLARATION:NO
yes

[SYMBOL-MACROS-AND-PROCLAIMED-SPECIALS:SIGNALS-AN-ERROR]
  yes
[SYMBOL-PRINT-ESCAPE-BEHAVIOR:CLARIFY]
  yes
[SYNTACTIC-ENVIRONMENT-ACCESS:RETRACTED-MAR91]
  yes
[TAGBODY-TAG-EXPANSION:NO]
  yes
[TAILP-NIL:T]
  yes
[TEST-NOT-IF-NOT:FLUSH-ALL]
  yes, but no warning
[THE-AMBIGUITY:FOR-DECLARATION]
  yes
[THE-VALUES:RETURN-NUMBER-RECEIVED]
  yes
[TIME-ZONE-NON-INTEGER:ALLOW]
  yes
[TYPE-DECLARATION-ABBREVIATION:ALLOW-ALL]
  no
[TYPE-OF-AND-PREDEFINED-CLASSES:TYPE-OF-HANDLES-FLOATS]
  yes
[TYPE-OF-AND-PREDEFINED-CLASSES:UNIFY-AND-EXTEND]
  yes
[TYPE-OF-UNDERCONSTRAINED:ADD-CONSTRAINTS]
  yes
[TYPE-SPECIFIER-ABBREVIATION:X3J13-JUN90-GUESS]
  yes
[UNDEFINED-VARIABLES-AND-FUNCTIONS:COMPROMISE]
  yes
[UNINITIALIZED-ELEMENTS:CONSEQUENCES-UNDEFINED]
  yes, could add error checking
[UNREAD-CHAR-AFTER-PEEK-CHAR:DONT-ALLOW]
  yes
[UNSOLICITED-MESSAGES:NOT-TO-SYSTEM-USER-STREAMS]
  yes
[VARIABLE-LIST-ASYMMETRY:SYMMETRIZE]
  yes
[WITH-ADDED-METHODS:DELETE]

yes

WITH-COMPILATION-UNIT:NEW-MACRO

yes

WITH-OPEN-FILE-DOES-NOT-EXIST:STREAM-IS-NIL

yes

WITH-OPEN-FILE-SETQ:EXPLICITLY-VAGUE

yes

WITH-OPEN-FILE-STREAM-EXTENT:DYNAMIC-EXTENT

yes

WITH-OUTPUT-TO-STRING-APPEND-STYLE:VECTOR-PUSH-EXTEND

yes

WITH-STANDARD-IO-SYNTAX-READTABLE:X3J13-MAR-91

yes

Part II. Common Portable Extensions

Table of Contents

Chapter 29. Meta-Object Protocol

Adapted from chapters 5 and 6 of [AMOP]

Table of Contents

29.1. Introduction

The **CLOS** specification ([ANSI CL standard] Chanpter 7) describes the standard Programmer Interface for the **Common Lisp** Object System (**CLOS**). This document extends that specification by defining a metaobject protocol for **CLOS** - that is, a description of **CLOS** itself as an extensible **CLOS** program. In this description, the fundamental elements of **CLOS** programs (classes, slot definitions, generic functions, methods, specializers and method combinations) are represented by first-class objects. The behavior of **CLOS** is provided by these objects, or, more precisely, by methods specialized to the classes of these objects.

Because these objects represent pieces of **CLOS** programs, and because their behavior provides the behavior of the **CLOS** language itself, they are considered meta-level objects or metaobjects. The protocol followed by the metaobjects to provide the behavior of **CLOS** is called the **CLOS** “Metaobject Protocol” (MOP).

29.1.1. Notation

The description of functions follows the same form as used in the **CLOS** specification. The description of generic functions is similar to that in the **CLOS** specification, but some minor changes have been made in the way methods are presented.

The following is an example of the format for the syntax description of a generic function:

`(gf1 x y &OPTIONAL v &KEY k)`

This description indicates that `gf1` is a generic function with two required parameters, *x* and *y*, an optional parameter *v* and a keyword parameter *k*.

The description of a generic function includes a description of its behavior. This provides the general behavior, or protocol of the generic function. All methods defined on the generic function, both portable and specified, must have behavior consistent with this description.

Every generic function described here is an instance of the class `STANDARD-GENERIC-FUNCTION` and uses the `STANDARD` method combination.

The description of a generic function also includes descriptions of the specified methods for that generic function. In the description of these methods, a ***method signature*** is used to describe the parameters and parameter specializers of each method. The following is an example of the format for a method signature:

`(gf1 (x CLASS) y &OPTIONAL v &KEY k)`

This signature indicates that this primary method on the generic function `gf1` has two required parameters, named *x* and *y*. In addition, there is an optional parameter *v* and a keyword parameter *k*. This signature also indicates that the method's parameter specializers are the classes `CLASS` and `T`.

The description of each method includes a description of the behavior particular to that method.

An abbreviated syntax is used when referring to a method defined elsewhere in the document. This abbreviated syntax includes the name of the generic function, the qualifiers, and the parameter specializers. A reference to the method with the signature shown above is written as: `gf1 (CLASS T)`.

29.1.2. Package

The package exporting the Meta-Object Protocol symbols is unspecified.

Implementation dependent: only in CLISP

The symbols specified by the Meta-Object Protocol are exported from the package **“CLOS”** and `EXT:RE-EXPORT`ed from the package **“EXT”**.

The package exporting the Meta-Object Protocol symbols is different in other implementations: In **SBCL** it is the package **“SB-MOP”**; in **OpenMCL** it is the package **“OPENMCL-MOP”**.

29.2. Overview

29.2.1. Metaobjects

For each kind of program element there is a corresponding ***basic metaobject class*** . These are the classes: `CLASS`, `CLOS:SLOT-DEFINITION`, `GENERIC-FUNCTION`, `METHOD` and `METHOD-COMBINATION`. A ***metaobject class*** is a subclass of exactly one of these classes. The results are undefined if an attempt is made to define a `CLASS` that is a subclass of more than one basic metaobject class. A ***metaobject*** is an instance of a metaobject class.

Each metaobject represents one program element. Associated with each metaobject is the information required to serve its role. This includes information that might be provided directly in a user interface macro such as `DEFCLASS` or `DEFMETHOD`. It also includes information computed indirectly from other metaobjects such as that computed from class inheritance or the full set of methods associated with a generic function.

Much of the information associated with a metaobject is in the form of connections to other metaobjects. This interconnection means that the role of a metaobject is always based on that of other metaobjects. As an introduction to this interconnected structure, this section presents a partial enumeration of the kinds of information associated with each kind of metaobject. More detailed information is presented later.

29.2.1.1. Classes

A ***class metaobject*** determines the structure and the default behavior of its instances. The following information is associated with class metaobjects:

- The name, if there is one, is available as an object.
- The direct subclasses, direct superclasses and class precedence list are available as lists of class metaobjects.
- The slots defined directly in the class are available as a list of direct slot definition metaobjects. The slots which are accessible in instances of the class are available as a list of effective slot definition metaobjects.
- The methods which use the class as a specializer, and the generic functions associated with those methods are available as lists of method and generic function metaobjects respectively.
- The documentation is available as a `STRING` or `NIL`.

See also Section 29.3, "Classes"

29.2.1.2. Slot Definitions

A ***slot definition metaobject*** contains information about the definition of a slot. There are two kinds of slot definition metaobjects: A ***direct*** slot definition metaobject is used to represent the direct definition of a slot in a class. This corresponds roughly to the slot specifiers found in `DEFCLASS` forms. An ***effective*** slot definition metaobject is used to represent information, including inherited information, about a slot which is accessible in instances of a particular class.

Associated with each class metaobject is a list of direct slot definition metaobjects representing the slots defined directly in the class. Also associated with each class metaobject is a list of effective slot definition metaobjects representing the set of slots accessible in instances of that class.

The following information is associated with both direct and effective slot definitions metaobjects:

- The name, allocation, and type are available as forms that could appear in a `DEFCLASS` form.
- The initialization form, if there is one, is available as a form that could appear in a `DEFCLASS` form. The initialization form together with its lexical environment is available as a function of no arguments which, when called, returns the result of evaluating the

initialization form in its lexical environment. This is called the ***initfunction*** of the slot.
- The slot filling initialization arguments are available as a list of symbols.
- The documentation is available as a `STRING` or `NIL`.

Certain other information is only associated with direct slot definition metaobjects. This information applies only to the direct definition of the slot in the class (it is not inherited).

- The function names of those generic functions for which there are automatically generated reader and writer methods. This information is available as lists of function names. Any accessors specified in the `DEFCLASS` form are broken down into their equivalent readers and writers in the direct slot definition.

Information, including inherited information, which applies to the definition of a slot in a particular class in which it is accessible is associated only with effective slot definition metaobjects.

- For certain slots, the location of the slot in instances of the class is available.

See also Section 29.4, “Slot Definitions”

29.2.1.3. Generic Functions

A ***generic function metaobject*** contains information about a generic function over and above the information associated with each of the generic function's methods.

- The name is available as a function name.
- The methods associated with the generic function are available as a list of method metaobjects.
- The default class for this generic function's method metaobjects is available as a class metaobject.
- The lambda list is available as a `LIST`.
- The method combination is available as a method combination metaobject.

- The argument precedence order is available as a permutation of those symbols from the lambda list which name the required arguments of the generic function.
- The “declarations” are available as a list of declaration specifiers.

Note

There is a slight misnomer in the naming of functions and options in this document: Where the term “declaration” is used, actually a declaration specifier is meant.

- The documentation is available as a `STRING` or `NIL`.

See also Section 29.5, “Generic Functions”

29.2.1.4. Methods

A ***method metaobject*** contains information about a specific `METHOD`.

- The qualifiers are available as a `LIST` of of non-`NIL` atoms.
- The lambda list is available as a `LIST`.
- The specializers are available as a list of specializer metaobjects.
- The function is available as a `FUNCTION`. This function can be applied to arguments and a list of next methods using `APPLY` or `FUNCALL`.
- When the method is associated with a generic function, that generic function metaobject is available. A method can be associated with at most one generic function at a time.
- The documentation is available as a `STRING` or `NIL`.

See also Section 29.6, “Methods”

29.2.1.5. Specializers

A ***specializer metaobject*** represents the specializers of a `METHOD`. class metaobjects are themselves specializer metaobjects. A special kind of specializer metaobject is used for `EQL` specializers.

See also Section 29.8, "Specializers"

29.2.1.6. Method Combinations

A ***method combination metaobject*** represents the information about the method combination being used by a generic function.

> **Note**
>
> This document does not specify the structure of method combination metaobjects.

See also Section 29.9, "Method Combinations"

29.2.2. Inheritance Structure of Metaobject Classes

Figure 29.1. Inheritance structure of metaobject classes

Inheritance structure of metaobject classes

The inheritance structure of the specified metaobject classes is shown in Table 29.1, "Direct Superclass Relationships Among The Specified

Metaobject Classes". The class of every class shown is STANDARD-CLASS except for the classes T and FUNCTION, which are instances of the class BUILT-IN-CLASS, and the classes GENERIC-FUNCTION and STANDARD-GENERIC-FUNCTION, which are instances of the class CLOS:FUNCALLABLE-STANDARD-CLASS.

Table 29.1. Direct Superclass Relationships Among The Specified Metaobject Classes

Metaobject Class	Abstract	Subclassable	Direct Superclasses
STANDARD-OBJECT	no	yes	(T)
CLOS:FUNCALLABLE-STANDARD-OBJECT	no	yes	(STANDARD-OBJECT FUNCTION)
CLOS:METAOBJECT	yes	no	(STANDARD-OBJECT)
GENERIC-FUNCTION	yes	no	(CLOS:METAOBJECT CLOS:FUNCALLABLE-STANDARD-OBJECT)
STANDARD-GENERIC-FUNCTION	no	yes	(GENERIC-FUNCTION)
METHOD	yes	no	(CLOS:METAOBJECT)
STANDARD-METHOD	no	yes	(METHOD)
CLOS:STANDARD-ACCESSOR-METHOD	yes	no	(STANDARD-METHOD)
CLOS:STANDARD-READER-METHOD	no	yes	(CLOS:STANDARD-ACCESSOR-METHOD)
CLOS:STANDARD-WRITER-METHOD	no	yes	(CLOS:STANDARD-ACCESSOR-METHOD)
METHOD-COMBINATION	yes	no	(CLOS:METAOBJECT)
CLOS:SLOT-DEFINITION	yes	no	(CLOS:METAOBJECT)
CLOS:DIRECT-SLOT-DEFINITION	yes	no	(CLOS:SLOT-DEFINITION)
CLOS:EFFECTIVE-SLOT-DEFINITION	yes	no	(CLOS:SLOT-DEFINITION)
CLOS:STANDARD-SLOT-DEFINITION	yes	no	(CLOS:SLOT-DEFINITION)

Metaobject Class	Abstract	Subclassable	Direct Superclasses
CLOS:STANDARD-DIRECT-SLOT-DEFINITION	no	yes	(CLOS:STANDARD-SLOT-DEFINITION CLOS:DIRECT-SLOT-DEFINITION)
CLOS:STANDARD-EFFECTIVE-SLOT-DEFINITION	no	yes	(CLOS:STANDARD-SLOT-DEFINITION CLOS:EFFECTIVE-SLOT-DEFINITION)
CLOS:SPECIALIZER	yes	no	(CLOS:METAOBJECT)
CLOS:EQL-SPECIALIZER	no	no	(CLOS:SPECIALIZER)
CLASS	yes	no	(CLOS:SPECIALIZER)
BUILT-IN-CLASS	no	no	(CLASS)
CLOS:FORWARD-REFERENCED-CLASS	no	no	(CLASS)
STANDARD-CLASS	no	yes	(CLASS)
CLOS:FUNCALLABLE-STANDARD-CLASS	no	yes	(CLASS)

Each class with a “yes” in the “Abstract” column is an *abstract class* and is not intended to be instantiated. The results are undefined if an attempt is made to make an instance of one of these classes with MAKE-INSTANCE.

Each class with a “yes” in the “Subclassable” column can be used as direct superclass for portable programs. It is not meaningful to subclass a class that has a “no” in this column.

Implementation dependent: only in CLISP

The class METHOD is also subclassable: It is possible to create subclasses of METHOD that do not inherit from STANDARD-METHOD.

Implementation dependent: only in CLISP and some other implementations

The class CLOS:FUNCALLABLE-STANDARD-OBJECT's class precedence list contains FUNCTION before STANDARD-OBJECT, not after STANDARD-OBJECT. This is the most transparent way to realize the [ANSI CL standard] requirement (see the [ANSI CL standard] section 4.2.2 "Type Relationships") that GENERIC-FUNCTION's class precedence list contains FUNCTION before STANDARD-OBJECT.

The classes STANDARD-CLASS, CLOS:STANDARD-DIRECT-SLOT-DEFINITION, CLOS:STANDARD-EFFECTIVE-SLOT-DEFINITION, STANDARD-METHOD, CLOS:STANDARD-READER-METHOD, CLOS:STANDARD-WRITER-METHOD and STANDARD-GENERIC-FUNCTION are called ***standard metaobject*** classes. For each kind of metaobject, this is the class the user interface macros presented in the **CLOS** use by default. These are also the classes on which user specializations are normally based.

The classes BUILT-IN-CLASS, CLOS:FUNCALLABLE-STANDARD-CLASS and CLOS:FORWARD-REFERENCED-CLASS are special-purpose class metaobject classes. Built-in classes are instances of the class BUILT-IN-CLASS. The class CLOS:FUNCALLABLE-STANDARD-CLASS provides a special kind of instances described in Section 29.10.2, "Funcallable Instances". When the definition of a class references another class which has not yet been defined, an instance of CLOS:FORWARD-REFERENCED-CLASS is used as a stand-in until the class is actually defined.

Implementation of class CLOS:FORWARD-REFERENCED-CLASS in CLISP

The class CLOS:FORWARD-REFERENCED-CLASS is implemented in a way that fixes several flaws in the [AMOP] specification.

It is not a subclass of CLASS and CLOS:SPECIALIZER, just a subclass of CLOS:METAOBJECT, because forward references to classes are not classes and cannot be used as specializers of methods. An [AMOP] compatibility mode is provided, however, if you set the variable ***CUSTOM:*FORWARD-REFERENCED-CLASS-MISDESIGN**** to T. In this mode, CLOS:FORWARD-REFERENCED-CLASS is formally a subclass of CLASS and CLOS:SPECIALIZER, but the behaviour of CLOS:FORWARD-REFERENCED-CLASS instances is the same.

The [AMOP] says that the first argument of CLOS:ENSURE-CLASS-USING-CLASS can be a CLOS:FORWARD-REFERENCED-CLASS. But from the description of CLOS:ENSURE-CLASS, it is clear that it can only be a class returned by FIND-CLASS, and [ANSI CL standard] FIND-CLASS cannot return a CLOS:FORWARD-REFERENCED-CLASS.

The [AMOP] says that CLOS:ENSURE-CLASS-USING-CLASS creates a CLOS:FORWARD-REFERENCED-CLASS for not-yet-defined class symbols among the direct-superclasses list. But this leads to many CLOS:FORWARD-REFERENCED-CLASS with the same name (since they cannot be stored and retrieved through FIND-CLASS), and since CHANGE-CLASS preserves the EQ-ness, after the class is defined, we have many class objects with the same name.

In the direct-superclasses list of non-finalized classes, CLOS:FORWARD-REFERENCED-CLASS instances can occur, denoting classes that have not yet been defined. When or after such a class gets defined, the CLOS:FORWARD-REFERENCED-CLASS instance is replaced with the real class. **CLISP** uses simple object replacement, not CHANGE-CLASS, in this process.

The class STANDARD-OBJECT is the *default direct superclass* of the class STANDARD-CLASS. When an instance of the class STANDARD-CLASS is created, and no direct superclasses are explicitly specified, it defaults to the class STANDARD-OBJECT. In this way, any behavior associated with the class STANDARD-OBJECT will be inherited, directly or indirectly, by all instances of the class STANDARD-CLASS. A subclass of STANDARD-

CLASS may have a different class as its default direct superclass, but that class must be a subclass of the class STANDARD-OBJECT.

The same is true for CLOS:FUNCALLABLE-STANDARD-CLASS and CLOS:FUNCALLABLE-STANDARD-OBJECT.

The class CLOS:SPECIALIZER captures only the most basic behavior of method specializers, and is not itself intended to be instantiated. The class CLASS is a direct subclass of CLOS:SPECIALIZER reflecting the property that classes by themselves can be used as method specializers. The class CLOS:EQL-SPECIALIZER is used for EQL specializers.

29.2.2.1. Implementation and User Specialization

The purpose of the Metaobject Protocol is to provide users with a powerful mechanism for extending and customizing the basic behavior of the **CLOS**. As an object-oriented description of the basic **CLOS** behavior, the Metaobject Protocol makes it possible to create these extensions by defining specialized subclasses of existing metaobject classes.

The Metaobject Protocol provides this capability without interfering with the implementor's ability to develop high-performance implementations. This balance between user extensibility and implementor freedom is mediated by placing explicit restrictions on each. Some of these restrictions are general---they apply to the entire class graph and the applicability of all methods. These are presented in this section.

The following additional terminology is used to present these restrictions:

- Metaobjects are divided into three categories. Those defined in this document are called ***specified***; those defined by an implementation but not mentioned in this document are called ***implementation-specific***; and those defined by a portable program are called ***portable***.
- A class *i* is ***interposed*** between two other classes k_1 and k_2 if and only if there is some path, following direct superclasses, from the class k_1 to the class k_2 which includes *i*.

- A method is ***specialized*** to a class if and only if that class is in the list of specializers associated with the method; and the method is in the list of methods associated with some generic function.
- In a given implementation, a specified method is said to have been ***promoted*** if and only if the specializers of the method, x_1 ... x_n, are defined in this specification as the classes k_1 ... k_n, but in the implementation, one or more of the specializers x_l, is a superclass of the class given in the specification k_l.

- For a given generic function and set of arguments, a method k_2 ***extends*** a method k_1 if and only if:

  i. k_1 and k_2 are both associated with the given generic function
  ii. k_1 and k_2 are both applicable to the given arguments,
  iii. the specializers and qualifiers of the methods are such that when the generic function is called, k_2 is executed before k_1,
  iv. k_1 will be executed if and only if `CALL-NEXT-METHOD` is invoked from within the body of k_2 and
  v. `CALL-NEXT-METHOD` is invoked from within the body of k_2, thereby causing k_1 to be executed.

- For a given generic function and set of arguments, a method k_2 ***overrides*** a method k_1 if and only if conditions *i* through *iv* above hold and, instead of *v*,

  vi. `CALL-NEXT-METHOD` is not invoked from within the body of k_2, thereby preventing k_1 from being executed.

29.2.2.1.1. Restrictions on Portable Programs

Portable programs are allowed to define subclasses of specified classes, and are permitted to define methods on specified generic functions, with the following restrictions:

- Portable programs must not redefine any specified classes, generic functions, methods or method combinations. Any method defined by a portable program on a specified generic function must have at least one specializer that is neither a specified class nor an `EQL` specializer whose associated value is an instance of a specified class.

- Portable programs may define methods that extend specified methods unless the description of the specified method explicitly prohibits this. Unless there is a specific statement to the contrary, these extending methods must return whatever value was returned by the call to `CALL-NEXT-METHOD`.

- Portable programs may define methods that override specified methods only when the description of the specified method explicitly allows this. Typically, when a method is allowed to be overridden, a small number of related methods will need to be overridden as well.

  An example of this is the specified methods on the generic functions `CLOS:ADD-DEPENDENT`, `CLOS:REMOVE-DEPENDENT` and `CLOS:MAP-DEPENDENTS`. Overriding a specified method on one of these generic functions requires that the corresponding method on the other two generic functions be overridden as well.

- Portable methods on specified generic functions specialized to portable metaobject classes must be defined before any instances of those classes (or any subclasses) are created, either directly or indirectly by a call to `MAKE-INSTANCE`. Methods can be defined after instances are created by `ALLOCATE-INSTANCE` however. Portable metaobject classes cannot be redefined.

Note

The purpose of this last restriction is to permit implementations to provide performance optimizations by analyzing, at the time the first instance of a metaobject class is initialized, what portable methods will be applicable to it. This can make it possible to optimize calls to those specified generic functions which would have no applicable portable methods.

Implementation dependent: only in CLISP

When a metaobject class is redefined, **CLISP** issues a `WARNING` that the redefinition has no effect. To avoid this warning, place all metaobject class definitions in a separate file, compile it in a *separate* session (because `DEFCLASS` in **CLISP** is evaluated at compile time too; see Section 29.2.3.2, "Compile-file Processing of Specific User Interface Macros"), and then `LOAD` it only *once* per session.

The results are undefined if any of these restrictions are violated.

Note

The specification technology used in this document needs further development. The concepts of object-oriented protocols and subclass specialization are intuitively familiar to programmers of object-oriented systems; the protocols presented here fit quite naturally into this framework. Nonetheless, in preparing this document, we have found it difficult to give specification-quality descriptions of the protocols in a way that makes it clear what extensions users can and cannot write. Object-oriented protocol specification is inherently about specifying leeway, and this seems difficult using current technology.

29.2.2.1.2. Restrictions on Implementations

Implementations are allowed latitude to modify the structure of specified classes and methods. This includes: the interposition of implementation-specific classes; the promotion of specified methods; and the consolidation of two or more specified methods into a single method specialized to interposed classes.

Any such modifications are permitted only so long as for any portable class k that is a subclass of one or more specified classes k_1 ... k_n, the following conditions are met:

- In the actual class precedence list of k, the classes k_1 ... k_n must appear in the same order as they would have if no implementation-specific modifications had been made.
- The method applicability of any specified generic function must be the same in terms of behavior as it would have been had no implementation-specific changes been made. This includes specified generic functions that have had portable methods added. In this context, the expression “the same in terms of behavior” means that methods with the same behavior as those specified are applicable, and in the same order.
- No portable class k may inherit, by virtue of being a direct or indirect subclass of a specified class, any slot for which the name is a symbol accessible in the **“COMMON-LISP-USER”** package or exported by any package defined in the [ANSI CL standard].
- Implementations are free to define implementation-specific before- and after-methods on specified generic functions. Implementations are also free to define implementation-specific around-methods with extending behavior.

29.2.3. Processing of the User Interface Macros

A list in which the first element is one of the symbols `DEFCLASS`, `DEFMETHOD`, `DEFGENERIC`, `DEFINE-METHOD-COMBINATION`, `CLOS:GENERIC-FUNCTION`, `CLOS:GENERIC-FLET` or `CLOS:GENERIC-LABELS`, and which has proper syntax for that macro is called a ***user interface macro form***. This document provides an extended specification of the `DEFCLASS`, `DEFMETHOD` and `DEFGENERIC` macros.

The user interface macros `DEFCLASS`, `DEFGENERIC` and `DEFMETHOD` can be used not only to define metaobjects that are instances of the corresponding standard metaobject class, but also to define metaobjects that are instances of appropriate portable metaobject classes. To make it possible for portable metaobject classes to properly process the information appearing in the macro form, this document provides a limited specification of the processing of these macro forms.

User interface macro forms can be *evaluated* or *compiled* and later *executed*. The effect of evaluating or executing a user interface macro form is specified in terms of calls to specified functions and generic functions which provide the actual behavior of the macro. The arguments received by these functions and generic functions are derived in a specified way from the macro form.

Converting a user interface macro form into the arguments to the appropriate functions and generic functions has two major aspects: the conversion of the macro argument syntax into a form more suitable for later processing, and the processing of macro arguments which are forms to be evaluated (including method bodies).

In the syntax of the `DEFCLASS` macro, the *`initform`* and *`default-initarg-initial-value-form`* arguments are forms which will be evaluated one or more times after the macro form is evaluated or executed. Special processing must be done on these arguments to ensure that the lexical scope of the forms is captured properly. This is done by building a function of zero arguments which, when called, returns the result of evaluating the form in the proper lexical environment.

In the syntax of the `DEFMETHOD` macro the *`forms`* argument is a list of forms that comprise the body of the method definition. This list of forms must be processed specially to capture the lexical scope of the macro form. In addition, the lexical functions available only in the body of methods must be introduced. To allow this and any other special processing (such as slot access optimization), a specializable protocol is used for processing the body of methods. This is discussed in Section 29.6.3.1.1, “Processing Method Bodies”.

29.2.3.1. Compile-file Processing of the User Interface Macros

It is a common practice for **Common Lisp** compilers, while processing a file or set of files, to maintain information about the definitions that have been compiled so far. Among other things, this makes it possible to ensure that a global macro definition (DEFMACRO form) which appears in a file will affect uses of the macro later in that file. This information about the state of the compilation is called the ***COMPILE-FILE* environment**.

When compiling files containing **CLOS** definitions, it is useful to maintain certain additional information in the COMPILE-FILE environment. This can make it possible to issue various kinds of warnings (e.g., lambda list congruence) and to do various performance optimizations that would not otherwise be possible.

At this time, there is such significant variance in the way existing **Common Lisp** implementations handle COMPILE-FILE environments that it would be premature to specify this mechanism. Consequently, this document specifies only the behavior of evaluating or executing user interface macro forms. What functions and generic functions are called during COMPILE-FILE processing of a user interface macro form is not specified. Implementations are free to define and document their own behavior. Users may need to check implementation-specific behavior before attempting to compile certain portable programs.

29.2.3.2. Compile-file Processing of Specific User Interface Macros

DEFCLASS
: Section 29.3.1, "Macro DEFCLASS"

Implementation dependent: only in CLISP

CLISP evaluates DEFCLASS forms also at compile time.

DEFMETHOD
: Section 29.6.3.1, "Macro DEFMETHOD"

Implementation dependent: only in CLISP

CLISP does **not** evaluate `DEFMETHOD` forms at compile time except as necessary for signature checking.

`DEFGENERIC`

Section 29.5.3.1, "Macro `DEFGENERIC`"

Implementation dependent: only in CLISP

CLISP does **not** evaluate `DEFGENERIC` forms at compile time except as necessary for signature checking.

29.2.4. Metaobject Initialization Protocol

Like other objects, metaobjects can be created by calling `MAKE-INSTANCE`. The initialization arguments passed to `MAKE-INSTANCE` are used to initialize the metaobject in the usual way. The set of legal initialization arguments, and their interpretation, depends on the kind of metaobject being created. Implementations and portable programs are free to extend the set of legal initialization arguments. Detailed information about the initialization of each kind of metaobject are provided in the appropriate sections:

- Section 29.3.5.1, "Initialization of class metaobjects"
- Section 29.3.5.2, "Reinitialization of class metaobjects"
- Section 29.5.3.3, "Initialization of generic function metaobjects"
- Section 29.3.4, "Class Finalization Protocol"
- Section 29.10.1, "Instance Structure Protocol"
- Section 29.10.2, "Funcallable Instances"
- Section 29.5.3.2, "Generic Function Invocation Protocol"
- Section 29.11, "Dependent Maintenance"

29.3. Classes

29.3.1. Macro DEFCLASS

The evaluation or execution of a `DEFCLASS` form results in a call to the `CLOS:ENSURE-CLASS` function. The arguments received by `CLOS:ENSURE-CLASS` are derived from the `DEFCLASS` form in a defined way. The exact macro-expansion of the `DEFCLASS` form is not defined, only the relationship between the arguments to the `DEFCLASS` macro and the arguments received by the `CLOS:ENSURE-CLASS` function. Examples of typical `DEFCLASS` forms and sample expansions are shown in the following two examples:

A `DEFCLASS` form with standard slot and class options and an expansion of it that would result in the proper call to `CLOS:ENSURE-CLASS`.

```
(defclass plane (moving-object graphics-object)
  ((altitude :initform 0 :accessor plane-altitude)
   (speed))
  (:default-initargs :engine *jet*))

(ensure-class 'plane
  ':direct-superclasses '(moving-object graphics-object)
  ':direct-slots (list (list ':name 'altitude
                             ':initform '0
```

```
                                                  ':initfunction #'(lambda () '
                                                  ':readers '(plane-altitude)
                                                  ':writers '((setf plane-alti
                                        (list ':name 'speed))
   ':direct-default-initargs (list (list ':engine
                                           '*jet*
                                           #'(lambda () *jet
```

A `DEFCLASS` form with non-standard class and slot options, and an expansion of it which results in the proper call to `CLOS:ENSURE-CLASS`. Note that the order of the slot options has not affected the order of the properties in the canonicalized slot specification, but has affected the order of the elements in the lists which are the values of those properties.

```
(defclass sst (plane)
  ((mach mag-step 2
         locator sst-mach
         locator mach-location
         :reader mach-speed
         :reader mach))
  (:metaclass faster-class)
  (another-option foo bar))

(ensure-class 'sst
  ':direct-superclasses '(plane)
  ':direct-slots (list (list ':name 'mach
                              ':readers '(mach-speed mach)
                              'mag-step '2
                              'locator '(sst-mach mach-loca
  ':metaclass 'faster-class
  'another-option '(foo bar))
```

- The *name* argument to `DEFCLASS` becomes the value of the first argument to `CLOS:ENSURE-CLASS`. This is the only positional argument accepted by `CLOS:ENSURE-CLASS`; all other arguments are keyword arguments.
- The `:DIRECT-SUPERCLASSES` argument to `DEFCLASS` becomes the value of the `:DIRECT-SUPERCLASSES` keyword argument to `CLOS:ENSURE-CLASS`.

- The `:DIRECT-SLOTS` argument to `DEFCLASS` becomes the value of the `:DIRECT-SLOTS` keyword argument to `CLOS:ENSURE-CLASS`. Special processing of this value is done to regularize the form of each slot specification and to properly capture the lexical scope of the initialization forms. This is done by converting each slot specification to a property list called a ***canonicalized slot specification***. The resulting list of canonicalized slot specifications is the value of the `:DIRECT-SLOTS` keyword argument.

Canonicalized slot specifications are later used as the keyword arguments to a generic function which will, in turn, pass them to `MAKE-INSTANCE` for use as a set of initialization arguments. Each canonicalized slot specification is formed from the corresponding slot specification as follows:

- The name of the slot is the value of the `:NAME` property. This property appears in every canonicalized slot specification.
- When the `:INITFORM` slot option is present in the slot specification, then both the `:INITFORM` and `:INITFUNCTION` properties are present in the canonicalized slot specification. The value of the `:INITFORM` property is the initialization form. The value of the `:INITFUNCTION` property is a function of zero arguments which, when called, returns the result of evaluating the initialization form in its proper lexical environment.
- If the `:INITFORM` slot option is not present in the slot specification, then either the `:INITFUNCTION` property will not appear, or its value will be false. In such cases, the value of the `:INITFORM` property, or whether it appears, is unspecified.
- The value of the `:INITARGS` property is a list of the values of each `:INITARG` slot option. If there are no `:INITARG` slot options, then either the `:INITARGS` property will not appear or its value will be the empty list.
- The value of the `:READERS` property is a list of the values of each `:READER` and `:ACCESSOR` slot option. If there are no `:READER` or `:ACCESSOR` slot options, then either the `:READERS` property will not appear or its value will be the empty list.
- The value of the `:WRITERS` property is a list of the values specified by each `:WRITER` and `:ACCESSOR` slot option. The value specified by a `:WRITER` slot option is just the value of the slot option. The value specified by an `:ACCESSOR` slot option is a two element list: the first element is the symbol `SETF`, the second element is the value of the slot option. If there are

no `:WRITER` or `:ACCESSOR` slot options, then either the `:WRITERS` property will not appear or its value will be the empty list.
  - The value of the `:DOCUMENTATION` property is the value of the `:DOCUMENTATION` slot option. If there is no `:DOCUMENTATION` slot option, then either the `:DOCUMENTATION` property will not appear or its value will be false.
  - All other slot options appear as the values of properties with the same name as the slot option. Note that this includes not only the remaining standard slot options (`:ALLOCATION` and `:TYPE`), but also any other options and values appearing in the slot specification. If one of these slot options appears more than once, the value of the property will be a list of the specified values.
  - An implementation is free to add additional properties to the canonicalized slot specification provided these are not symbols accessible in the **"COMMON-LISP-USER"** package, or exported by any package defined in the [ANSI CL standard].

- The *`default initargs`* class option, if it is present in the `DEFCLASS` form, becomes the value of the `:DIRECT-DEFAULT-INITARGS` keyword argument to `CLOS:ENSURE-CLASS`. Special processing of this value is done to properly capture the lexical scope of the default value forms. This is done by converting each default initarg in the class option into a ***canonicalized default initialization argument***. The resulting list of canonicalized default initialization arguments is the value of the `:DIRECT-DEFAULT-INITARGS` keyword argument to `CLOS:ENSURE-CLASS`.

  A canonicalized default initarg is a list of three elements. The first element is the name; the second is the actual form itself; and the third is a function of zero arguments which, when called, returns the result of evaluating the default value form in its proper lexical environment.

Implementation dependent: only in CLISP

If a *`default initargs`* class option is not present in the `DEFCLASS` form, `:DIRECT-DEFAULT-INITARGS` `NIL` is passed to `CLOS:ENSURE-CLASS`.

This is needed to fulfill the [ANSI CL standard] requirement (see Section 4.6, "Redefining Classes [CLHS-4.3.6]") that the resulting `CLASS` object reflects the `DEFCLASS` form.

- The *`metaclass`* class option, if it is present in the `DEFCLASS` form, becomes the value of the `:METACLASS` keyword argument to `CLOS:ENSURE-CLASS`.

Implementation dependent: only in CLISP

If a *`metaclass`* class option is not present in the `DEFCLASS` form, `:METACLASS` `STANDARD-CLASS` is passed to `CLOS:ENSURE-CLASS`.

This is needed to fulfill the [ANSI CL standard] requirement (see Section 4.6, "Redefining Classes [CLHS-4.3.6]") that the resulting `CLASS` object reflects the `DEFCLASS` form.

- The *`documentation`* class option, if it is present in the `DEFCLASS` form, becomes the value of the `:DOCUMENTATION` keyword argument to `CLOS:ENSURE-CLASS`.

Implementation dependent: only in CLISP

If a *documentation* class option is not present in the DEFCLASS form, :DIRECT-DEFAULT-INITARGS NIL is passed to CLOS:ENSURE-CLASS.

This is needed to fulfill the [ANSI CL standard] requirement (see Section 4.6, "Redefining Classes [CLHS-4.3.6]") that the resulting CLASS object reflects the DEFCLASS form.

- Any other class options become the value of keyword arguments with the same name. The value of the keyword argument is the tail of the class option. An ERROR is SIGNALed if any class option appears more than once in the DEFCLASS form.

Implementation dependent: only in CLISP

The default initargs of the *metaclass* are added at the end of the list of arguments to pass to CLOS:ENSURE-CLASS.

This is needed to fulfill the [ANSI CL standard] requirement (see Section 4.6, "Redefining Classes [CLHS-4.3.6]") that the resulting CLASS object reflects the DEFCLASS form.

In the call to CLOS:ENSURE-CLASS, every element of its arguments appears in the same left-to-right order as the corresponding element of the DEFCLASS form, except that the order of the properties of canonicalized slot specifications is unspecified. The values of properties in canonicalized slot specifications do follow this ordering requirement. Other ordering relationships in the keyword arguments to CLOS:ENSURE-CLASS are unspecified.

The result of the call to `CLOS:ENSURE-CLASS` is returned as the result of evaluating or executing the `DEFCLASS` form.

29.3.2. Inheritance Structure of class metaobject Classes

Figure 29.2. Inheritance structure of class metaobject classes

Inheritance structure of class metaobject classes

29.3.3. Introspection: Readers for class metaobjects

In this and the following sections, the “reader” generic functions which simply return information associated with a particular kind of metaobject are presented together. General information is presented first, followed by a description of the purpose of each, and ending with the specified methods for these generic functions.

The reader generic functions which simply return information associated with class metaobjects are presented together in this section.

Each of the reader generic functions for class metaobjects has the same syntax, accepting one required argument called *`class`*, which must be a class metaobject; otherwise, an `ERROR` is `SIGNAL`ed. An `ERROR` is also `SIGNAL`ed if the class metaobject has not been initialized.

These generic functions can be called by the user or the implementation.

For any of these generic functions which returns a list, such lists will not be mutated by the implementation. The results are undefined if a portable program allows such a list to be mutated.

29.3.3.1. Generic Function `CLASS-NAME`

`(CLASS-NAME class)`

Returns the name of *`class`*. This value can be any Lisp object, but is usually a symbol, or `NIL` if the class has no name. This is the defaulted value of the `:NAME` initialization argument that was associated with the class during initialization or reinitialization. (Also see `(SETF CLASS-NAME)`.)

29.3.3.2. Generic Function `CLOS:CLASS-DIRECT-SUPERCLASSES`

`(CLOS:CLASS-DIRECT-SUPERCLASSES class)`

Returns a list of the direct superclasses of *`class`*. The elements of this list are class metaobjects. The empty list is returned if *`class`* has no direct superclasses. This is the defaulted value of the `:DIRECT-SUPERCLASSES` initialization argument that was associated with the class during initialization or reinitialization.

Implementation dependent: only in CLISP

For a class that has not yet been finalized, the returned list may contain `CLOS:FORWARD-REFERENCED-CLASS` instances as placeholder for classes that were not yet defined when finalization of the class was last attempted.

29.3.3.3. Generic Function `CLOS:CLASS-DIRECT-SLOTS`

`(CLOS:CLASS-DIRECT-SLOTS *class*)`

Returns a set of the direct slots of *`class`*. The elements of this set are direct slot definition metaobjects. If the class has no direct slots, the empty set is returned. This is the defaulted value of the `:DIRECT-SLOTS` initialization argument that was associated with the class during initialization and reinitialization.

29.3.3.4. Generic Function `CLOS:CLASS-DIRECT-DEFAULT-INITARGS`

`(CLOS:CLASS-DIRECT-DEFAULT-INITARGS *class*)`

Returns a list of the direct default initialization arguments for *`class`*. Each element of this list is a canonicalized default initialization argument. The empty list is returned if *`class`* has no direct default initialization arguments. This is the defaulted value of the `:DIRECT-DEFAULT-INITARGS` initialization argument that was associated with the class during initialization or reinitialization.

29.3.3.5. Generic Function CLOS:CLASS-PRECEDENCE-LIST

(CLOS:CLASS-PRECEDENCE-LIST *class*)

Returns the class precedence list of *class*. The elements of this list are class metaobjects.

During class finalization CLOS:FINALIZE-INHERITANCE calls CLOS:COMPUTE-CLASS-PRECEDENCE-LIST to compute the class precedence list of the class. That value is associated with the class metaobject and is returned by CLOS:CLASS-PRECEDENCE-LIST.

This generic function SIGNALs an ERROR if *class* has not been finalized.

29.3.3.6. Generic Function CLOS:CLASS-DIRECT-SUBCLASSES

(CLOS:CLASS-DIRECT-SUBCLASSES *class*)

Returns a set of the direct subclasses of *class*. The elements of this set are class metaobjects that all mention this class among their direct superclasses. The empty set is returned if *class* has no direct subclasses. This value is maintained by the generic functions CLOS:ADD-DIRECT-SUBCLASS and CLOS:REMOVE-DIRECT-SUBCLASS.

Implementation dependent: only in CLISP

The set of direct subclasses of a class is internally managed as a EXT:WEAK-LIST. Therefore the existence of the CLOS:CLASS-DIRECT-SUBCLASSES function does not prevent otherwise unreferenced classes from being garbage-collected.

29.3.3.7. Generic Function CLOS:CLASS-SLOTS

(CLOS:CLASS-SLOTS *class*)

Returns a possibly empty set of the slots accessible in instances of *class*. The elements of this set are effective slot definition metaobjects.

During class finalization CLOS:FINALIZE-INHERITANCE calls CLOS:COMPUTE-SLOTS to compute the slots of the class. That value is associated with the class metaobject and is returned by CLOS:CLASS-SLOTS.

This generic function SIGNALs an ERROR if *class* has not been finalized.

29.3.3.8. Generic Function CLOS:CLASS-DEFAULT-INITARGS

(CLOS:CLASS-DEFAULT-INITARGS *class*)

Returns a list of the default initialization arguments for *class*. Each element of this list is a canonicalized default initialization argument. The empty list is returned if *class* has no default initialization arguments.

During finalization CLOS:FINALIZE-INHERITANCE calls CLOS:COMPUTE-DEFAULT-INITARGS to compute the default initialization arguments for the class. That value is associated with the class metaobject and is returned by CLOS:CLASS-DEFAULT-INITARGS.

This generic function SIGNALs an ERROR if *class* has not been finalized.

29.3.3.9. Generic Function CLOS:CLASS-FINALIZED-P

(CLOS:CLASS-FINALIZED-P *class*)

Returns true if *class* has been finalized. Returns false otherwise. Also returns false if the *class* has not been initialized.

29.3.3.10. Generic Function CLOS:CLASS-PROTOTYPE

(CLOS:CLASS-PROTOTYPE *class*)

Returns a prototype instance of *class*. Whether the instance is initialized is not specified. The results are undefined if a portable program modifies the binding of any slot of a prototype instance.

This generic function SIGNALs an ERROR if *class* has not been finalized.

This allows non-consing[3] access to slots with allocation :CLASS:

```
(defclass counter ()
  ((count :allocation :class :initform 0 :reader how-many
(defmethod initialize-instance :after ((obj counter) &res
  (incf (slot-value obj 'count)))
(defclass counted-object (counter) ((name :initarg :name)
```

Now one can find out how many COUNTED-OBJECTs have been created by using (HOW-MANY (CLOS:CLASS-PROTOTYPE (FIND-CLASS 'COUNTER))):

```
(MAKE-INSTANCE 'counted-object :name 'foo)
⇒ #<COUNTED-OBJECT #x203028C9>
(HOW-MANY (CLOS:CLASS-PROTOTYPE (FIND-CLASS 'counter)))
⇒ 1
(MAKE-INSTANCE 'counted-object :name 'bar)
```

```
⇒ #<COUNTED-OBJECT #x20306CB1>
(HOW-MANY (CLOS:CLASS-PROTOTYPE (FIND-CLASS 'counter)))
⇒ 2
```

29.3.3.11. Methods

The specified methods for the class metaobject reader generic functions are presented below.

Each entry in the table indicates a method on one of the reader generic functions, specialized to a specified class. The number in each entry is a reference to the full description of the method. The full descriptions appear after the table.

Generic Function	STANDARD-CLASS, CLOS:FUNCALLABLE-STANDARD-CLASS	CLOS:FORWARD-REFERENCED-CLASS	BUILT-IN-CLASS
CLASS-NAME	1	1	8
CLOS:CLASS-DIRECT-SUPERCLASSES	1	4	7
CLOS:CLASS-DIRECT-SLOTS	1	4	4
CLOS:CLASS-DIRECT-DEFAULT-INITARGS	1	4	4
CLOS:CLASS-PRECEDENCE-LIST	2	3	7
CLOS:CLASS-DIRECT-SUBCLASSES	9	9	7
CLOS:CLASS-SLOTS	2	3	4
CLOS:CLASS-DEFAULT-INITARGS	2	3	4
CLOS:CLASS-FINALIZED-P	2	6	5

Generic Function	STANDARD-CLASS, CLOS:FUNCALLABLE-STANDARD-CLASS	CLOS:FORWARD-REFERENCED-CLASS	BUILT-IN-CLASS
CLOS:CLASS-PROTOTYPE	10	10	10

Class Reader Methods

1. This method returns the value which was associated with the class metaobject during initialization or reinitialization.
2. This method returns the value associated with the class metaobject by `CLOS:FINALIZE-INHERITANCE (STANDARD-CLASS)` or `CLOS:FINALIZE-INHERITANCE (CLOS:FUNCALLABLE-STANDARD-CLASS)`
3. This method `SIGNAL`s an `ERROR`.
4. This method returns the empty list.
5. This method returns true.
6. This method returns false.
7. This method returns a value derived from the information in Table 29.1, “Direct Superclass Relationships Among The Specified Metaobject Classes”, except that implementation-specific modifications are permitted as described in Section 29.2.2.1, “Implementation and User Specialization”.
8. This method returns the name of the built-in class.
9. This methods returns a value which is maintained by `CLOS:ADD-DIRECT-SUBCLASS(CLASS CLASS)` and `CLOS:REMOVE-DIRECT-SUBCLASS (CLASS CLASS)`. This method can be overridden only if those methods are overridden as well.
10. No behavior is specified for this method beyond that which is specified for the generic function.

29.3.4. Class Finalization Protocol

Class ***finalization*** is the process of computing the information a class inherits from its superclasses and preparing to actually allocate instances of the class. The class finalization process includes computing the class's class precedence list, the full set of slots accessible in instances of the class and the full set of default initialization arguments for the class. These values are associated with the class metaobject and can be accessed by calling the appropriate reader. In addition, the class finalization process makes decisions about how instances of the class will be implemented.

To support forward-referenced superclasses, and to account for the fact that not all classes are actually instantiated, class finalization is not done as part of the initialization of the class metaobject. Instead, finalization is done as a separate protocol, invoked by calling the generic function `CLOS:FINALIZE-INHERITANCE`. The exact point at which `CLOS:FINALIZE-INHERITANCE` is called depends on the class of the class metaobject; for `STANDARD-CLASS` it is called sometime after all the classes superclasses are defined, but no later than when the first instance of the class is allocated (by `ALLOCATE-INSTANCE`).

The first step of class finalization is computing the class precedence list. Doing this first allows subsequent steps to access the class precedence list. This step is performed by calling the generic function `CLOS:COMPUTE-CLASS-PRECEDENCE-LIST`. The value returned from this call is associated with the class metaobject and can be accessed by calling the `CLOS:CLASS-PRECEDENCE-LIST` generic function.

The second step is computing the full set of slots that will be accessible in instances of the class. This step is performed by calling the generic function `CLOS:COMPUTE-SLOTS`. The result of this call is a list of effective slot definition metaobjects. This value is associated with the class metaobject and can be accessed by calling the `CLOS:CLASS-SLOTS` generic function.

The behavior of `CLOS:COMPUTE-SLOTS` is itself layered, consisting of calls to `CLOS:EFFECTIVE-SLOT-DEFINITION-CLASS` and `CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION`.

The final step of class finalization is computing the full set of initialization arguments for the class. This is done by calling the generic function `CLOS:COMPUTE-DEFAULT-INITARGS`. The value returned by this generic function is associated with the class metaobject and can be accessed by calling `CLOS:CLASS-DEFAULT-INITARGS`.

If the class was previously finalized, `CLOS:FINALIZE-INHERITANCE` may call `MAKE-INSTANCES-OBSOLETE`. The circumstances under which this happens are described in the [ANSI CL standard] section Section 4.6, "Redefining Classes [CLHS-4.3.6]".

Forward-referenced classes, which provide a temporary definition for a class which has been referenced but not yet defined, can never be finalized. An `ERROR` is `SIGNAL`ed if `CLOS:FINALIZE-INHERITANCE` is called on a forward-referenced class.

29.3.5. Class Initialization

29.3.5.1. Initialization of class metaobjects

A class metaobject can be created by calling `MAKE-INSTANCE`. The initialization arguments establish the definition of the class. A class metaobject can be redefined by calling `REINITIALIZE-INSTANCE`. Some classes of class metaobject do not support redefinition; in these cases, `REINITIALIZE-INSTANCE` `SIGNAL`s an `ERROR`.

Initialization of a class metaobject must be done by calling `MAKE-INSTANCE` and allowing it to call `INITIALIZE-INSTANCE`. Reinitialization of a class metaobject must be done by calling `REINITIALIZE-INSTANCE`. Portable programs must **not**

- ... call `INITIALIZE-INSTANCE` directly to initialize a class metaobject;
- ... call `SHARED-INITIALIZE` directly to initialize or reinitialize a class metaobject;

- ... call `CHANGE-CLASS` to change the class of any class metaobject or to turn a non-class object into a class metaobject.

Since metaobject classes may not be redefined, no behavior is specified for the result of calls to `UPDATE-INSTANCE-FOR-REDEFINED-CLASS` on class metaobjects. Since the class of class metaobjects may not be changed, no behavior is specified for the result of calls to `UPDATE-INSTANCE-FOR-DIFFERENT-CLASS` on class metaobjects.

During initialization or reinitialization, each initialization argument is checked for errors and then associated with the class metaobject. The value can then be accessed by calling the appropriate accessor as shown in Table 29.2, “Initialization arguments and accessors for class metaobjects”.

This section begins with a description of the error checking and processing of each initialization argument. This is followed by a table showing the generic functions that can be used to access the stored initialization arguments. Initialization behavior specific to the different specified class metaobject classes comes next. The section ends with a set of restrictions on portable methods affecting class metaobject initialization and reinitialization.

In these descriptions, the phrase “this argument defaults to *`value`*” means that when that initialization argument is not supplied, initialization or reinitialization is performed as if *`value`* had been supplied. For some initialization arguments this could be done by the use of default initialization arguments, but whether it is done this way is not specified. Implementations are free to define default initialization arguments for specified class metaobject classes. Portable programs are free to define default initialization arguments for portable subclasses of the class `CLASS`.

Unless there is a specific note to the contrary, then during reinitialization, if an initialization argument is not supplied, the previously stored value is left unchanged.

- The `:DIRECT-DEFAULT-INITARGS` argument is a list of canonicalized default initialization arguments.

An ERROR is SIGNALed if this value is not a proper list, or if any element of the list is not a canonicalized default initialization argument.

If the class metaobject is being initialized, this argument defaults to the empty list.

- The :DIRECT-SLOTS argument is a list of canonicalized slot specifications.

  An ERROR is SIGNALed if this value is not a proper list or if any element of the list is not a canonicalized slot specification.

  After error checking, this value is converted to a list of direct slot definition metaobjects before it is associated with the class metaobject. Conversion of each canonicalized slot specification to a direct slot definition metaobject is a two-step process. First, the generic function CLOS:DIRECT-SLOT-DEFINITION-CLASS is called with the class metaobject and the canonicalized slot specification to determine the class of the new direct slot definition metaobject; this permits both the class metaobject and the canonicalized slot specification to control the resulting direct slot definition metaobject class. Second, MAKE-INSTANCE is applied to the direct slot definition metaobject class and the canonicalized slot specification. This conversion could be implemented as shown in the following code:

```
(DEFUN convert-to-direct-slot-definition (class canoni
  (APPLY #'MAKE-INSTANCE
         (APPLY #'CLOS:DIRECT-SLOT-DEFINITION-CLASS
                class canonicalized-slot)
         canonicalized-slot))
```

  If the class metaobject is being initialized, this argument defaults to the empty list.

  Once the direct slot definition metaobjects have been created, the specified reader and writer methods are created. The generic functions CLOS:READER-METHOD-CLASS and CLOS:WRITER-METHOD-CLASS are called to determine the classes of the method metaobjects created.

- The :DIRECT-SUPERCLASSES argument is a list of class metaobjects. Classes which do not support multiple inheritance signal an error if the list contains more than one element.

  An ERROR is SIGNALed if this value is not a proper list or if CLOS:VALIDATE-SUPERCLASS applied to *class* and any element of this list returns false.

  When the class metaobject is being initialized, and this argument is either not supplied or is the empty list, this argument defaults as follows: if the class is an instance of STANDARD-CLASS or one of its subclasses the default value is a list of the class STANDARD-OBJECT; if the class is an instance of CLOS:FUNCALLABLE-STANDARD-CLASS or one of its subclasses the default value is a list of the class CLOS:FUNCALLABLE-STANDARD-OBJECT.

  ### Implementation dependent: only in CLISP

  If the class is an instance of STRUCTURE-CLASS or one of its subclasses the default value is a list of the class STRUCTURE-OBJECT

  After any defaulting of the value, the generic function CLOS:ADD-DIRECT-SUBCLASS is called once for each element of the list.

  When the class metaobject is being reinitialized and this argument is supplied, the generic function CLOS:REMOVE-DIRECT-SUBCLASS is called once for each class metaobject in the previously stored value but not in the new value; the generic function CLOS:ADD-DIRECT-SUBCLASS is called once for each class metaobject in the new value but not in the previously stored value.

- The :DOCUMENTATION argument is a STRING or NIL. An ERROR is SIGNALed if it is not. This argument default to NIL during initialization.

- The :NAME argument is an object.

If the class is being initialized, this argument defaults to NIL.

After the processing and defaulting of initialization arguments described above, the value of each initialization argument is associated with the class metaobject. These values can then be accessed by calling the corresponding generic function. The correspondences are as follows:

Table 29.2. Initialization arguments and accessors for class metaobjects

Initialization Argument	Generic Function
:DIRECT-DEFAULT-INITARGS	CLOS:CLASS-DIRECT-DEFAULT-INITARGS
:DIRECT-SLOTS	CLOS:CLASS-DIRECT-SLOTS
:DIRECT-SUPERCLASSES	CLOS:CLASS-DIRECT-SUPERCLASSES
:DOCUMENTATION	DOCUMENTATION
:NAME	CLASS-NAME

Instances of the class STANDARD-CLASS support multiple inheritance and reinitialization. Instances of the class CLOS:FUNCALLABLE-STANDARD-CLASS support multiple inheritance and reinitialization. For forward referenced classes, all of the initialization arguments default to NIL.

Implementation dependent: only in CLISP

Instances of the class STRUCTURE-CLASS do not support multiple inheritance and reinitialization.

Since built-in classes cannot be created or reinitialized by the user, an ERROR is SIGNALed if INITIALIZE-INSTANCE or REINITIALIZE-INSTANCE are called to initialize or reinitialize a derived instance of the class BUILT-IN-CLASS.

29.3.5.1.1. Methods

It is not specified which methods provide the initialization and reinitialization behavior described above. Instead, the information needed to allow portable programs to specialize this behavior is presented as a set of restrictions on the methods a portable program can define. The model is that portable initialization methods have access to the class metaobject when either all or none of the specified initialization has taken effect.

These restrictions govern the methods that a portable program can define on the generic functions `INITIALIZE-INSTANCE`, `REINITIALIZE-INSTANCE`, and `SHARED-INITIALIZE`. These restrictions apply only to methods on these generic functions for which the first specializer is a subclass of the class `CLASS`. Other portable methods on these generic functions are not affected by these restrictions.

- Portable programs must not define methods on `SHARED-INITIALIZE`.
- For `INITIALIZE-INSTANCE` and `REINITIALIZE-INSTANCE`:
    - Portable programs must not define primary methods.
    - Portable programs may define around-methods, but these must be extending, not overriding methods.
    - Portable before-methods must assume that when they are run, none of the initialization behavior described above has been completed.
    - Portable after-methods must assume that when they are run, all of the initialization behavior described above has been completed.

The results are undefined if any of these restrictions are violated.

29.3.5.1.2. Initialization of Anonymous Classes

class metaobjects created with `MAKE-INSTANCE` are usually ***anonymous***; that is, they have no proper name. An anonymous class metaobject can be given a proper name using (SETF FIND-CLASS) and (SETF CLASS-NAME).

When a class metaobject is created with MAKE-INSTANCE, it is initialized in the usual way. The initialization arguments passed to MAKE-INSTANCE are use to establish the definition of the class. Each initialization argument is checked for errors and associated with the class metaobject. The initialization arguments correspond roughly to the arguments accepted by the DEFCLASS macro, and more closely to the arguments accepted by the CLOS:ENSURE-CLASS function.

Some class metaobject classes allow their instances to be redefined. When permissible, this is done by calling REINITIALIZE-INSTANCE. This is discussed in the next section.

An example of creating an anonymous class directly using MAKE-INSTANCE follows:

```
(flet ((zero () 0)
       (propellor () *propellor*))
  (make-instance 'standard-class
    :name '(my-class foo)
    :direct-superclasses (list (find-class 'plane)
                               another-anonymous-class)
    :direct-slots `((:name x
                     :initform 0
                     :initfunction ,#'zero
                     :initargs (:x)
                     :readers (position-x)
                     :writers ((setf position-x)))
                    (:name y
                     :initform 0
                     :initfunction ,#'zero
                     :initargs (:y)
                     :readers (position-y)
                     :writers ((setf position-y))))
    :direct-default-initargs `((:engine *propellor* ,#'pr
```

29.3.5.2. Reinitialization of class metaobjects

Some class metaobject classes allow their instances to be reinitialized. This is done by calling `REINITIALIZE-INSTANCE`. The initialization arguments have the same interpretation as in class initialization.

If the class metaobject was finalized before the call to `REINITIALIZE-INSTANCE`, `CLOS:FINALIZE-INHERITANCE` will be called again once all the initialization arguments have been processed and associated with the class metaobject. In addition, once finalization is complete, any dependents of the class metaobject will be updated by calling `CLOS:UPDATE-DEPENDENT`.

29.3.6. Customization

29.3.6.1. Generic Function (SETF CLASS-NAME)

Syntax

`((SETF CLASS-NAME) new-name class)`

Arguments

class
: a class metaobject.

new-name
: any Lisp object.

Value

The *new-name* argument.

Purpose

This function changes the name of *class* to *new-name*. This value is usually a symbol, or `NIL` if the class has no name.

This function works by calling `REINITIALIZE-INSTANCE` with *class* as its first argument, the symbol `:NAME` as its second argument and *new-name* as its third argument.

29.3.6.2. Generic Function CLOS:ENSURE-CLASS

Syntax

`(CLOS:ENSURE-CLASS name &KEY &ALLOW-OTHER-KEYS)`

Arguments

name
: a `SYMBOL`.

keyword arguments
: Some of the keyword arguments accepted by this function are actually processed by `CLOS:ENSURE-CLASS-USING-CLASS`, others are processed during initialization of the class metaobject (as described in Section 29.3.5.1, “Initialization of class metaobjects”).

Value

A class metaobject.

Purpose

This function is called to define or redefine a class with the specified name, and can be called by the user or the implementation. It is the functional equivalent of `DEFCLASS`, and is called by the expansion of the `DEFCLASS` macro.
The behavior of this function is actually implemented by the generic function `CLOS:ENSURE-CLASS-USING-CLASS`. When `CLOS:ENSURE-CLASS` is called, it immediately calls `CLOS:ENSURE-CLASS-USING-CLASS` and returns that result as its own.
The first argument to `CLOS:ENSURE-CLASS-USING-CLASS` is computed as follows:

- If *name* names a class (`FIND-CLASS` returns a class when called with *name*) use that class.
- Otherwise use `NIL`.

The second argument is *name*. The remaining arguments are the complete set of keyword arguments received by the `CLOS:ENSURE-CLASS` function.

29.3.6.3. Generic Function `CLOS:ENSURE-CLASS-USING-CLASS`

Syntax

(`CLOS:ENSURE-CLASS-USING-CLASS` *class name* `&KEY` `:DIRECT-DEFAULT-INITARGS` `:DIRECT-SLOTS` `:DIRECT-SUPERCLASSES` `:NAME` `:METACLASS` `&ALLOW-OTHER-KEYS`)

Arguments

class
: a class metaobject or `NIL`.

name
: a class name.

`:METACLASS`
: a class metaobject class or a class metaobject class name. If this argument is not supplied, it defaults to the class named `STANDARD-CLASS`. If a class name is supplied, it is interpreted as the class with that name. If a class name is supplied, but there is no such class, an `ERROR` is `SIGNAL`ed.

`:DIRECT-SUPERCLASSES`

a list of which each element is a class metaobject or a class name. An `ERROR` is `SIGNAL`ed if this argument is not a proper list.

additional keyword arguments
See Section 29.3.5.1, "Initialization of class metaobjects"

Value
A class metaobject.

Purpose
This generic function is called to define or modify the definition of a named class. It is called by the `CLOS:ENSURE-CLASS` function. It can also be called directly.

The first step performed by this generic function is to compute the set of initialization arguments which will be used to create or reinitialize the named class. The initialization arguments are computed from the full set of keyword arguments received by this generic function as follows:

- The `:METACLASS` argument is not included in the initialization arguments.
- If the `:DIRECT-SUPERCLASSES` argument was received by this generic function, it is converted into a list of class metaobjects. This conversion does not affect the structure of the supplied `:DIRECT-SUPERCLASSES` argument. For each element in the `:DIRECT-SUPERCLASSES` argument:
  - If the element is a class metaobject, that class metaobject is used.
  - If the element names a class, that class metaobject is used.
  - Otherwise an instance of the class `CLOS:FORWARD-REFERENCED-CLASS` is created and used. The proper name of the newly created forward referenced class metaobject is set to the element.

Implementation dependent: only in CLISP

A new `CLOS:FORWARD-REFERENCED-CLASS` instance is only created when one for the given class name does not yet exist; otherwise the existing one is reused. See Implementation of class `CLOS:FORWARD-REFERENCED-CLASS` in **CLISP**.

- All other keyword arguments are included directly in the initialization arguments.

If the *`class`* argument is `NIL`, a new class metaobject is created by calling the `MAKE-INSTANCE` generic function with the value of the `:METACLASS` argument as its first argument, and the previously computed initialization arguments. The proper name of the newly created class metaobject is set to *`name`*. The newly created class metaobject is returned.
If the *`class`* argument is a forward referenced class, `CHANGE-CLASS` is called to change its class to the value specified by the `:METACLASS` argument. The class metaobject is then reinitialized with the previously initialization arguments. (This is a documented violation of the general constraint that `CHANGE-CLASS` may not be used with class metaobjects.)

Implementation dependent: only in CLISP

The *`class`* argument cannot be a forward referenced class. See Implementation of class `CLOS:FORWARD-REFERENCED-CLASS` in **CLISP**.

If the class of the *`class`* argument is not the same as the class specified by the `:METACLASS` argument, an `ERROR` is `SIGNAL`ed.
Otherwise, the class metaobject *`class`* is redefined by calling the `REINITIALIZE-INSTANCE` generic function with *`class`* and the initialization arguments. The *`class`* argument is then returned.

Methods

`(CLOS:ENSURE-CLASS-USING-CLASS (`*`class`* `CLASS)` *`name`* `&KEY :METACLASS :DIRECT-SUPERCLASSES &ALLOW-OTHER-KEYS)`

This method implements the behavior of the generic function in the case where the *`class`* argument is a class.
This method can be overridden.

(CLOS:ENSURE-CLASS-USING-CLASS (*class* CLOS:FORWARD-REFERENCED-CLASS) *name* &KEY :METACLASS :DIRECT-SUPERCLASSES &ALLOW-OTHER-KEYS)

This method implements the behavior of the generic function in the case where the *class* argument is a forward referenced class.

Implementation dependent: only in CLISP

This method does not exist. See Implementation of class CLOS:FORWARD-REFERENCED-CLASS in **CLISP**. Use the method specialized on NULL instead.

(CLOS:ENSURE-CLASS-USING-CLASS (*class* NULL) *name* &KEY :METACLASS :DIRECT-SUPERCLASSES &ALLOW-OTHER-KEYS)

This method implements the behavior of the generic function in the case where the *class* argument is NIL.

29.3.6.4. Generic Function CLOS:FINALIZE-INHERITANCE

Syntax

(CLOS:FINALIZE-INHERITANCE *class*)

Arguments

class

a class metaobject.

Values

The values returned by this generic function is unspecified.

Purpose

This generic function is called to finalize a class metaobject. This is described in Section 29.3.4, “Class Finalization Protocol”

After CLOS:FINALIZE-INHERITANCE returns, the class metaobject is finalized and the result of calling CLOS:CLASS-FINALIZED-P on the class metaobject will be true.

Methods

(CLOS:FINALIZE-INHERITANCE (*class* STANDARD-CLASS))
(CLOS:FINALIZE-INHERITANCE (*class* CLOS:FUNCALLABLE-STANDARD-CLASS))

No behavior is specified for these methods beyond that which is specified for their respective generic functions.

(CLOS:FINALIZE-INHERITANCE (*class* CLOS:FORWARD-REFERENCED-CLASS))

This method `SIGNAL`s an `ERROR`.

29.3.6.5. Generic Function MAKE-INSTANCE

Syntax

`(MAKE-INSTANCE` *`class`* `&REST` *`initargs`*`)`

Arguments

`class`

a class metaobject or a class name.

`initargs`

a list of alternating initialization argument names and values.

Value

A newly allocated and initialized instance of *`class`*.

Purpose

The generic function `MAKE-INSTANCE` creates and returns a new instance of the given class. Its behavior and use is described in the [ANSI CL standard].

Methods

(MAKE-INSTANCE (*class* SYMBOL) &REST *initargs*)

This method simply invokes `MAKE-INSTANCE` recursively on the arguments `(FIND-CLASS` *`class`*`)` and *`initargs`*.

(MAKE-INSTANCE (*class* STANDARD-CLASS) &REST *initargs*)
(MAKE-INSTANCE (*class* CLOS:FUNCALLABLE-STANDARD-CLASS) &REST *initargs*)

These methods implement the behavior of `MAKE-INSTANCE` described in the [ANSI CL standard] section 7.1 “Object Creation and Initialization”.

29.3.6.6. Generic Function ALLOCATE-INSTANCE

Syntax

(ALLOCATE-INSTANCE *class* &REST *initargs*)

Arguments

class
: a class metaobject.

initargs
: alternating initialization argument names and values.

Value

A newly allocated instance of *class*

Purpose

This generic function is called to create a new, uninitialized instance of a class. The interpretation of the concept of an ***uninitialized*** instance depends on the class metaobject class.

Before allocating the new instance, CLOS:CLASS-FINALIZED-P is called to see if *class* has been finalized. If it has not been finalized, CLOS:FINALIZE-INHERITANCE is called before the new instance is allocated.

Methods

(ALLOCATE-INSTANCE (*class* STANDARD-CLASS) &REST *initargs*

This method allocates storage in the instance for each slot with allocation :INSTANCE. These slots are unbound. Slots with any other allocation are ignored by this method (no ERROR is SIGNALed).

(ALLOCATE-INSTANCE (*class* CLOS:FUNCALLABLE-STANDARD-CLASS) &REST *initargs*)

This method allocates storage in the instance for each slot with allocation :INSTANCE. These slots are unbound. Slots with any other allocation are ignored by this method (no ERROR is SIGNALed).

The funcallable instance function of the instance is undefined - the results are undefined if the instance is applied to arguments before CLOS:SET-FUNCALLABLE-INSTANCE-FUNCTION has been used to set the funcallable instance function.

(ALLOCATE-INSTANCE (*class* BUILT-IN-CLASS) &REST *initargs*)

This method SIGNALs an ERROR.

29.3.6.7. Generic Function CLOS:VALIDATE-SUPERCLASS

Syntax

(CLOS:VALIDATE-SUPERCLASS *class superclass*)

Arguments

class
: a class metaobject.

superclass
: A class metaobject.

Value

BOOLEAN.

Purpose

This generic function is called to determine whether the class *superclass* is suitable for use as a superclass of *class*.

This generic function can be be called by the implementation or user code. It is called during class metaobject initialization and reinitialization, before the direct superclasses are stored. If this generic function returns false, the initialization or reinitialization will signal an error.

Methods

(CLOS:VALIDATE-SUPERCLASS (*class* CLASS) (*superclass* CLASS))

This method returns true in three situations:

i. If the *superclass* argument is the class named T,
ii. if the class of the *class* argument is the same as the class of the *superclass* argument, or
iii. if the class of one of the arguments is STANDARD-CLASS and the class of the other is CLOS:FUNCALLABLE-STANDARD-CLASS.

In all other cases, this method returns false.

This method can be overridden.

Implementation dependent: only in CLISP

This method also returns true in a fourth situation:

iv. If the class of the `class` argument is a subclass of the class of the `superclass` argument.

Remarks. Defining a method on `CLOS:VALIDATE-SUPERCLASS` requires detailed knowledge of of the internal protocol followed by each of the two class metaobject classes. A method on `CLOS:VALIDATE-SUPERCLASS` which returns true for two different class metaobject classes declares that they are compatible.

29.3.6.8. Generic Function `CLOS:COMPUTE-DIRECT-SLOT-DEFINITION-INITARGS`

Implementation dependent: only in CLISP

Syntax

`(CLOS:COMPUTE-DIRECT-SLOT-DEFINITION-INITARGS` *`class`* `&REST` *`slot-spec`*`)`

Arguments

`class`

a class metaobject.

`slot-spec`

a canonicalized slot specification.

Value

A list of initialization arguments for a direct slot definition metaobject.

Purpose

This generic function determines the initialization arguments for the direct slot definition for a slot in a class. It is called during initialization of a class. The resulting initialization arguments are

passed to `CLOS:DIRECT-SLOT-DEFINITION-CLASS` and then to `MAKE-INSTANCE`.
This generic function uses the supplied canonicalized slot specification. The value of `:NAME` in the returned initargs is the same as the value of `:NAME` in the supplied *`slot-spec`* argument.

Methods

(`CLOS:COMPUTE-DIRECT-SLOT-DEFINITION-INITARGS` (*`class`* `STANDARD-CLASS`) `&REST` *`slot-spec`*)
(`CLOS:COMPUTE-DIRECT-SLOT-DEFINITION-INITARGS` (*`class`* `CLOS:FUNCALLABLE-STANDARD-CLASS`) `&REST` *`slot-spec`*)
This method returns *`slot-spec`* unmodified.
This method can be overridden.

29.3.6.9. Generic Function `CLOS:DIRECT-SLOT-DEFINITION-CLASS`

Syntax
(`CLOS:DIRECT-SLOT-DEFINITION-CLASS` *`class`* `&REST` *`initargs`*)

Arguments

`class`
a class metaobject.

`initargs`
a set of initialization arguments and values.

Value
A subclass of the class `CLOS:DIRECT-SLOT-DEFINITION`.

Purpose
When a class is initialized, each of the canonicalized slot specifications must be converted to a direct slot definition metaobject. This generic function is called to determine the class of that direct slot definition metaobject.
The *`initargs`* argument is simply the canonicalized slot specification for the slot.

Methods

(CLOS:DIRECT-SLOT-DEFINITION-CLASS (*class* STANDARD-CLASS) &REST *initargs*)
(CLOS:DIRECT-SLOT-DEFINITION-CLASS (*class* CLOS:FUNCALLABLE-STANDARD-CLASS) &REST *initargs*)

These methods return the class `CLOS:STANDARD-DIRECT-SLOT-DEFINITION`.
These methods can be overridden.

29.3.6.10. Generic Function CLOS:COMPUTE-CLASS-PRECEDENCE-LIST

Syntax

`(CLOS:COMPUTE-CLASS-PRECEDENCE-LIST class)`

Arguments

class

a class metaobject.

Value

A list of class metaobjects.

Purpose

This generic-function is called to determine the class precedence list of a class.

The result is a list which contains each of *class* and its superclasses once and only once. The first element of the list is *class* and the last element is the class named `T`.

All methods on this generic function must compute the class precedence list as a function of the ordered direct superclasses of the superclasses of *class*. The results are undefined if the rules used to compute the class precedence list depend on any other factors.

When a class is finalized, `CLOS:FINALIZE-INHERITANCE` calls this generic function and associates the returned value with the class metaobject. The value can then be accessed by calling `CLOS:CLASS-PRECEDENCE-LIST`.

The list returned by this function will not be mutated by the implementation. The results are undefined if a portable program mutates the list returned by this function.

Methods

(CLOS:COMPUTE-CLASS-PRECEDENCE-LIST (*class* CLASS))

This method computes the class precedence list according to the rules described in the [ANSI CL standard] section 4.3.5 “Determining the Class Precedence List”.

This method `SIGNAL`s an `ERROR` if *`class`* or any of its superclasses is a forward referenced class.

This method can be overridden.

29.3.6.11. Generic Function CLOS:COMPUTE-SLOTS

Syntax

`(CLOS:COMPUTE-SLOTS class)`

Arguments

`class`

a class metaobject.

Value

A set of effective slot definition metaobjects.

Purpose

This generic function computes a set of effective slot definition metaobjects for the class *`class`*. The result is a list of effective slot definition metaobjects: one for each slot that will be accessible in instances of *`class`*.

This generic function proceeds in 3 steps:

The first step collects the full set of direct slot definitions from the superclasses of *`class`*.

The direct slot definitions are then collected into individual lists, one list for each slot name associated with any of the direct slot definitions. The slot names are compared with `EQL`. Each such list is then sorted into class precedence list order. Direct slot definitions coming from classes earlier in the class precedence list of *`class`* appear before those coming from classes later in the class precedence list. For each slot name, the generic function `CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION` is called to compute an effective slot definition. The result of `CLOS:COMPUTE-SLOTS` is a list of these effective slot definitions, in unspecified order.

In the final step, the location for each effective slot definition is set. This is done by specified around-methods; portable methods cannot take over this behavior. For more information on the slot definition locations, see Section 29.10.1, “Instance Structure Protocol”.
The list returned by this function will not be mutated by the implementation. The results are undefined if a portable program mutates the list returned by this function.

Methods

(CLOS:COMPUTE-SLOTS (*class* STANDARD-CLASS))
(CLOS:COMPUTE-SLOTS (*class* CLOS:FUNCALLABLE-STANDARD-CLASS))
These methods implement the specified behavior of the generic function.
These methods can be overridden.

(CLOS:COMPUTE-SLOTS :AROUND (*class* STANDARD-CLASS))
(CLOS:COMPUTE-SLOTS :AROUND (*class* CLOS:FUNCALLABLE-STANDARD-CLASS))
These methods implement the specified behavior of computing and storing slot locations. These methods cannot be overridden.

29.3.6.12. Generic Function CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION

Syntax

(CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION *class* *name* *direct-slot-definitions*)

Arguments

class
a class metaobject.

name
a slot name.

direct-slot-definitions
an ordered list of direct slot definition metaobjects. The most specific direct slot definition metaobject appears first in the list.

Value

An effective slot definition metaobject.

Purpose

This generic function determines the effective slot definition for a slot in a class. It is called by `CLOS:COMPUTE-SLOTS` once for each slot accessible in instances of *`class`*.

This generic function uses the supplied list of direct slot definition metaobjects to compute the inheritance of slot properties for a single slot. The returned effective slot definition represents the result of computing the inheritance. The name of the new effective slot definition is the same as the name of the direct slot definitions supplied.

The class of the effective slot definition metaobject is determined by calling `CLOS:EFFECTIVE-SLOT-DEFINITION-CLASS`. The effective slot definition is then created by calling `MAKE-INSTANCE`. The initialization arguments passed in this call to `MAKE-INSTANCE` are used to initialize the new effective slot definition metaobject. See Section 29.4, “Slot Definitions” for details.

Methods

(`CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION` (*class* `STANDARD-CLASS`) *name direct-slot-definitions*)
(`CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION` (*class* `CLOS:FUNCALLABLE-STANDARD-CLASS`) *name direct-slot-definitions*)

This method implements the inheritance and defaulting of slot options following the rules described in the [ANSI CL standard] section 7.5.3 “Inheritance of Slots and Options”.

This method can be extended, but the value returned by the extending method must be the value returned by this method.

Implementation dependent: only in CLISP

The initialization arguments that are passed to `CLOS:EFFECTIVE-SLOT-DEFINITION-CLASS` and `MAKE-INSTANCE` are computed through a call to `CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION-INITARGS`. It is the `CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION-INITARGS` method that implements the inheritance rules.

29.3.6.13. Generic Function CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION-INITARGS

Implementation dependent: only in CLISP

Syntax

(CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION-INITARGS *class direct-slot-definitions*)

Arguments

class
a class metaobject.

direct-slot-definitions
an ordered list of direct slot definition metaobjects. The most specific direct slot definition metaobject appears first in the list.

Value

A list of initialization arguments for an effective slot definition metaobject.

Purpose

This generic function determines the initialization arguments for the effective slot definition for a slot in a class. It is called by CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION. The resulting initialization arguments are passed to CLOS:EFFECTIVE-SLOT-DEFINITION-CLASS and then to MAKE-INSTANCE.

This generic function uses the supplied list of direct slot definition metaobjects to compute the inheritance of slot properties for a single slot. The returned effective slot definition initargs represent the result of computing the inheritance. The value of :NAME in the returned initargs is the same as the name of the direct slot definitions supplied.

Methods

(CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION-INITARGS (*class* STANDARD-CLASS) *direct-slot-definitions*)
(CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION-INITARGS (*class*

CLOS:FUNCALLABLE-STANDARD-CLASS) *direct-slot-definitions*)

This method implements the inheritance and defaulting of slot options following the rules described in the [ANSI CL standard] section 7.5.3 “Inheritance of Slots and Options”.

This method can be extended.

29.3.6.14. Generic Function `CLOS:EFFECTIVE-SLOT-DEFINITION-CLASS`

Syntax

`(CLOS:EFFECTIVE-SLOT-DEFINITION-CLASS` ***class*** `&REST` ***initargs***`)`

Arguments

class

a class metaobject.

initargs

set of initialization arguments and values.

Value

A subclass of the class `CLOS:EFFECTIVE-SLOT-DEFINITION-CLASS`.

Purpose

This generic function is called by `CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION` to determine the class of the resulting effective slot definition metaobject. The *`initargs`* argument is the set of initialization arguments and values that will be passed to `MAKE-INSTANCE` when the effective slot definition metaobject is created.

Methods

(CLOS:EFFECTIVE-SLOT-DEFINITION-CLASS (*class* STANDARD-CLASS) &REST *initargs*)

(CLOS:EFFECTIVE-SLOT-DEFINITION-CLASS (*class* CLOS:FUNCALLABLE-STANDARD-CLASS) &REST *initargs*)

These methods return the class `CLOS:STANDARD-EFFECTIVE-SLOT-DEFINITION`.

These methods can be overridden.

29.3.6.15. Generic Function `CLOS:COMPUTE-DEFAULT-INITARGS`

Syntax

`(CLOS:COMPUTE-DEFAULT-INITARGS class)`

Arguments

class
: a class metaobject.

Value

A list of canonicalized default initialization arguments.

Purpose

This generic-function is called to determine the default initialization arguments for a class.

The result is a list of canonicalized default initialization arguments, with no duplication among initialization argument names.

All methods on this generic function must compute the default initialization arguments as a function of only:

i. the class precedence list of *class*, and
ii. the direct default initialization arguments of each class in that list.

The results are undefined if the rules used to compute the default initialization arguments depend on any other factors.

When a class is finalized, `CLOS:FINALIZE-INHERITANCE` calls this generic function and associates the returned value with the class metaobject. The value can then be accessed by calling `CLOS:CLASS-DEFAULT-INITARGS`.

The list returned by this function will not be mutated by the implementation. The results are undefined if a portable program mutates the list returned by this function.

Methods

```
(CLOS:COMPUTE-DEFAULT-INITARGS (class STANDARD-CLASS))
(CLOS:COMPUTE-DEFAULT-INITARGS (class CLOS:FUNCALLABLE-STANDARD-CLASS))
```

These methods compute the default initialization arguments according to the rules described in the [ANSI CL standard] section 7.1.3 “Defaulting of Initialization Arguments”.
These methods signal an error if *class* or any of its superclasses is a forward referenced class.
These methods can be overridden.

29.3.7. Updating Dependencies

29.3.7.1. Generic Function CLOS:ADD-DIRECT-SUBCLASS

Syntax

`(CLOS:ADD-DIRECT-SUBCLASS superclass subclass)`

Arguments

superclass
a class metaobject.

subclass
a class metaobject.

Values

The values returned by this generic function is unspecified.

Purpose

This generic function is called to maintain a set of backpointers from a class to its direct subclasses. This generic function adds *subclass* to the set of direct subclasses of *superclass*.
When a class is initialized, this generic function is called once for each direct superclass of the class.
When a class is reinitialized, this generic function is called once for each added direct superclass of the class. The generic function CLOS:REMOVE-DIRECT-SUBCLASS is called once for each deleted direct superclass of the class.

Methods

(CLOS:ADD-DIRECT-SUBCLASS (*superclass* CLASS) (*subclass* CLASS))

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

- `CLOS:REMOVE-DIRECT-SUBCLASS (CLASS CLASS)`
- `CLOS:CLASS-DIRECT-SUBCLASSES (CLASS)`

29.3.7.2. Generic Function CLOS:REMOVE-DIRECT-SUBCLASS

Syntax

`(CLOS:REMOVE-DIRECT-SUBCLASS` *superclass* *subclass*`)`

Arguments

superclass

a class metaobject.

subclass

a class metaobject.

Values

The values returned by this generic function is unspecified.

Purpose

This generic function is called to maintain a set of backpointers from a class to its direct subclasses. It removes *subclass* from the set of direct subclasses of *superclass*. No `ERROR` is `SIGNAL`ed if *subclass* is not in this set.

Whenever a class is reinitialized, this generic function is called once with each deleted direct superclass of the class.

Methods

(CLOS:REMOVE-DIRECT-SUBCLASS (*superclass* CLASS) (*subclass* CLASS))

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

- `CLOS:ADD-DIRECT-SUBCLASS` (`CLASS` `CLASS`)
- `CLOS:CLASS-DIRECT-SUBCLASSES` (`CLASS`)

29.4. Slot Definitions

29.4.1. Inheritance Structure of slot definition metaobject Classes

Figure 29.3. Inheritance structure of slot definition metaobject classes

Inheritance structure of slot definition metaobject classes

29.4.2. Introspection: Readers for slot definition metaobjects

The reader generic functions which simply return information associated with slot definition metaobjects are presented together here in the format described in Section 29.3.3, “Introspection: Readers for class metaobjects”.

Each of the reader generic functions for slot definition metaobjects has the same syntax, accepting one required argument called `slot`, which must be a slot definition metaobject; otherwise, an `ERROR` is `SIGNAL`ed. An `ERROR` is also `SIGNAL`ed if the slot definition metaobject has not been initialized.

These generic functions can be called by the user or the implementation.

For any of these generic functions which returns a list, such lists will not be mutated by the implementation. The results are undefined if a portable program allows such a list to be mutated.

29.4.2.1. Generic Functions

29.4.2.1.1. Generic Function `CLOS:SLOT-DEFINITION-NAME`

(`CLOS:SLOT-DEFINITION-NAME` *`slot`*)

Returns the name of `slot`. This value is a symbol that can be used as a variable name. This is the value of the `:NAME` initialization argument that was associated with the slot definition metaobject during initialization.

Implementation dependent: only in CLISP

The slot name does not need to be usable as a variable name. Slot names like `NIL` or `T` are perfectly valid.

29.4.2.1.2. Generic Function `CLOS:SLOT-DEFINITION-ALLOCATION`

(`CLOS:SLOT-DEFINITION-ALLOCATION` *`slot`*)

Returns the allocation of *`slot`*. This is a symbol. This is the defaulted value of the `:ALLOCATION` initialization argument that was associated with the slot definition metaobject during initialization.

29.4.2.1.3. Generic Function `CLOS:SLOT-DEFINITION-INITFORM`

(`CLOS:SLOT-DEFINITION-INITFORM` *`slot`*)

Returns the initialization form of *`slot`*. This can be any form. This is the defaulted value of the `:INITFORM` initialization argument that was associated with the slot definition metaobject during initialization. When *`slot`* has no initialization form, the value returned is unspecified (however, `CLOS:SLOT-DEFINITION-INITFUNCTION` is guaranteed to return `NIL`).

29.4.2.1.4. Generic Function CLOS:SLOT-DEFINITION-INITFUNCTION

(CLOS:SLOT-DEFINITION-INITFUNCTION *slot*)

Returns the initialization function of *slot*. This value is either a function of no arguments, or NIL, indicating that the slot has no initialization function. This is the defaulted value of the :INITFUNCTION initialization argument that was associated with the slot definition metaobject during initialization.

29.4.2.1.5. Generic Function CLOS:SLOT-DEFINITION-TYPE

(CLOS:SLOT-DEFINITION-TYPE *slot*)

Returns the type of *slot*. This is a type specifier name. This is the defaulted value of the :TYPE initialization argument that was associated with the slot definition metaobject during initialization.

29.4.2.1.6. Generic Function CLOS:SLOT-DEFINITION-INITARGS

(CLOS:SLOT-DEFINITION-INITARGS *slot*)

Returns the set of initialization argument keywords for *slot*. This is the defaulted value of the :INITARGS initialization argument that was associated with the slot definition metaobject during initialization.

29.4.2.2. Methods

The specified methods for the slot definition metaobject readers

```
(CLOS:SLOT-DEFINITION-NAME (slot-definition
CLOS:STANDARD-SLOT-DEFINITION))
(CLOS:SLOT-DEFINITION-ALLOCATION (slot-definition
CLOS:STANDARD-SLOT-DEFINITION))
(CLOS:SLOT-DEFINITION-INITFORM (slot-definition
CLOS:STANDARD-SLOT-DEFINITION))
(CLOS:SLOT-DEFINITION-INITFUNCTION (slot-definition
CLOS:STANDARD-SLOT-DEFINITION))
(CLOS:SLOT-DEFINITION-TYPE (slot-definition
CLOS:STANDARD-SLOT-DEFINITION))
(CLOS:SLOT-DEFINITION-INITARGS (slot-definition
CLOS:STANDARD-SLOT-DEFINITION))
```

No behavior is specified for these methods beyond that which is specified for their respective generic functions.

29.4.2.3. Readers for direct slot definition metaobjects

The following additional reader generic functions are defined for direct slot definition metaobjects.

29.4.2.3.1. Generic Function `CLOS:SLOT-DEFINITION-READERS`

(`CLOS:SLOT-DEFINITION-READERS` *`direct-slot-definition`*)

Returns a (possibly empty) set of readers of the *`direct-slot-definition`*. This value is a list of function names. This is the defaulted value of the `:READERS` initialization argument that was associated with the direct slot definition metaobject during initialization.

29.4.2.3.2. Generic Function `CLOS:SLOT-DEFINITION-WRITERS`

(`CLOS:SLOT-DEFINITION-WRITERS` *`direct-slot-definition`*)

Returns a (possibly empty) set of writers of the *`direct-slot-definition`*. This value is a list of function names. This is the defaulted value of the `:WRITERS` initialization argument that was associated with the direct slot definition metaobject during initialization.

(`CLOS:SLOT-DEFINITION-READERS` (*`direct-slot-definition`* `CLOS:STANDARD-DIRECT-SLOT-DEFINITION`))
(`CLOS:SLOT-DEFINITION-WRITERS` (*`direct-slot-definition`* `CLOS:STANDARD-DIRECT-SLOT-DEFINITION`))

No behavior is specified for these methods beyond that which is specified for their respective generic functions.

29.4.2.4. Readers for effective slot definition metaobjects

The following reader generic function is defined for effective slot definition metaobjects.

29.4.2.4.1. Generic Function CLOS:SLOT-DEFINITION-LOCATION

(CLOS:SLOT-DEFINITION-LOCATION *effective-slot-definition*)

Returns the location of *effective-slot-definition*. The meaning and interpretation of this value is described in Section 29.10.1, "Instance Structure Protocol".

(CLOS:SLOT-DEFINITION-LOCATION (*effective-slot-definition* CLOS:STANDARD-EFFECTIVE-SLOT-DEFINITION))

This method returns the value stored by `CLOS:COMPUTE-SLOTS :AROUND (STANDARD-CLASS)` and `CLOS:COMPUTE-SLOTS :AROUND (CLOS:FUNCALLABLE-STANDARD-CLASS)`.

29.4.3. Initialization of slot definition metaobjects

A slot definition metaobject can be created by calling `MAKE-INSTANCE`. The initialization arguments establish the definition of the slot definition. A slot definition metaobject cannot be redefined; calling `REINITIALIZE-INSTANCE` `SIGNAL`s an `ERROR`.

Initialization of a slot definition metaobject must be done by calling `MAKE-INSTANCE` and allowing it to call `INITIALIZE-INSTANCE`. Portable programs must **not**...

- ... call `INITIALIZE-INSTANCE` directly to initialize a slot definition metaobject;
- ... call `SHARED-INITIALIZE` directly to initialize a slot definition metaobject;
- ... call `CHANGE-CLASS` to change the class of any slot definition metaobject or to turn a non-slot-definition object into a slot definition metaobject.

Since metaobject classes may not be redefined, no behavior is specified for the result of calls to `UPDATE-INSTANCE-FOR-REDEFINED-CLASS` on slot definition metaobjects. Since the class of a slot definition metaobject cannot be changed, no behavior is specified for the result of calls to `UPDATE-INSTANCE-FOR-DIFFERENT-CLASS` on slot definition metaobjects.

During initialization, each initialization argument is checked for errors and then associated with the slot definition metaobject. The value can then be accessed by calling the appropriate accessor as shown in Table 29.3, “Initialization arguments and accessors for slot definition metaobjects”.

This section begins with a description of the error checking and processing of each initialization argument. This is followed by a table showing the generic functions that can be used to access the stored initialization arguments.

In these descriptions, the phrase “this argument defaults to *`value`*” means that when that initialization argument is not supplied, initialization is performed as if *`value`* had been supplied. For some initialization arguments this could be done by the use of default initialization arguments, but whether it is done this way is not specified. Implementations are free to define default initialization arguments for specified slot definition metaobject classes. Portable programs are free to define default initialization arguments for portable subclasses of the class `CLOS:SLOT-DEFINITION`.

- The `:NAME` argument is a slot name. An `ERROR` is `SIGNAL`ed if this argument is not a symbol which can be used as a variable name. An `ERROR` is `SIGNAL`ed if this argument is not supplied.

Implementation dependent: only in CLISP

The `:NAME` argument does not need to be usable as a variable name. Slot names like `NIL` or `T` are perfectly valid.

- The `:INITFORM` argument is a form. The `:INITFORM` argument defaults to `NIL`. An `ERROR` is `SIGNAL`ed if the `:INITFORM` argument is supplied, but the `:INITFUNCTION` argument is not supplied.
- The `:INITFUNCTION` argument is a function of zero arguments which, when called, evaluates the `:INITFORM` in the appropriate lexical environment. The `:INITFUNCTION` argument defaults to false. An `ERROR` is `SIGNAL`ed if the `:INITFUNCTION` argument is supplied, but the `:INITFORM` argument is not supplied.
- The `:TYPE` argument is a type specifier name. An `ERROR` is `SIGNAL`ed otherwise. The `:TYPE` argument defaults to the symbol `T`.
- The `:ALLOCATION` argument is a `SYMBOL`. An `ERROR` is `SIGNAL`ed otherwise. The `:ALLOCATION` argument defaults to the symbol `:INSTANCE`.
- The `:INITARGS` argument is a `LIST` of `SYMBOL`s. An `ERROR` is `SIGNAL`ed if this argument is not a proper list, or if any element of this list is not a `SYMBOL`. The `:INITARGS` argument defaults to the empty list.
- The `:READERS` and `:WRITERS` arguments are `LIST`s of function names. An `ERROR` is `SIGNAL`ed if they are not proper lists, or if any element is not a valid function name. They default to the empty list. An `ERROR` is `SIGNAL`ed if either of these arguments is supplied and the metaobject is not a `CLOS:DIRECT-SLOT-DEFINITION`.
- The `:DOCUMENTATION` argument is a `STRING` or `NIL`. An `ERROR` is `SIGNAL`ed if it is not. This argument default to `NIL` during initialization.

After the processing and defaulting of initialization arguments described above, the value of each initialization argument is associated with the slot definition metaobject. These values can then be accessed by calling the corresponding generic function. The correspondences are as follows:

Table 29.3. Initialization arguments and accessors for slot definition metaobjects

Initialization Argument	Generic Function
:NAME	CLOS:SLOT-DEFINITION-NAME
:INITFORM	CLOS:SLOT-DEFINITION-INITFORM
:INITFUNCTION	CLOS:SLOT-DEFINITION-INITFUNCTION
:TYPE	CLOS:SLOT-DEFINITION-TYPE
:ALLOCATION	CLOS:SLOT-DEFINITION-ALLOCATION
:INITARGS	CLOS:SLOT-DEFINITION-INITARGS
:READERS	CLOS:SLOT-DEFINITION-READERS
:WRITERS	CLOS:SLOT-DEFINITION-WRITERS
:DOCUMENTATION	DOCUMENTATION

29.4.3.1. Methods

It is not specified which methods provide the initialization and reinitialization behavior described above. Instead, the information needed to allow portable programs to specialize this behavior is presented as a set of restrictions on the methods a portable program can define. The model is that portable initialization methods have access to the slot definition metaobject when either all or none of the specified initialization has taken effect.

These restrictions govern the methods that a portable program can define on the generic functions `INITIALIZE-INSTANCE`, `REINITIALIZE-INSTANCE`, and `SHARED-INITIALIZE`. These restrictions apply only to methods on these generic functions for which the first specializer is a subclass of the class `CLOS:SLOT-DEFINITION`. Other portable methods on these generic functions are not affected by these restrictions.

- Portable programs must not define methods on SHARED-INITIALIZE or REINITIALIZE-INSTANCE.
- For INITIALIZE-INSTANCE:
  - Portable programs must not define primary methods.
  - Portable programs may define around-methods, but these must be extending, not overriding methods.
  - Portable before-methods must assume that when they are run, none of the initialization behavior described above has been completed.
  - Portable after-methods must assume that when they are run, all of the initialization behavior described above has been completed.

The results are undefined if any of these restrictions are violated.

29.5. Generic Functions

29.5.1. Inheritance Structure of generic function metaobject Classes

Figure 29.4. Inheritance structure of generic function metaobject classes

Inheritance structure of generic function metaobject classes

29.5.2. Introspection: Readers for generic function metaobjects

The reader generic functions which simply return information associated with generic function metaobjects are presented together here in the format described in Section 29.3.3, "Introspection: Readers for class metaobjects".

Each of the reader generic functions for generic function metaobjects has the same syntax, accepting one required argument called *`generic-function`*, which must be a generic function metaobject; otherwise, an `ERROR` is `SIGNAL`ed. An `ERROR` is also `SIGNAL`ed if the generic function metaobject has not been initialized.

These generic functions can be called by the user or the implementation.

For any of these generic functions which returns a list, such lists will not be mutated by the implementation. The results are undefined if a portable program allows such a list to be mutated.

29.5.2.1. Generic Function `CLOS:GENERIC-FUNCTION-NAME`

(`CLOS:GENERIC-FUNCTION-NAME` *generic-function*)

Returns the name of the generic function, or `NIL` if the generic function has no name. This is the defaulted value of the `:NAME` initialization argument that was associated with the generic function metaobject during initialization or reinitialization. (See also `(SETF CLOS:GENERIC-FUNCTION-NAME)`.)

29.5.2.2. Generic Function `CLOS:GENERIC-FUNCTION-METHODS`

(`CLOS:GENERIC-FUNCTION-METHODS` *generic-function*)

Returns the set of methods currently connected to the generic function. This is a set of method metaobjects. This value is maintained by the generic functions `ADD-METHOD` and `REMOVE-METHOD`.

29.5.2.3. Generic Function `CLOS:GENERIC-FUNCTION-LAMBDA-LIST`

(`CLOS:GENERIC-FUNCTION-LAMBDA-LIST` *generic-function*)

Returns the lambda list of the generic function. This is the defaulted value of the `:LAMBDA-LIST` initialization argument that was associated with the generic function metaobject during initialization or reinitialization. An `ERROR` is `SIGNAL`ed if the lambda list has yet to be supplied.

29.5.2.4. Generic Function `CLOS:GENERIC-FUNCTION-ARGUMENT-PRECEDENCE-ORDER`

(`CLOS:GENERIC-FUNCTION-ARGUMENT-PRECEDENCE-ORDER` *generic-function*)

Returns the argument precedence order of the generic function. This value is a list of symbols, a permutation of the required parameters in the lambda list of the generic function. This is the defaulted value of the `:ARGUMENT-PRECEDENCE-ORDER` initialization argument that was associated with the generic function metaobject during initialization or reinitialization.

Implementation dependent: only in CLISP

An `ERROR` is `SIGNAL`ed if the lambda list has not yet been supplied.

29.5.2.5. Generic Function `CLOS:GENERIC-FUNCTION-DECLARATIONS`

(`CLOS:GENERIC-FUNCTION-DECLARATIONS` *generic-function*)

Returns a possibly empty list of the “declarations” of the generic function. The elements of this list are declaration specifiers. This list is the defaulted value of the `:DECLARATIONS` initialization argument that was associated with the generic function metaobject during initialization or reinitialization.

29.5.2.6. Generic Function `CLOS:GENERIC-FUNCTION-METHOD-CLASS`

(`CLOS:GENERIC-FUNCTION-METHOD-CLASS` *generic-function*)

Returns the default method class of the generic function. This class must be a subclass of the class `METHOD`. This is the defaulted value of the `:METHOD-CLASS` initialization argument that was associated with the generic function metaobject during initialization or reinitialization.

29.5.2.7. Generic Function `CLOS:GENERIC-FUNCTION-METHOD-COMBINATION`

(`CLOS:GENERIC-FUNCTION-METHOD-COMBINATION` *generic-function*)

Returns the method combination of the generic function. This is a method combination metaobject. This is the defaulted value of the `:METHOD-COMBINATION` initialization argument that was associated with the generic function metaobject during initialization or reinitialization.

29.5.2.8. Methods

The specified methods for the generic function metaobject reader generic functions

(`CLOS:GENERIC-FUNCTION-NAME` (*generic-function* `STANDARD-GENERIC-FUNCTION`))
(`CLOS:GENERIC-FUNCTION-LAMBDA-LIST` (*generic-function* `STANDARD-GENERIC-FUNCTION`))
(`CLOS:GENERIC-FUNCTION-ARGUMENT-PRECEDENCE-ORDER` (*generic-function* `STANDARD-GENERIC-FUNCTION`))
(`CLOS:GENERIC-FUNCTION-DECLARATIONS` (*generic-function* `STANDARD-GENERIC-FUNCTION`))
(`CLOS:GENERIC-FUNCTION-METHOD-CLASS` (*generic-function* `STANDARD-GENERIC-FUNCTION`))
(`CLOS:GENERIC-FUNCTION-METHOD-COMBINATION` (*generic-function* `STANDARD-GENERIC-FUNCTION`))

> No behavior is specified for these methods beyond that which is specified for their respective generic functions.

(`CLOS:GENERIC-FUNCTION-METHODS` (*generic-function* `STANDARD-GENERIC-FUNCTION`))

> No behavior is specified for this method beyond that which is specified for the generic function.

The value returned by this method is maintained by ADD-METHOD (STANDARD-GENERIC-FUNCTION STANDARD-METHOD) and REMOVE-METHOD(STANDARD-GENERIC-FUNCTION STANDARD-METHOD).

29.5.3. Initialization of Generic Functions

29.5.3.1. Macro DEFGENERIC

The evaluation or execution of a DEFGENERIC form results in a call to the ENSURE-GENERIC-FUNCTION function. The arguments received by ENSURE-GENERIC-FUNCTION are derived from the DEFGENERIC form in a defined way. As with DEFCLASS and DEFMETHOD, the exact macro-expansion of the DEFGENERIC form is not defined, only the relationship between the arguments to the macro and the arguments received by ENSURE-GENERIC-FUNCTION.

- The *function-name* argument to DEFGENERIC becomes the first argument to ENSURE-GENERIC-FUNCTION. This is the only positional argument accepted by ENSURE-GENERIC-FUNCTION; all other arguments are keyword arguments.
- The *lambda-list* argument to DEFGENERIC becomes the value of the :LAMBDA-LIST keyword argument to ENSURE-GENERIC-FUNCTION.

- For each of the options :ARGUMENT-PRECEDENCE-ORDER, :DOCUMENTATION, :GENERIC-FUNCTION-CLASS and :METHOD-CLASS, the value of the option becomes the value of the keyword argument with the same name. If the option does not appear in the macro form, the keyword argument does not appear in the resulting call to ENSURE-GENERIC-FUNCTION.

Implementation dependent: only in CLISP

If the option does not appear in the macro form, the keyword argument appears in the resulting call to `ENSURE-GENERIC-FUNCTION`, with a default value: the *`lambda list`* for `:ARGUMENT-PRECEDENCE-ORDER`, `NIL` for `:DOCUMENTATION`, the class `STANDARD-GENERIC-FUNCTION` for `:GENERIC-FUNCTION-CLASS`, the class `STANDARD-METHOD` for `:METHOD-CLASS`. This is needed to make the generic function reflect the `DEFGENERIC` form.

- For the option `:DECLARE`, the list of “declarations” becomes the value of the `:DECLARATIONS` keyword argument. If the `:DECLARE` option does not appear in the macro form, the `:DECLARATIONS` keyword argument does not appear in the call to `ENSURE-GENERIC-FUNCTION`.

Implementation dependent: only in CLISP

If the `:DECLARE` option does not appear in the macro form, the `:DECLARATIONS` keyword argument appears in the resulting call to `ENSURE-GENERIC-FUNCTION`, with a default value of `NIL`. This is needed to make the generic function reflect the `DEFGENERIC` form.

- The handling of the `:METHOD-COMBINATION` option is not specified.

Implementation dependent: only in CLISP

If the `:METHOD-COMBINATION` option does not appear in the macro form, the `:METHOD-COMBINATION` keyword argument still appears in the resulting call to `ENSURE-`

GENERIC-FUNCTION, but in a position where it can be overridden by user-defined initargs and default initargs.

- ## Implementation dependent: only in CLISP

The :DECLARE keyword is recognized as equivalent to the :DECLARATIONS keyword, for compatibility with ENSURE-GENERIC-FUNCTION in [ANSI CL standard]. If both :DECLARE and :DECLARATIONS keyword arguments are specified, an ERROR is SIGNALed.

Any other generic function options become the value of keyword arguments with the same name. The value of the keyword argument is the tail of the generic function option. An ERROR is SIGNALed if any generic function option appears more than once in the DEFGENERIC form.

The default initargs of the *generic-function-class* are added at the end of the list of arguments to pass to ENSURE-GENERIC-FUNCTION. This is needed to make the generic function reflect the DEFGENERIC form.

- ## Implementation dependent: only in CLISP

User-defined options. Any other options become the value of keyword arguments with the same name. The value of the keyword argument is the tail of the option. An ERROR is SIGNALed if any option appears more than once in the DEFGENERIC form.

The result of the call to ENSURE-GENERIC-FUNCTION is returned as the result of evaluating or executing the DEFGENERIC form.

29.5.3.2. Generic Function Invocation Protocol

Associated with each generic function is its discriminating function. Each time the generic function is called, the discriminating function is called to provide the behavior of the generic function. The discriminating function receives the full set of arguments received by the generic function. It must lookup and execute the appropriate methods, and return the appropriate values.

The discriminating function is computed by the highest layer of the generic function invocation protocol, `CLOS:COMPUTE-DISCRIMINATING-FUNCTION`. Whenever a generic function metaobject is initialized, reinitialized, or a method is added or removed, the discriminating function is recomputed. The new discriminating function is then stored with `CLOS:SET-FUNCALLABLE-INSTANCE-FUNCTION`.

Discriminating functions call `CLOS:COMPUTE-APPLICABLE-METHODS` and `CLOS:COMPUTE-APPLICABLE-METHODS-USING-CLASSES` to compute the methods applicable to the generic functions arguments. Applicable methods are combined by `CLOS:COMPUTE-EFFECTIVE-METHOD` to produce an ***effective method***. Provisions are made to allow memoization of the method applicability and effective methods computations. (See the description of `CLOS:COMPUTE-DISCRIMINATING-FUNCTION` for details.)

The body of method definitions are processed by `CLOS:MAKE-METHOD-LAMBDA`. The result of this generic function is a lambda expression which is processed by either `COMPILE` or `COMPILE-FILE` to produce a ***method function***. The arguments received by the method function are controlled by the `CALL-METHOD` forms appearing in the effective methods. By default, method functions accept two arguments: a list of arguments to the generic function, and a list of next methods. The list of next methods corresponds to the next methods argument to `CALL-METHOD`. If `CALL-METHOD` appears with additional arguments, these will be passed to the method functions as well; in these cases, `CLOS:MAKE-METHOD-LAMBDA` must have created the method lambdas to expect additional arguments.

Implementation dependent: only in CLISP

See The generic function CLOS:MAKE-METHOD-LAMBDA is not implemented.

See Method function arguments.

29.5.3.3. Initialization of generic function metaobjects

29.5.3.3.1. Methods

A generic function metaobject can be created by calling MAKE-INSTANCE. The initialization arguments establish the definition of the generic function. A generic function metaobject can be redefined by calling REINITIALIZE-INSTANCE. Some classes of generic function metaobject do not support redefinition; in these cases, REINITIALIZE-INSTANCE SIGNALs an ERROR.

Initialization of a generic function metaobject must be done by calling MAKE-INSTANCE and allowing it to call INITIALIZE-INSTANCE. Reinitialization of a generic-function metaobject must be done by calling REINITIALIZE-INSTANCE. Portable programs must **not**

- ... call INITIALIZE-INSTANCE directly to initialize a generic function metaobject;
- ... call SHARED-INITIALIZE directly to initialize or reinitialize a generic function metaobject;
- ... call CHANGE-CLASS to change the class of any generic function metaobject or to turn a non-generic-function object into a generic function metaobject.

Since metaobject classes may not be redefined, no behavior is specified for the result of calls to UPDATE-INSTANCE-FOR-REDEFINED-CLASS on generic function metaobjects. Since the class of a generic function metaobject may not be changed, no behavior is specified for the results of calls to UPDATE-INSTANCE-FOR-DIFFERENT-CLASS on generic function metaobjects.

During initialization or reinitialization, each initialization argument is checked for errors and then associated with the generic function metaobject. The value can then be accessed by calling the appropriate accessor as shown in Table 29.4, "Initialization arguments and accessors for generic function metaobjects".

This section begins with a description of the error checking and processing of each initialization argument. This is followed by a table showing the generic functions that can be used to access the stored initialization arguments. The section ends with a set of restrictions on portable methods affecting generic function metaobject initialization and reinitialization.

In these descriptions, the phrase "this argument defaults to *value*" means that when that initialization argument is not supplied, initialization or reinitialization is performed as if *value* had been supplied. For some initialization arguments this could be done by the use of default initialization arguments, but whether it is done this way is not specified. Implementations are free to define default initialization arguments for specified generic function metaobject classes. Portable programs are free to define default initialization arguments for portable subclasses of the class GENERIC-FUNCTION.

Unless there is a specific note to the contrary, then during reinitialization, if an initialization argument is not supplied, the previously stored value is left unchanged.

- The :ARGUMENT-PRECEDENCE-ORDER argument is a list of symbols.

  An ERROR is SIGNALed if this argument appears but the :LAMBDA-LIST argument does not appear. An ERROR is SIGNALed if this value is not a proper list or if this value is not a permutation of the symbols from the required arguments part of the :LAMBDA-LIST initialization argument.

  When the generic function is being initialized or reinitialized, and this argument is not supplied, but the :LAMBDA-LIST argument is supplied, this value defaults to the symbols from the required arguments part of the :LAMBDA-LIST argument, in the order they appear in that argument. If neither argument is supplied, neither are initialized (see the description of :LAMBDA-LIST.)

- The :DECLARATIONS argument is a list of declaration specifiers.

  An ERROR is SIGNALed if this value is not a proper list or if each of its elements is not a legal declaration specifier.

  When the generic function is being initialized, and this argument is not supplied, it defaults to the empty list.

- The :DOCUMENTATION argument is a STRING or NIL. An ERROR is SIGNALed if it is not. This argument default to NIL during initialization.

- The :LAMBDA-LIST argument is a lambda list.

  An ERROR is SIGNALed if this value is not a proper generic function lambda list.

  When the generic function is being initialized, and this argument is not supplied, the generic function's lambda list is not initialized. The lambda list will be initialized later, either when the first method is added to the generic function, or a later reinitialization of the generic function.

- The :METHOD-COMBINATION argument is a method combination metaobject.

- The :METHOD-CLASS argument is a class metaobject.

  An ERROR is SIGNALed if this value is not a subclass of the class METHOD.

  When the generic function is being initialized, and this argument is not supplied, it defaults to the class STANDARD-METHOD.

- The :NAME argument is an object.

  If the generic function is being initialized, this argument defaults to NIL.

After the processing and defaulting of initialization arguments described above, the value of each initialization argument is associated with the generic function metaobject. These values can then be accessed by calling the corresponding generic function. The correspondences are as follows:

Table 29.4. Initialization arguments and accessors for generic function metaobjects

Initialization Argument	Generic Function
:ARGUMENT-PRECEDENCE-ORDER	CLOS:GENERIC-FUNCTION-ARGUMENT-PRECEDENCE-ORDER
:DECLARATIONS	CLOS:GENERIC-FUNCTION-DECLARATIONS
:DOCUMENTATION	DOCUMENTATION
:LAMBDA-LIST	CLOS:GENERIC-FUNCTION-LAMBDA-LIST
:METHOD-COMBINATION	CLOS:GENERIC-FUNCTION-METHOD-COMBINATION
:METHOD-CLASS	CLOS:GENERIC-FUNCTION-METHOD-CLASS
:NAME	CLOS:GENERIC-FUNCTION-NAME

29.5.3.3.1. Methods

It is not specified which methods provide the initialization and reinitialization behavior described above. Instead, the information needed to allow portable programs to specialize this behavior is presented as a set of restrictions on the methods a portable program can define. The model is that portable initialization methods have access to the generic function metaobject when either all or none of the specified initialization has taken effect.

These restrictions govern the methods that a portable program can define on the generic functions INITIALIZE-INSTANCE, REINITIALIZE-INSTANCE, and SHARED-INITIALIZE. These restrictions apply only to methods on these generic functions for which the first specializer is a subclass of the class `GENERIC-FUNCTION`. Other portable methods on these generic functions are not affected by these restrictions.

- Portable programs must not define methods on `SHARED-INITIALIZE`.
- For `INITIALIZE-INSTANCE` and `REINITIALIZE-INSTANCE`:
    - Portable programs must not define primary methods.
    - Portable programs may define around-methods, but these must be extending, not overriding methods.
    - Portable before-methods must assume that when they are run, none of the initialization behavior described above has been completed.
    - Portable after-methods must assume that when they are run, all of the initialization behavior described above has been completed.

The results are undefined if any of these restrictions are violated.

29.5.4. Customization

29.5.4.1. Generic Function (SETF CLOS:GENERIC-FUNCTION-NAME)

Syntax

((SETF CLOS:GENERIC-FUNCTION-NAME) *new-name generic-function*)

Arguments

generic-function
: a generic function metaobject.

new-name
: a function name or NIL.

Value

The *new-name* argument.

Purpose

This function changes the name of *generic-function* to *new-name*. This value is usually a function name or NIL, if the generic function is to have no name.

This function works by calling REINITIALIZE-INSTANCE with *generic-function* as its first argument, the symbol :NAME as its second argument and *new-name* as its third argument.

29.5.4.2. Generic Function ENSURE-GENERIC-FUNCTION

Syntax

(ENSURE-GENERIC-FUNCTION *function-name* &KEY &ALLOW-OTHER-KEYS)

Arguments

function-name
: a function name

keyword arguments
: Some of the keyword arguments accepted by this function are actually processed by CLOS:ENSURE-GENERIC-FUNCTION-USING-CLASS, others are processed during initialization of the

generic function metaobject (as described in Section 29.5.3.3, "Initialization of generic function metaobjects").

Value

A generic function metaobject.

Purpose

This function is called to define a globally named generic function or to specify or modify options and declarations that pertain to a globally named generic function as a whole. It can be called by the user or the implementation.

It is the functional equivalent of `DEFGENERIC`, and is called by the expansion of the `DEFGENERIC` and `DEFMETHOD` macros.

The behavior of this function is actually implemented by the generic function `CLOS:ENSURE-GENERIC-FUNCTION-USING-CLASS`. When `ENSURE-GENERIC-FUNCTION` is called, it immediately calls `CLOS:ENSURE-GENERIC-FUNCTION-USING-CLASS` and returns that result as its own.

The first argument to `CLOS:ENSURE-GENERIC-FUNCTION-USING-CLASS` is computed as follows:

- If *`function-name`* names a non-generic function, a macro, or a special form, an `ERROR` is `SIGNAL`ed.
- If *`function-name`* names a generic function, that generic function metaobject is used.
- Otherwise, `NIL` is used.

The second argument is *`function-name`*. The remaining arguments are the complete set of keyword arguments received by `ENSURE-GENERIC-FUNCTION`.

29.5.4.3. Generic Function `CLOS:ENSURE-GENERIC-FUNCTION-USING-CLASS`

Syntax

(`CLOS:ENSURE-GENERIC-FUNCTION-USING-CLASS` *`generic-function`* *`function-name`* `&KEY` `:ARGUMENT-PRECEDENCE-ORDER` `:DECLARATIONS` `:DOCUMENTATION` `:GENERIC-FUNCTION-CLASS` `:LAMBDA-LIST` `:METHOD-CLASS` `:METHOD-COMBINATION` `:NAME` `&ALLOW-OTHER-KEYS`)

Arguments

`generic-function`
: a generic function metaobject or `NIL`.

`function-name`
: a function name

`:GENERIC-FUNCTION-CLASS`
: a class metaobject or a class name. If it is not supplied, it defaults to the class named `STANDARD-GENERIC-FUNCTION`. If a class name is supplied, it is interpreted as the class with that name. If a class name is supplied, but there is no such class, an `ERROR` is `SIGNAL`ed.

additional keyword arguments
: see Section 29.5.3.3, "Initialization of generic function metaobjects".

Implementation dependent: only in CLISP

The `:DECLARE` keyword is recognized as equivalent to the `:DECLARATIONS` keyword, for compatibility with `ENSURE-GENERIC-FUNCTION` in [ANSI CL standard].

Value

A generic function metaobject.

Purpose

The generic function `CLOS:ENSURE-GENERIC-FUNCTION-USING-CLASS` is called to define or modify the definition of a globally named generic function. It is called by the `ENSURE-GENERIC-FUNCTION` function. It can also be called directly.

The first step performed by this generic function is to compute the set of initialization arguments which will be used to create or reinitialize the globally named generic function. These initialization arguments are computed from the full set of keyword arguments received by this generic function as follows:

- The `:GENERIC-FUNCTION-CLASS` argument is not included in the initialization arguments.

- If the :METHOD-CLASS argument was received by this generic function, it is converted into a class metaobject. This is done by looking up the class name with FIND-CLASS. If there is no such class, an ERROR is SIGNALed.
- All other keyword arguments are included directly in the initialization arguments.

If the *generic-function* argument is NIL, an instance of the class specified by the :GENERIC-FUNCTION-CLASS argument is created by calling MAKE-INSTANCE with the previously computed initialization arguments. The function name *function-name* is set to name the generic function. The newly created generic function metaobject is returned.
If the class of the *generic-function* argument is not the same as the class specified by the :GENERIC-FUNCTION-CLASS argument, an ERROR is SIGNALed.

Implementation dependent: only in CLISP

The description of ENSURE-GENERIC-FUNCTION in [ANSI CL standard] specifies that in this case, CHANGE-CLASS is called if the class of the *generic-function* argument and the class specified by the :GENERIC-FUNCTION-CLASS argument are compatible. Given the description of ENSURE-GENERIC-FUNCTION, this also applies to the CLOS:ENSURE-GENERIC-FUNCTION-USING-CLASS function. **CLISP**'s implementation calls CHANGE-CLASS always, and leaves it to the CHANGE-CLASS function to signal an error if needed.

Otherwise the generic function *generic-function* is redefined by calling the REINITIALIZE-INSTANCE generic function with *generic-function* and the initialization arguments. The *generic-function* argument is then returned.

Methods

(CLOS:ENSURE-GENERIC-FUNCTION-USING-CLASS (*generic-function* GENERIC-FUNCTION) *function-name* &KEY :GENERIC-FUNCTION-CLASS &ALLOW-OTHER-KEYS)

This method implements the behavior of the generic function in the case where `function-name` names an existing generic function.

This method can be overridden.

(CLOS:ENSURE-GENERIC-FUNCTION-USING-CLASS (*generic-function* NULL) *function-name* &KEY :GENERIC-FUNCTION-CLASS &ALLOW-OTHER-KEYS)

This method implements the behavior of the generic function in the case where `function-name` names no function, generic function, macro or special form.

29.5.4.4. Generic Function ADD-METHOD

Syntax

`(ADD-METHOD generic-function method)`

Arguments

generic-function

a generic function metaobject.

method

a method metaobject.

Value

The `generic-function` argument.

Purpose

This generic function associates an unattached method with a generic function.

An `ERROR` is `SIGNAL`ed if the lambda list of the method is not congruent with the lambda list of the generic function.

An `ERROR` is `SIGNAL`ed if the method is already associated with some other generic function.

If the given method agrees with an existing method of the generic function on parameter specializers and qualifiers, the existing method is removed by calling `REMOVE-METHOD` before the new method is added. See the [ANSI CL standard] section 7.6.3 “Agreement on Parameter Specializers and Qualifiers” for a definition of agreement in this context.

Associating the method with the generic function then proceeds in four steps:

i. add *`method`* to the set returned by `CLOS:GENERIC-FUNCTION-METHODS` and arrange for `CLOS:METHOD-GENERIC-FUNCTION` to return *`generic-function`*;
ii. call `CLOS:ADD-DIRECT-METHOD` for each of the method's specializers;
iii. call `CLOS:COMPUTE-DISCRIMINATING-FUNCTION` and install its result with `CLOS:SET-FUNCALLABLE-INSTANCE-FUNCTION`; and
iv. update the dependents of the generic function.

The generic function `ADD-METHOD` can be called by the user or the implementation.

Methods

`(ADD-METHOD (`*`generic-function`*` STANDARD-GENERIC-FUNCTION) (`*`method`*` STANDARD-METHOD))`
: No behavior is specified for this method beyond that which is specified for the generic function.

`(ADD-METHOD (`*`generic-function`*` STANDARD-GENERIC-FUNCTION) (`*`method`*` METHOD))`
: This method is specified by [ANSI CL standard].

29.5.4.5. Generic Function `REMOVE-METHOD`

Syntax

`(REMOVE-METHOD `*`generic-function method`*`)`

Arguments

`generic-function`
: a generic function metaobject.

`method`
: a method metaobject.

Value

The *`generic-function`* argument.

Purpose

This generic function breaks the association between a generic function and one of its methods.
No `ERROR` is `SIGNAL`ed if the method is not among the methods of the generic function.
Breaking the association between the method and the generic function proceeds in four steps:

i. remove *`method`* from the set returned by `CLOS:GENERIC-FUNCTION-METHODS` and arrange for `CLOS:METHOD-GENERIC-FUNCTION` to return `NIL`;
ii. call `CLOS:REMOVE-DIRECT-METHOD` for each of the method's specializers;
iii. call `CLOS:COMPUTE-DISCRIMINATING-FUNCTION` and install its result with `CLOS:SET-FUNCALLABLE-INSTANCE-FUNCTION`; and
iv. update the dependents of the generic function.

The generic function `REMOVE-METHOD` can be called by the user or the implementation.

Methods

`(REMOVE-METHOD (`*`generic-function`*` STANDARD-GENERIC-FUNCTION) (`*`method`*` STANDARD-METHOD))`
No behavior is specified for this method beyond that which is specified for the generic function.

`(REMOVE-METHOD (`*`generic-function`*` STANDARD-GENERIC-FUNCTION) (`*`method`*` METHOD))`
This method is specified by [ANSI CL standard].

29.5.4.6. Generic Function `CLOS:COMPUTE-APPLICABLE-METHODS`

Syntax

`(CLOS:COMPUTE-APPLICABLE-METHODS` *`generic-function arguments`*`)`

Arguments

`generic-function`

a generic function metaobject.

`arguments`

a list of objects.

Value

A possibly empty list of method metaobjects.

Purpose

This generic function determines the method applicability of a generic function given a list of required arguments. The returned list of method metaobjects is sorted by precedence order with the most specific method appearing first. If no methods are applicable to the supplied arguments the empty list is returned.

When a generic function is invoked, the discriminating function must determine the ordered list of methods applicable to the arguments. Depending on the generic function and the arguments, this is done in one of three ways: using a memoized value; calling `CLOS:COMPUTE-APPLICABLE-METHODS-USING-CLASSES`; or calling `CLOS:COMPUTE-APPLICABLE-METHODS`. (Refer to the description of `CLOS:COMPUTE-DISCRIMINATING-FUNCTION` for the details of this process.)

The *`arguments`* argument is permitted to contain more elements than the generic function accepts required arguments; in these cases the extra arguments will be ignored. An `ERROR` is `SIGNAL`ed if *`arguments`* contains fewer elements than the generic function accepts required arguments.

The list returned by this function will not be mutated by the implementation. The results are undefined if a portable program mutates the list returned by this function.

Methods

`(CLOS:COMPUTE-APPLICABLE-METHODS (`*`generic-function`*` STANDARD-GENERIC-FUNCTION) `*`arguments`*`)`

This method `SIGNAL`s an `ERROR` if any method of the generic function has a specializer which is neither a class metaobject nor an `EQL` specializer metaobject.

Otherwise, this method computes the sorted list of applicable methods according to the rules described in the [ANSI CL standard] section 7.6.6 “Method Selection and Combination”

This method can be overridden. Because of the consistency requirements between this generic function and `CLOS:COMPUTE-`

APPLICABLE-METHODS-USING-CLASSES, doing so may require also overriding CLOS:COMPUTE-APPLICABLE-METHODS-USING-CLASSES (STANDARD-GENERIC-FUNCTION T).

Remarks. See also the [ANSI CL standard] function COMPUTE-APPLICABLE-METHODS.

29.5.4.7. Generic Function CLOS:COMPUTE-APPLICABLE-METHODS-USING-CLASSES

Syntax

(CLOS:COMPUTE-APPLICABLE-METHODS-USING-CLASSES *generic-function classes*)

Arguments

generic-function
a generic function metaobject.

classes
a list of class metaobjects.

Values

1. A possibly empty list of method metaobjects.
2. BOOLEAN

Purpose

This generic function is called to attempt to determine the method applicability of a generic function given only the classes of the required arguments.

If it is possible to completely determine the ordered list of applicable methods based only on the supplied classes, this generic function returns that list as its primary value and true as its second value. The returned list of method metaobjects is sorted by precedence order, the most specific method coming first. If no methods are applicable to arguments with the specified classes, the empty list and true are returned.

If it is not possible to completely determine the ordered list of applicable methods based only on the supplied classes, this generic function returns an unspecified primary value and false as its second value.

When a generic function is invoked, the discriminating function must determine the ordered list of methods applicable to the arguments. Depending on the generic function and the arguments, this is done in one of three ways: using a memoized value; calling `CLOS:COMPUTE-APPLICABLE-METHODS-USING-CLASSES`; or calling `CLOS:COMPUTE-APPLICABLE-METHODS`. (Refer to the description of `CLOS:COMPUTE-DISCRIMINATING-FUNCTION` for the details of this process.)

The following consistency relationship between `CLOS:COMPUTE-APPLICABLE-METHODS-USING-CLASSES` and `CLOS:COMPUTE-APPLICABLE-METHODS` must be maintained: for any given generic function and set of arguments, if `CLOS:COMPUTE-APPLICABLE-METHODS-USING-CLASSES` returns a second value of true, the primary value must be equal to the value that would be returned by a corresponding call to `CLOS:COMPUTE-APPLICABLE-METHODS`. The results are undefined if a portable method on either of these generic functions causes this consistency to be violated.

The list returned by this function will not be mutated by the implementation. The results are undefined if a portable program mutates the list returned by this function.

Methods

(`CLOS:COMPUTE-APPLICABLE-METHODS-USING-CLASSES` (*generic-function* `STANDARD-GENERIC-FUNCTION`) *classes*)

If any method of the generic function has a specializer which is neither a class metaobject nor an `EQL` specializer metaobject, this method `SIGNAL`s an `ERROR`.

In cases where the generic function has no methods with `EQL` specializers, or has no methods with `EQL` specializers that could be applicable to arguments of the supplied classes, this method returns the ordered list of applicable methods as its first value and true as its second value.

Otherwise this method returns an unspecified primary value and false as its second value.

This method can be overridden. Because of the consistency requirements between this generic function and `CLOS:COMPUTE-APPLICABLE-METHODS`, doing so may require also overriding `CLOS:COMPUTE-APPLICABLE-METHODS (STANDARD-GENERIC-FUNCTION T)`.

Remarks

This generic function exists to allow user extensions which alter method lookup rules, but which base the new rules only on the classes of the required arguments, to take advantage of the class-based method lookup memoization found in many implementations. (There is of course no requirement for an implementation to provide this optimization.)

Such an extension can be implemented by two methods, one on this generic function and one on `CLOS:COMPUTE-APPLICABLE-METHODS`. Whenever the user extension is in effect, the first method will return a second value of true. This should allow the implementation to absorb these cases into its own memoization scheme.

To get appropriate performance, other kinds of extensions may require methods on `CLOS:COMPUTE-DISCRIMINATING-FUNCTION` which implement their own memoization scheme.

29.5.4.8. Generic Function `CLOS:COMPUTE-EFFECTIVE-METHOD`

Syntax

(`CLOS:COMPUTE-EFFECTIVE-METHOD` *generic-function method-combination methods*)

Arguments

generic-function
: a generic function metaobject.

method-combination
: a method combination metaobject.

methods
: a list of method metaobjects.

Values

1. An effective method
2. A list of effective method options

Purpose

This generic function is called to determine the effective method from a sorted list of method metaobjects.
An effective method is a form that describes how the applicable methods are to be combined. Inside of effective method forms are `CALL-METHOD` forms which indicate that a particular method is to be called. The arguments to the `CALL-METHOD` form indicate exactly how the method function of the method should be called. (See `CLOS:MAKE-METHOD-LAMBDA` for more details about method functions.)
An effective method option has the same interpretation and syntax as either the `:ARGUMENTS` or the `:GENERIC-FUNCTION` option in the long form of `DEFINE-METHOD-COMBINATION`.
More information about the form and interpretation of effective methods and effective method options can be found under the description of the `DEFINE-METHOD-COMBINATION` macro in the **CLOS** specification.
This generic function can be called by the user or the implementation. It is called by discriminating functions whenever a sorted list of applicable methods must be converted to an effective method.

Methods

(`CLOS:COMPUTE-EFFECTIVE-METHOD` (*`generic-function`* `STANDARD-GENERIC-FUNCTION`) *`method-combination methods`*)
This method computes the effective method according to the rules of the method combination type implemented by *`method-combination`*.
This method can be overridden.

Implementation dependent: only in CLISP

The second return value may contain only one `:ARGUMENTS` option and only one `:GENERIC-FUNCTION` option. When overriding a `CLOS:COMPUTE-EFFECTIVE-METHOD` method, before adding an `:ARGUMENTS` or `:GENERIC-FUNCTION` option, you therefore need to check whether it this option is already present.

29.5.4.9. Function CLOS:COMPUTE-EFFECTIVE-METHOD-AS-FUNCTION

Implementation dependent: only in CLISP

Syntax

`(CLOS:COMPUTE-EFFECTIVE-METHOD-AS-FUNCTION generic-function methods arguments)`

Arguments

generic-function
: a generic function metaobject.

methods
: a list of method metaobjects.

arguments
: a list of arguments.

Value

The effective method as a function, accepting any set of arguments for which all of the given methods are applicable.

Purpose

This function is called to determine the effective method from a sorted list of method metaobjects, and convert it to a function. The *arguments* are a set of arguments to which the methods are applicable, and are used solely for error message purposes.

This function calls `CLOS:COMPUTE-EFFECTIVE-METHOD` using the *generic-function*'s method combination, wraps local macro definitions for `CALL-METHOD` and `MAKE-METHOD` around it, handles the `:ARGUMENTS` and `:GENERIC-FUNCTION` options, and compiles the resulting form to a function.

29.5.4.10. Generic Function CLOS:MAKE-METHOD-LAMBDA

Syntax

`(CLOS:MAKE-METHOD-LAMBDA` ***`generic-function method lambda-expression environment`***`)`

Arguments

`generic-function`
: a generic function metaobject.

`method`
: a (possibly uninitialized) method metaobject.

`lambda-expression`
: a lambda expression.

`environment`
: the same as the `&ENVIRONMENT` argument to macro expansion functions.

Values

1. A lambda expression
2. A list of initialization arguments and values

Purpose

This generic function is called to produce a lambda expression which can itself be used to produce a method function for a method and generic function with the specified classes. The generic function and method the method function will be used with are not required to be the given ones. Moreover, the method metaobject may be uninitialized.

Either the function `COMPILE`, the special form `FUNCTION` or the function `COERCE` must be used to convert the lambda expression a method function. The method function itself can be applied to arguments with `APPLY` or `FUNCALL`.

When a method is actually called by an effective method, its first argument will be a list of the arguments to the generic function. Its remaining arguments will be all but the first argument passed to `CALL-METHOD`. By default, all method functions must accept two arguments: the list of arguments to the generic function and the list of next methods.

For a given generic function and method class, the applicable methods on `CLOS:MAKE-METHOD-LAMBDA` and `CLOS:COMPUTE-EFFECTIVE-METHOD` must be consistent in the following way: each use of `CALL-METHOD` returned by the method on `CLOS:COMPUTE-EFFECTIVE-METHOD` must have the same number of arguments, and

the method lambda returned by the method on CLOS:MAKE-METHOD-LAMBDA must accept a corresponding number of arguments.
Note that the system-supplied implementation of CALL-NEXT-METHOD is not required to handle extra arguments to the method function. Users who define additional arguments to the method function must either redefine or forego CALL-NEXT-METHOD. (See the example below.)
When the method metaobject is created with MAKE-INSTANCE, the method function must be the value of the :FUNCTION initialization argument. The additional initialization arguments, returned as the second value of this generic function, must also be passed in this call to MAKE-INSTANCE.

Methods

(CLOS:MAKE-METHOD-LAMBDA (*generic-function* STANDARD-GENERIC-FUNCTION) (*method* STANDARD-METHOD) *lambda-expression* *environment*)

This method returns a method lambda which accepts two arguments, the list of arguments to the generic function, and the list of next methods. What initialization arguments may be returned in the second value are unspecified.
This method can be overridden.

This example shows how to define a kind of method which, from within the body of the method, has access to the actual method metaobject for the method. This simplified code overrides whatever method combination is specified for the generic function, implementing a simple method combination supporting only primary methods, CALL-NEXT-METHOD and NEXT-METHOD-P. (In addition, its a simplified version of CALL-NEXT-METHOD which does no error checking.)

Notice that the extra lexical function bindings get wrapped around the body before CALL-NEXT-METHOD is called. In this way, the user's definition of CALL-NEXT-METHOD and NEXT-METHOD-P are sure to override the system's definitions.

```
(defclass my-generic-function (standard-generic-function)
  ()
  (:default-initargs :method-class (find-class 'my-method
```

```
(defclass my-method (standard-method) ())

(defmethod make-method-lambda ((gf my-generic-function)
                               (method my-method)
                               lambda-expression
                               environment)
  (declare (ignore environment))
  `(lambda (args next-methods this-method)
     (,(call-next-method gf method
         `(lambda ,(cadr lambda-expression)
            (flet ((this-method () this-method)
                   (call-next-method (&REST cnm-args)
                     (funcall (method-function (car next-r
                              (or cnm-args args)
                              (cdr next-methods)
                              (car next-methods)))
                   (next-method-p ()
                     (not (null next-methods))))
              ,@(cddr lambda-expression)))
         environment)
       args next-methods)))

(defmethod compute-effective-method ((gf my-generic-funct
                                     method-combination
                                     methods)
  `(call-method ,(car methods) ,(cdr methods) ,(car metho
```

Implementation dependent: only in CLISP

The generic function CLOS:MAKE-METHOD-LAMBDA is not implemented. Its specification is misdesigned: it mixes compile time and execution time behaviour. The essential problem is: where could the generic-function argument come from?

- If a DEFMETHOD form occurs in a source file, is CLOS:MAKE-METHOD-LAMBDA then called at compile time or at load time? If it was called at compile time, there's no possible value for the first argument, since the class of the generic function to which the method will belong is not known until load time. If it was called at load time, it would mean that the method's source code could only be compiled at load time, not earlier - which defeats the purpose of `COMPILE-FILE`

- When a method is removed from a generic function using `REMOVE-METHOD` and then added through `ADD-METHOD` to a different generic function, possibly belonging to a different generic function class, would `CLOS:MAKE-METHOD-LAMBDA` then be called again or not? If no, then `CLOS:MAKE-METHOD-LAMBDA`'s first argument is useless. If yes, then the source code of every method would have to be present at runtime, and its lexical environment as well.

Method function arguments.

- `CALL-METHOD` always expect exactly two arguments: the method and a list of next methods.
- Method functions always expect exactly two arguments: the list of arguments passed to the generic function, and the list of next methods.

29.5.4.11. Generic Function `CLOS:COMPUTE-DISCRIMINATING-FUNCTION`

Syntax

(`CLOS:COMPUTE-DISCRIMINATING-FUNCTION` *`generic-function`*)

Arguments

`generic-function`

a generic function metaobject.

Value

A function.

Purpose

This generic function is called to determine the discriminating function for a generic function. When a generic function is called, the ***installed*** discriminating function is called with the full set of arguments received by the generic function, and must implement the

behavior of calling the generic function: determining the ordered set of applicable methods, determining the effective method, and running the effective method.
To determine the ordered set of applicable methods, the discriminating function first calls `CLOS:COMPUTE-APPLICABLE-METHODS-USING-CLASSES`. If `CLOS:COMPUTE-APPLICABLE-METHODS-USING-CLASSES` returns a second value of false, the discriminating function then calls `CLOS:COMPUTE-APPLICABLE-METHODS`.
When `CLOS:COMPUTE-APPLICABLE-METHODS-USING-CLASSES` returns a second value of true, the discriminating function is permitted to memoize the primary value as follows. The discriminating function may reuse the list of applicable methods without calling `CLOS:COMPUTE-APPLICABLE-METHODS-USING-CLASSES` again provided that:

i. the generic function is being called again with required arguments which are instances of the same classes,
ii. the generic function has not been reinitialized,
iii. no method has been added to or removed from the generic function,
iv. for all the specializers of all the generic function's methods which are classes, their class precedence lists have not changed, and
v. for any such memoized value, the class precedence list of the class of each of the required arguments has not changed.

Determination of the effective method is done by calling `CLOS:COMPUTE-EFFECTIVE-METHOD`. When the effective method is run, each method's function is called, and receives as arguments:

i. a list of the arguments to the generic function,
ii. whatever other arguments are specified in the `CALL-METHOD` form indicating that the method should be called.

(See `CLOS:MAKE-METHOD-LAMBDA` for more information about how method functions are called.)
The generic function `CLOS:COMPUTE-DISCRIMINATING-FUNCTION` is called, and its result installed, by `ADD-METHOD`, `REMOVE-METHOD`, `INITIALIZE-INSTANCE` and `REINITIALIZE-INSTANCE`.

Methods

(CLOS:COMPUTE-DISCRIMINATING-FUNCTION (*generic-function* STANDARD-GENERIC-FUNCTION))

No behavior is specified for this method beyond that which is specified for the generic function.
This method can be overridden.

Implementation dependent: only in CLISP

Overriding methods can make use of the function `CLOS:COMPUTE-EFFECTIVE-METHOD-AS-FUNCTION`. It is more convenient to call `CLOS:COMPUTE-EFFECTIVE-METHOD-AS-FUNCTION` than `CLOS:COMPUTE-EFFECTIVE-METHOD` because the in the latter case one needs a lot of “glue code” for implementing the local macros `CALL-METHOD` and `MAKE-METHOD`, and this glue code is implementation dependent because it needs

1. to retrieve the declarations list stored in the method-combination object and
2. to handle implementation dependent options that are returned as second value from `CLOS:COMPUTE-EFFECTIVE-METHOD`.

29.6. Methods

29.6.1. Inheritance Structure of method metaobject Classes

Figure 29.5. Inheritance structure of method metaobject classes

Inheritance structure of method metaobject classes

29.6.2. Introspection: Readers for method metaobjects

The reader generic functions which simply return information associated with method metaobjects are presented together here in the format described in Section 29.3.3, "Introspection: Readers for class metaobjects".

Each of these reader generic functions have the same syntax, accepting one required argument called *method*, which must be a method metaobject; otherwise, an `ERROR` is `SIGNAL`ed. An `ERROR` is also `SIGNAL`ed if the method metaobject has not been initialized.

These generic functions can be called by the user or the implementation.

For any of these generic functions which returns a list, such lists will not be mutated by the implementation. The results are undefined if a portable program allows such a list to be mutated.

29.6.2.1. Generic Function `CLOS:METHOD-SPECIALIZERS`

(`CLOS:METHOD-SPECIALIZERS` *method*)

Returns a list of the specializers of *method*. This value is a list of specializer metaobjects. This is the value of the `:SPECIALIZERS` initialization argument that was associated with the method during initialization.

29.6.2.2. Generic Function `METHOD-QUALIFIERS`

(`METHOD-QUALIFIERS` *method*)

Returns a (possibly empty) list of the qualifiers of *method*. This value is a list of non-`NIL` atoms. This is the defaulted value of the `:QUALIFIERS` initialization argument that was associated with the method during initialization.

29.6.2.3. Generic Function CLOS:METHOD-LAMBDA-LIST

(CLOS:METHOD-LAMBDA-LIST *method*)

Returns the (unspecialized) lambda list of *method*. This value is a **Common Lisp** lambda list. This is the value of the `:LAMBDA-LIST` initialization argument that was associated with the method during initialization.

29.6.2.4. Generic Function CLOS:METHOD-GENERIC-FUNCTION

(CLOS:METHOD-GENERIC-FUNCTION *method*)

Returns the generic function that *method* is currently connected to, or `NIL` if it is not currently connected to any generic function. This value is either a generic function metaobject or `NIL`. When a method is first created it is not connected to any generic function. This connection is maintained by the generic functions `ADD-METHOD` and `REMOVE-METHOD`.

29.6.2.5. Generic Function CLOS:METHOD-FUNCTION

(CLOS:METHOD-FUNCTION *method*)

Returns the method function of *method*. This is the value of the `:FUNCTION` initialization argument that was associated with the method during initialization.

29.6.2.6. Methods

The specified methods for the method metaobject readers

(CLOS:METHOD-SPECIALIZERS (*method* STANDARD-METHOD))
(METHOD-QUALIFIERS (*method* STANDARD-METHOD))
(CLOS:METHOD-LAMBDA-LIST (*method* STANDARD-METHOD))
(CLOS:METHOD-FUNCTION (*method* STANDARD-METHOD))

No behavior is specified for these methods beyond that which is specified for their respective generic functions.

(CLOS:METHOD-GENERIC-FUNCTION (*method* STANDARD-METHOD))

No behavior is specified for this method beyond that which is specified for the generic function.
The value returned by this method is maintained by `ADD-METHOD (STANDARD-GENERIC-FUNCTION STANDARD-METHOD)` and `REMOVE-METHOD(STANDARD-GENERIC-FUNCTION STANDARD-METHOD)`.

29.6.3. Initialization of Methods

29.6.3.1. Macro `DEFMETHOD`

The evaluation or execution of a DEFMETHOD form requires first that the body of the method be converted to a method function. This process is described below. The result of this process is a method function and a set of additional initialization arguments to be used when creating the new method. Given these two values, the evaluation or execution of a DEFMETHOD form proceeds in three steps.

The first step ensures the existence of a generic function with the specified name. This is done by calling the function ENSURE-GENERIC-FUNCTION. The first argument in this call is the generic function name specified in the DEFMETHOD form.

The second step is the creation of the new method metaobject by calling MAKE-INSTANCE. The class of the new method metaobject is determined by calling CLOS:GENERIC-FUNCTION-METHOD-CLASS on the result of the call to ENSURE-GENERIC-FUNCTION from the first step.

The initialization arguments received by the call to MAKE-INSTANCE are as follows:

- The value of the :QUALIFIERS initialization argument is a list of the qualifiers which appeared in the DEFMETHOD form. No special processing is done on these values. The order of the elements of this list is the same as in the DEFMETHOD form.
- The value of the :LAMBDA-LIST initialization argument is the unspecialized lambda list from the DEFMETHOD form.
- The value of the :SPECIALIZERS initialization argument is a list of the specializers for the method. For specializers which are classes, the specializer is the class metaobject itself. In the case of EQL specializers, it will be an CLOS:EQL-SPECIALIZER metaobject obtained by calling CLOS:INTERN-EQL-SPECIALIZER on the result of evaluating the EQL specializer form in the lexical environment of the DEFMETHOD form.
- The value of the :FUNCTION initialization argument is the method function.

- The value of the :DECLARATIONS initialization argument is a list of the declaration specifiers from the DEFMETHOD form. If there are no declarations in the macro form, this initialization argument either does not appear, or appears with a value of the empty list.

Implementation dependent: only in CLISP

No `:DECLARATIONS` initialization argument is provided, because method initialization does not support a `:DECLARATIONS` argument, and because the method function is already completely provided through the `:FUNCTION` initialization argument.

- The value of the `:DOCUMENTATION` initialization argument is the documentation string from the `DEFMETHOD` form. If there is no documentation string in the macro form this initialization argument either does not appear, or appears with a value of false.
- Any other initialization argument produced in conjunction with the method function are also included.
- The implementation is free to include additional initialization arguments provided these are not symbols accessible in the **"COMMON-LISP-USER"** package, or exported by any package defined in the [ANSI CL standard].

In the third step, `ADD-METHOD` is called to add the newly created method to the set of methods associated with the generic function metaobject.

The result of the call to `ADD-METHOD` is returned as the result of evaluating or executing the `DEFMETHOD` form.

An example showing a typical `DEFMETHOD` form and a sample expansion is shown in the following example:

An example `DEFMETHOD` form and one possible correct expansion. In the expansion, *`method-lambda`* is the result of calling `CLOS:MAKE-METHOD-LAMBDA` as described in Section 29.6.3.1.1, "Processing Method Bodies". The initargs appearing after `:FUNCTION` are assumed to be additional initargs returned from the call to `CLOS:MAKE-METHOD-LAMBDA`.

```
(defmethod move :before ((p position) (l (eql 0))
                         &OPTIONAL (visiblyp t)
                         &KEY color)
  (set-to-origin p)
  (when visiblyp (show-move p 0 color)))
```

```
(let ((#:g001 (ensure-generic-function 'move)))
  (add-method #:g001
    (make-instance (generic-function-method-class #:g001)
                   :qualifiers '(:before)
                   :specializers (list (find-class 'posit
                                       (intern-eql-specia
                   :lambda-list '(p l &OPTIONAL (visiblyp
                                  &KEY color)
                   :function (function method-lambda)
                   'additional-initarg-1 't
                   'additional-initarg-2 '39)))
```

The processing of the method body for this method is shown below.

29.6.3.1.1. Processing Method Bodies

Before a method can be created, the list of forms comprising the method body must be converted to a method function. This conversion is a two step process.

Note

The body of methods can also appear in the `:METHOD` option of `DEFGENERIC` forms. Initial methods are not considered by any of the protocols specified in this document.

During macro-expansion of the `DEFMETHOD` macro shown in the previous example code similar to this would be run to produce the method lambda and additional initargs. In this example, *`environment`* is the macroexpansion environment of the `DEFMETHOD` macro form.

```
(let ((gf (ensure-generic-function 'move)))
  (make-method-lambda
    gf
    (class-prototype (generic-function-method-class gf))
    '(lambda (p l &OPTIONAL (visiblyp t) &KEY color)
       (set-to-origin p)
       (when visiblyp (show-move p 0 color)))
    environment))
```

The first step occurs during macro-expansion of the macro form. In this step, the method lambda list, declarations and body are converted to a lambda expression called a ***method lambda***. This conversion is based on information associated with the generic function definition in effect at the time the macro form is expanded.

The generic function definition is obtained by calling `ENSURE-GENERIC-FUNCTION` with a first argument of the generic function name specified in the macro form. The `:LAMBDA-LIST` keyword argument is not passed in this call.

Given the generic function, production of the method lambda proceeds by calling `CLOS:MAKE-METHOD-LAMBDA`. The first argument in this call is the generic function obtained as described above. The second argument is the result of calling `CLOS:CLASS-PROTOTYPE` on the result of calling `CLOS:GENERIC-FUNCTION-METHOD-CLASS` on the generic function. The third argument is a lambda expression formed from the method lambda list, declarations and body. The fourth argument is the macro-expansion environment of the macro form; this is the value of the `&ENVIRONMENT` argument to the `DEFMETHOD` macro.

The generic function `CLOS:MAKE-METHOD-LAMBDA` returns two values. The first is the method lambda itself. The second is a list of initialization arguments and values. These are included in the initialization arguments when the method is created.

In the second step, the method lambda is converted to a function which properly captures the lexical scope of the macro form. This is done by having the method lambda appear in the macro-expansion as the argument of the `FUNCTION` special form. During the subsequent evaluation of the macro-expansion, the result of the `FUNCTION` special form is the method function.

Implementation dependent: only in CLISP

See The generic function `CLOS:MAKE-METHOD-LAMBDA` is not implemented.

29.6.3.1.2. Initialization of Generic Function and method metaobjects

An example of creating a generic function and a method metaobject, and then adding the method to the generic function is shown below. This example is comparable to the method definition shown above:

```
(let* ((gf (make-instance 'standard-generic-function
                          :lambda-list '(p l &OPTIONAL vi
       (method-class (generic-function-method-class gf)))
  (multiple-value-bind (lambda initargs)
       (make-method-lambda
         gf
         (class-prototype method-class)
         '(lambda (p l &OPTIONAL (visiblyp t) &KEY color)
            (set-to-origin p)
            (when visiblyp (show-move p 0 color)))
         nil)
    (add-method gf
                (apply #'make-instance method-class
                       :function (compile nil lambda)
                       :specializers (list (find-class 'p
                                           (intern-eql-sp
                       :qualifiers ()
                       :lambda-list '(p l &OPTIONAL (visi
                                      &KEY color)
                       initargs))))
```

29.6.3.1.3. Efficiency

Implementation dependent: only in CLISP and some other implementations

Methods created through `DEFMETHOD` have a faster calling convention than methods created through a portable `MAKE-INSTANCE` invocation.

29.6.3.2. Initialization of method metaobjects

29.6.3.2.1. Methods

A method metaobject can be created by calling `MAKE-INSTANCE`. The initialization arguments establish the definition of the method. A method metaobject cannot be redefined; calling `REINITIALIZE-INSTANCE` `SIGNAL`s an `ERROR`.

Initialization of a method metaobject must be done by calling `MAKE-INSTANCE` and allowing it to call `INITIALIZE-INSTANCE`. Portable programs must **not**

- ... call `INITIALIZE-INSTANCE` directly to initialize a method metaobject;
- ... call `SHARED-INITIALIZE` directly to initialize a method metaobject;
- ... call `CHANGE-CLASS` to change the class of any method metaobject or to turn a non-method object into a method metaobject.

Since metaobject classes may not be redefined, no behavior is specified for the result of calls to `UPDATE-INSTANCE-FOR-REDEFINED-CLASS` on method metaobjects. Since the class of a method metaobject cannot be changed, no behavior is specified for the result of calls to `UPDATE-INSTANCE-FOR-DIFFERENT-CLASS` on method metaobjects.

During initialization, each initialization argument is checked for errors and then associated with the method metaobject. The value can then be accessed by calling the appropriate accessor as shown in Table 29.5, "Initialization arguments and accessors for method metaobjects".

This section begins with a description of the error checking and processing of each initialization argument. This is followed by a table showing the generic functions that can be used to access the stored initialization arguments. The section ends with a set of restrictions on portable methods affecting method metaobject initialization.

In these descriptions, the phrase "this argument defaults to *`value`*" means that when that initialization argument is not supplied, initialization is performed as if *`value`* had been supplied. For some initialization arguments this could be done by the use of default initialization arguments, but whether it is done this way is not specified. Implementations are free to define default initialization arguments for specified method metaobject classes. Portable programs are free to define default initialization arguments for portable subclasses of the class `METHOD`.

- The `:QUALIFIERS` argument is a list of method qualifiers. An `ERROR` is `SIGNAL`ed if this value is not a proper list, or if any element of the list is not a non-null atom. This argument defaults to the empty list.
- The `:LAMBDA-LIST` argument is the unspecialized lambda list of the method. An `ERROR` is `SIGNAL`ed if this value is not a proper lambda list. If this value is not supplied, an `ERROR` is `SIGNAL`ed.
- The `:SPECIALIZERS` argument is a list of the specializer metaobjects for the method. An `ERROR` is `SIGNAL`ed if this value is not a proper list, or if the length of the list differs from the number of required arguments in the `:LAMBDA-LIST` argument, or if any element of the list is not a specializer metaobject. If this value is not supplied, an `ERROR` is `SIGNAL`ed.
- The `:FUNCTION` argument is a method function. It must be compatible with the methods on `CLOS:COMPUTE-EFFECTIVE-METHOD` defined for this class of method and generic function with which it will be used. That is, it must accept the same number of arguments as all uses of `CALL-METHOD` that will call it supply. (See `CLOS:COMPUTE-EFFECTIVE-METHOD` and `CLOS:MAKE-METHOD-LAMBDA` for more information.) An `ERROR` is `SIGNAL`ed if this argument is not supplied.
- When the method being initialized is an instance of a subclass of `CLOS:STANDARD-ACCESSOR-METHOD`, the `:SLOT-DEFINITION` initialization argument must be provided. Its value is the direct slot definition metaobject which defines this accessor method. An `ERROR` is `SIGNAL`ed if the value is not an instance of a subclass of `CLOS:DIRECT-SLOT-DEFINITION`.
- The `:DOCUMENTATION` argument is a string or `NIL`. An `ERROR` is `SIGNAL`ed if this value is not a string or `NIL`. This argument defaults to `NIL`.

After the processing and defaulting of initialization arguments described above, the value of each initialization argument is associated with the method metaobject. These values can then be accessed by calling the corresponding generic function. The correspondences are as follows:

Table 29.5. Initialization arguments and accessors for method metaobjects

Initialization Argument	Generic Function
`:QUALIFIERS`	`METHOD-QUALIFIERS`
`:LAMBDA-LIST`	`CLOS:METHOD-LAMBDA-LIST`
`:SPECIALIZERS`	`CLOS:METHOD-SPECIALIZERS`
`:FUNCTION`	`CLOS:METHOD-FUNCTION`
`:SLOT-DEFINITION`	`CLOS:ACCESSOR-METHOD-SLOT-DEFINITION`
`:DOCUMENTATION`	`DOCUMENTATION`

29.6.3.2.1. Methods

It is not specified which methods provide the initialization behavior described above. Instead, the information needed to allow portable programs to specialize this behavior is presented in as a set of restrictions on the methods a portable program can define. The model is that portable initialization methods have access to the method metaobject when either all or none of the specified initialization has taken effect.

These restrictions govern the methods that a portable program can define on the generic functions `INITIALIZE-INSTANCE`, `REINITIALIZE-INSTANCE`, and `SHARED-INITIALIZE`. These restrictions apply only to methods on these generic functions for which the first specializer is a subclass of the class `METHOD`. Other portable methods on these generic functions are not affected by these restrictions.

- Portable programs must not define methods on `SHARED-INITIALIZE` or `REINITIALIZE-INSTANCE`.
- For `INITIALIZE-INSTANCE`:

- Portable programs must not define primary methods.
- Portable programs may define around-methods, but these must be extending, not overriding methods.
- Portable before-methods must assume that when they are run, none of the initialization behavior described above has been completed.
- Portable after-methods must assume that when they are run, all of the initialization behavior described above has been completed.

The results are undefined if any of these restrictions are violated.

29.6.4. Customization

29.6.4.1. Function `CLOS:EXTRACT-LAMBDA-LIST`

Syntax
`(CLOS:EXTRACT-LAMBDA-LIST` ***`specialized-lambda-list`***`)`

Arguments

`specialized-lambda-list`
a specialized lambda list as accepted by `DEFMETHOD`.

Value
An unspecialized lambda list.

Purpose
This function takes a specialized lambda list and returns the lambda list with the specializers removed. This is a non-destructive operation. Whether the result shares any structure with the argument is unspecified.
If the *`specialized-lambda-list`* argument does not have legal syntax, an `ERROR` is `SIGNAL`ed. This syntax checking does not check the syntax of the actual specializer names, only the syntax of the lambda list and where the specializers appear.

```
(CLOS:EXTRACT-LAMBDA-LIST '((p position)))
⇒ (P)
(CLOS:EXTRACT-LAMBDA-LIST '((p position) x y))
⇒ (P X Y)
(CLOS:EXTRACT-LAMBDA-LIST '(a (b (eql x)) c &REST i))
⇒ (A B C &OPTIONAL I)
```

29.6.4.2. Function `CLOS:EXTRACT-SPECIALIZER-NAMES`

Syntax

(`CLOS:EXTRACT-SPECIALIZER-NAMES` *`specialized-lambda-list`*)

Arguments

`specialized-lambda-list`

a specialized lambda list as accepted by `DEFMETHOD`.

Value

A list of specializer names.

Purpose

This function takes a specialized lambda list and returns its specializer names. This is a non-destructive operation. Whether the result shares structure with the argument is unspecified.

The list returned by this function will not be mutated by the implementation. The results are undefined if a portable program mutates the list returned by this function.

The result of this function will be a list with a number of elements equal to the number of required arguments in *`specialized-lambda-list`*. Specializers are defaulted to the symbol `T`.

If the *`specialized-lambda-list`* argument does not have legal syntax, an `ERROR` is `SIGNAL`ed. This syntax checking does not check the syntax of the actual specializer names, only the syntax of the lambda list and where the specializers appear.

```
(CLOS:EXTRACT-SPECIALIZER-NAMES '((p position)))
⇒ (POSITION)
(CLOS:EXTRACT-SPECIALIZER-NAMES '((p position) x y))
⇒ (POSITION T T)
(CLOS:EXTRACT-SPECIALIZER-NAMES '(a (b (eql x)) c &REST i
⇒ (T (EQL X) T)
```

29.7. Accessor Methods

29.7.1. Introspection

29.7.1.1. Generic Function CLOS:ACCESSOR-METHOD-SLOT-DEFINITION

(CLOS:ACCESSOR-METHOD-SLOT-DEFINITION *method*)

This accessor can only be called on accessor methods. It returns the direct slot definition metaobject that defined this method. This is the value of the `:SLOT-DEFINITION` initialization argument associated with the method during initialization.

The specified methods for the accessor method metaobject readers

(CLOS:ACCESSOR-METHOD-SLOT-DEFINITION (*method* CLOS:STANDARD-ACCESSOR-METHOD))

No behavior is specified for this method beyond that which is specified for the generic function.

29.7.2. Customization

29.7.2.1. Generic Function **CLOS:READER-METHOD-CLASS**

Syntax

(CLOS:READER-METHOD-CLASS *class direct-slot-definition* &REST *initargs*)

Arguments

class
a class metaobject.

direct-slot-definition
a direct slot definition metaobject.

initargs
alternating initialization argument names and values.

Value

A class metaobject.

Purpose

This generic function is called to determine the class of reader methods created during class initialization and reinitialization. The result must be a subclass of CLOS:STANDARD-READER-METHOD. The *initargs* argument must be the same as will be passed to MAKE-INSTANCE to create the reader method. The *initargs* must include :SLOT-DEFINITION with *slot-definition* as its value.

Methods

(CLOS:READER-METHOD-CLASS (*class* STANDARD-CLASS) (*direct-slot-definition* CLOS:STANDARD-DIRECT-SLOT-DEFINITION) &REST *initargs*)
(CLOS:READER-METHOD-CLASS (*class* CLOS:FUNCALLABLE-STANDARD-CLASS) (*direct-slot-definition* CLOS:STANDARD-DIRECT-SLOT-DEFINITION) &REST *initargs*)

These methods return the class CLOS:STANDARD-READER-METHOD.

These methods can be overridden.

29.7.2.2. Generic Function CLOS:WRITER-METHOD-CLASS

Syntax

(CLOS:WRITER-METHOD-CLASS *class direct-slot* &REST *initargs*)

Arguments

class
: a class metaobject.

direct-slot
: a direct slot definition metaobject.

initargs
: a list of initialization arguments and values.

Value

A class metaobject.

Purpose

This generic function is called to determine the class of writer methods created during class initialization and reinitialization. The result must be a subclass of CLOS:STANDARD-WRITER-METHOD. The *initargs* argument must be the same as will be passed to MAKE-INSTANCE to create the reader method. The *initargs* must include :SLOT-DEFINITION with CLOS:SLOT-DEFINITION as its value.

Methods

(CLOS:WRITER-METHOD-CLASS (*class* STANDARD-CLASS) (*direct-slot* CLOS:STANDARD-DIRECT-SLOT-DEFINITION) &REST *initargs*)
(CLOS:WRITER-METHOD-CLASS (*class* CLOS:FUNCALLABLE-STANDARD-CLASS) (*direct-slot* CLOS:STANDARD-DIRECT-SLOT-DEFINITION) &REST *initargs*)

These methods returns the class CLOS:STANDARD-WRITER-METHOD.

These methods can be overridden.

29.8. Specializers

29.8.1. Inheritance Structure of Specializer Metaobject Classes

Figure 29.6. Inheritance structure of specializer metaobject classes

Inheritance structure of specializer metaobject classes

29.8.2. Introspection

29.8.2.1. Function `CLOS:EQL-SPECIALIZER-OBJECT`

Syntax

(`CLOS:EQL-SPECIALIZER-OBJECT` *`eql-specializer`*)

Arguments

`eql-specializer`
: an `EQL` specializer metaobject.

Value
: an object

Purpose
: This function returns the object associated with *`eql-specializer`* during initialization. The value is guaranteed to be `EQL` to the value originally passed to `CLOS:INTERN-EQL-SPECIALIZER`, but it is not necessarily `EQ` to that value.
: This function `SIGNAL`s an `ERROR` if *`eql-specializer`* is not an `EQL` specializer.

29.8.3. Initialization

29.8.3.1. Function `CLOS:INTERN-EQL-SPECIALIZER`

Syntax
: (`CLOS:INTERN-EQL-SPECIALIZER` *`object`*)

Arguments

`object`
: any Lisp object.

Values
: The `EQL` specializer metaobject for *`object`*.

Purpose
: This function returns the unique `EQL` specializer metaobject for *`object`*, creating one if necessary. Two calls to `CLOS:INTERN-EQL-SPECIALIZER` with `EQL` arguments will return the same (i.e., `EQ`) value.

Remarks. The result of calling `CLOS:EQL-SPECIALIZER-OBJECT` on the result of a call to `CLOS:INTERN-EQL-SPECIALIZER` is only guaranteed to be `EQL` to the original *`object`* argument, not necessarily `EQ`.

29.8.4. Updating Dependencies

29.8.4.1. Generic Function `CLOS:SPECIALIZER-DIRECT-METHODS`

Syntax

(`CLOS:SPECIALIZER-DIRECT-METHODS` *`specializer`*)

Arguments

`specializer`
: a specializer metaobject.

Value

A possibly empty list of method metaobjects.

Purpose

This generic function returns the possibly empty set of those methods, connected to generic functions, which have *`specializer`* as a specializer. The elements of this set are method metaobjects. This value is maintained by the generic functions `CLOS:ADD-DIRECT-METHOD` and `CLOS:REMOVE-DIRECT-METHOD`.

Methods

(`CLOS:SPECIALIZER-DIRECT-METHODS` (*`specializer`* `CLASS`))

No behavior is specified for this method beyond that which is specified for the generic function.
This method cannot be overridden unless the following methods are overridden as well:

- CLOS:ADD-DIRECT-METHOD (CLASS METHOD)
- CLOS:REMOVE-DIRECT-METHOD (CLASS METHOD)
- CLOS:SPECIALIZER-DIRECT-GENERIC-FUNCTIONS (CLASS)

(CLOS:SPECIALIZER-DIRECT-METHODS (*specializer* CLOS:EQL-SPECIALIZER))

No behavior is specified for this method beyond that which is specified for the generic function.

29.8.4.2. Generic Function CLOS:SPECIALIZER-DIRECT-GENERIC-FUNCTIONS

Syntax

(CLOS:SPECIALIZER-DIRECT-GENERIC-FUNCTIONS *specializer*)

Arguments

specializer

a specializer metaobject.

Value

A possibly empty list of generic function metaobjects.

Purpose

This generic function returns the possibly empty set of those generic functions which have a method with *specializer* as a specializer. The elements of this set are generic function metaobjects. This value is maintained by the generic functions CLOS:ADD-DIRECT-METHOD and CLOS:REMOVE-DIRECT-METHOD.

Methods

(CLOS:SPECIALIZER-DIRECT-GENERIC-FUNCTIONS (*specializer* CLASS))

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

- CLOS:ADD-DIRECT-METHOD (CLASS METHOD)

- `CLOS:REMOVE-DIRECT-METHOD (CLASS METHOD)`
- `CLOS:SPECIALIZER-DIRECT-METHODS (CLASS)`

`(CLOS:SPECIALIZER-DIRECT-GENERIC-FUNCTIONS (`*`specializer`*` CLOS:EQL-SPECIALIZER))`

No behavior is specified for this method beyond that which is specified for the generic function.

29.8.4.3. Generic Function `CLOS:ADD-DIRECT-METHOD`

Syntax

`(CLOS:ADD-DIRECT-METHOD` *`specializer method`*`)`

Arguments

`specializer`
: a specializer metaobject.

`method`
: a method metaobject.

Values

The values returned by this generic function is unspecified.

Purpose

This generic function is called to maintain a set of backpointers from a specializer to the set of methods specialized to it. If *`method`* is already in the set, it is not added again (no `ERROR` is `SIGNAL`ed).

This set can be accessed as a list by calling the generic function `CLOS:SPECIALIZER-DIRECT-METHODS`. Methods are removed from the set by `CLOS:REMOVE-DIRECT-METHOD`.

The generic function `CLOS:ADD-DIRECT-METHOD` is called by `ADD-METHOD` whenever a method is added to a generic function. It is called once for each of the specializers of the method. Note that in cases where a specializer appears more than once in the specializers of a method, this generic function will be called more than once with the same specializer as argument.

The results are undefined if the *`specializer`* argument is not one of the specializers of the *`method`* argument.

Methods

(CLOS:ADD-DIRECT-METHOD (*specializer* CLASS) (*method* METHOD))

This method implements the behavior of the generic function for class specializers.
No behavior is specified for this method beyond that which is specified for the generic function.
This method cannot be overridden unless the following methods are overridden as well:

- `CLOS:REMOVE-DIRECT-METHOD (CLASS METHOD)`
- `CLOS:SPECIALIZER-DIRECT-GENERIC-FUNCTIONS (CLASS)`
- `CLOS:SPECIALIZER-DIRECT-METHODS (CLASS)`

(CLOS:ADD-DIRECT-METHOD (*specializer* CLOS:EQL-SPECIALIZER) (*method* METHOD))

This method implements the behavior of the generic function for `EQL` specializers.
No behavior is specified for this method beyond that which is specified for the generic function.

29.8.4.4. Generic Function `CLOS:REMOVE-DIRECT-METHOD`

Syntax

`(CLOS:REMOVE-DIRECT-METHOD` *`specializer method`*`)`

Arguments

`specializer`
: a specializer metaobject.

`method`
: a method metaobject.

Values

The values returned by this generic function is unspecified.

Purpose

This generic function is called to maintain a set of backpointers from a specializer to the set of methods specialized to it. If *`method`* is in the set it is removed. If it is not, no `ERROR` is `SIGNAL`ed.

This set can be accessed as a list by calling the generic function `CLOS:SPECIALIZER-DIRECT-METHODS`. Methods are added to the set by `CLOS:ADD-DIRECT-METHOD`.

The generic function `CLOS:REMOVE-DIRECT-METHOD` is called by `REMOVE-METHOD` whenever a method is removed from a generic function. It is called once for each of the specializers of the method. Note that in cases where a specializer appears more than once in the specializers of a method, this generic function will be called more than once with the same specializer as argument.

The results are undefined if the *`specializer`* argument is not one of the specializers of the *`method`* argument.

Methods

(`CLOS:REMOVE-DIRECT-METHOD` (*specializer* `CLASS`) (*method* `METHOD`))

This method implements the behavior of the generic function for class specializers.

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

- `CLOS:ADD-DIRECT-METHOD (CLASS METHOD)`
- `CLOS:SPECIALIZER-DIRECT-GENERIC-FUNCTIONS (CLASS)`
- `CLOS:SPECIALIZER-DIRECT-METHODS (CLASS)`

(`CLOS:REMOVE-DIRECT-METHOD` (*specializer* `CLOS:EQL-SPECIALIZER`) (*method* `METHOD`))

This method implements the behavior of the generic function for `EQL` specializers.

No behavior is specified for this method beyond that which is specified for the generic function.

29.9. Method Combinations

29.9.1. Inheritance Structure of method combination metaobject Classes

Figure 29.7. Inheritance structure of method combination metaobject classes

Inheritance structure of method combination metaobject classes

29.9.2. Customization

29.9.2.1. Generic Function `CLOS:FIND-METHOD-COMBINATION`

Syntax

`(CLOS:FIND-METHOD-COMBINATION` *`generic-function method-combination-type-name method-combination-options`*`)`

Arguments

`generic-function`
: a generic function metaobject.

`method-combination-type-name`
: a symbol which names a type of method combination.

`method-combination-options`
: a list of arguments to the method combination type.

Value

A method combination metaobject.

Purpose

This generic function is called to determine the method combination object used by a generic function.

Remarks. Further details of method combination metaobjects are not specified.

29.10. Slot Access

29.10.1. Instance Structure Protocol

The instance structure protocol is responsible for implementing the behavior of the slot access functions like `SLOT-VALUE` and `(SETF SLOT-VALUE)`.

For each **CLOS** slot access function other than `SLOT-EXISTS-P`, there is a corresponding generic function which actually provides the behavior of the function. When called, the slot access function finds the pertinent effective slot definition metaobject, calls the corresponding generic function and returns its result. The arguments passed on to the generic function include one additional value, the class of the *`object`* argument, which always immediately precedes the *`object`* argument.

Table 29.6. The correspondence between slot access function and underlying slot access generic function

Slot Access Function	Corresponding Slot Access Generic Function
`SLOT-VALUE` *`object slot-name`*	`CLOS:SLOT-VALUE-USING-CLASS` *`class object slot`*

Slot Access Function	Corresponding Slot Access Generic Function
`(SETF SLOT-VALUE)` *new-value object slot-name*	`(SETF CLOS:SLOT-VALUE-USING-CLASS)` *new-value class object slot*
`SLOT-BOUNDP` *object slot-name*	`CLOS:SLOT-BOUNDP-USING-CLASS` *class object slot*
`SLOT-MAKUNBOUND` *object slot-name*	`CLOS:SLOT-MAKUNBOUND-USING-CLASS` *class object slot*

At the lowest level, the instance structure protocol provides only limited mechanisms for portable programs to control the implementation of instances and to directly access the storage associated with instances without going through the indirection of slot access. This is done to allow portable programs to perform certain commonly requested slot access optimizations.

In particular, portable programs can control the implementation of, and obtain direct access to, slots with allocation `:INSTANCE` and type `T`. These are called ***directly accessible slots***.

The relevant specified around-method on `CLOS:COMPUTE-SLOTS` determines the implementation of instances by deciding how each slot in the instance will be stored. For each directly accessible slot, this method allocates a ***location*** and associates it with the effective slot definition metaobject. The location can be accessed by calling the `CLOS:SLOT-DEFINITION-LOCATION` generic function. Locations are non-negative integers. For a given class, the locations increase consecutively, in the order that the directly accessible slots appear in the list of effective slots. (Note that here, the next paragraph, and the specification of this around-method are the only places where the value returned by `CLOS:COMPUTE-SLOTS` is described as a list rather than a set.)

Given the location of a directly accessible slot, the value of that slot in an instance can be accessed with the appropriate accessor. For `STANDARD-CLASS`, this accessor is the function `CLOS:STANDARD-INSTANCE-ACCESS`. For `CLOS:FUNCALLABLE-STANDARD-CLASS`, this accessor is the function `CLOS:FUNCALLABLE-STANDARD-INSTANCE-ACCESS`. In each case, the arguments to the accessor are the instance and the slot location, in that order. See the definition of each accessor for additional restrictions on the use of these function.

Portable programs are permitted to affect and rely on the allocation of locations only in the following limited way: By first defining a portable primary method on `CLOS:COMPUTE-SLOTS` which orders the returned value in a predictable way, and then relying on the defined behavior of the specified around-method to assign locations to all directly accessible slots. Portable programs may compile-in calls to low-level accessors which take advantage of the resulting predictable allocation of slot locations.

This example shows the use of this mechanism to implement a new class metaobject class, `ordered-class` and class option `:SLOT-ORDER`. This option provides control over the allocation of slot locations. In this simple example implementation, the `:SLOT-ORDER` option is not inherited by subclasses; it controls only instances of the class itself.

```
(defclass ordered-class (standard-class)
  ((slot-order :initform ()
               :initarg :slot-order
               :reader class-slot-order)))

(defmethod compute-slots ((class ordered-class))
  (let ((order (class-slot-order class)))
    (sort (copy-list (call-next-method))
          #'(lambda (a b)
              (< (position (slot-definition-name a) order
                 (position (slot-definition-name a) order
```

Following is the source code the user of this extension would write. Note that because the code above does not implement inheritance of the `:SLOT-ORDER` option, the function `distance` must not be called on instances of subclasses of `point`; it can only be called on instances of `point` itself.

```
(defclass point ()
  ((x :initform 0)
   (y :initform 0))
  (:metaclass ordered-class)
  (:slot-order x y))

(defun distance (point)
  (sqrt (/ (+ (expt (standard-instance-access point 0) 2)
```

```
            (expt (standard-instance-access point 1) 2)
        2.0)))
```

Implementation dependent: only in CLISP

You cannot assume that the slot-location values start at 0. In class `point`, for example, *x* and *y* will be at slot locations 1 and 2, not 0 and 1.

In more realistic uses of this mechanism, the calls to the low-level instance structure accessors would not actually appear textually in the source program, but rather would be generated by a meta-level analysis program run during the process of compiling the source program.

29.10.2. Funcallable Instances

Instances of classes which are themselves instances of `CLOS:FUNCALLABLE-STANDARD-CLASS` or one of its subclasses are called ***funcallable instances***. Funcallable instances can only be created by `ALLOCATE-INSTANCE (CLOS:FUNCALLABLE-STANDARD-CLASS)`.

Like standard instances, funcallable instances have slots with the normal behavior. They differ from standard instances in that they can be used as functions as well; that is, they can be passed to `FUNCALL` and `APPLY`, and they can be stored as the definition of a function name. Associated with each funcallable instance is the function which it runs when it is called. This function can be changed with `CLOS:SET-FUNCALLABLE-INSTANCE-FUNCTION`.

The following simple example shows the use of funcallable instances to create a simple, `DEFSTRUCT`-like facility. (Funcallable instances are useful when a program needs to construct and maintain a set of functions and information about those functions. They make it possible to maintain both as the same object rather than two separate objects linked, for example, by hash tables.)

```
(defclass constructor ()
  ((name :initarg :name :accessor constructor-name)
```

```
    (fields :initarg :fields :accessor constructor-fields)
   (:metaclass funcallable-standard-class))
⇒ #>FUNCALLABLE-STANDARD-CLASS CONSTRUCTOR>
(defmethod initialize-instance :after ((c constructor) &KI
   (with-slots (name fields) c
     (set-funcallable-instance-function
       c
       #'(lambda ()
           (let ((new (make-array (1+ (length fields)))))
             (setf (aref new 0) name)
             new)))))
⇒ #<STANDARD-METHOD :AFTER (#<FUNCALLABLE-STANDARD-CLASS
(setq c1 (make-instance 'constructor :name 'position :fie
⇒ #<CONSTRUCTOR #<UNBOUND>>
(setq p1 (funcall c1))
⇒ #(POSITION NIL NIL)
```

29.10.3. Customization

29.10.3.1. Function CLOS:STANDARD-INSTANCE-ACCESS

Syntax

(`CLOS:STANDARD-INSTANCE-ACCESS` *`instance location`*)

Arguments

`instance`
: an object

`location`
: a slot location

Value

an object

Purpose

This function is called to provide direct access to a slot in an instance. By usurping the normal slot lookup protocol, this function is intended to provide highly optimized access to the slots associated with an instance.

The following restrictions apply to the use of this function:

- The *`instance`* argument must be a standard instance (it must have been returned by `ALLOCATE-INSTANCE(STANDARD-CLASS)`).
- The *`instance`* argument cannot be an non-updated obsolete instance.
- The *`location`* argument must be a location of one of the directly accessible slots of the instance's class.
- The slot must be bound.

The results are undefined if any of these restrictions are violated.

Implementation dependent: only in CLISP

The second and third restrictions do not apply in **CLISP**. **CLISP**'s implementation supports non-updated obsolete instances and also supports slots with allocation `:CLASS`.

29.10.3.2. Function `CLOS:FUNCALLABLE-STANDARD-INSTANCE-ACCESS`

Syntax

`(CLOS:FUNCALLABLE-STANDARD-INSTANCE-ACCESS` *`instance location`*`)`

Arguments

`instance`
: an object

`location`
: a slot location

Value

an object

Purpose

This function is called to provide direct access to a slot in an instance. By usurping the normal slot lookup protocol, this function is intended to provide highly optimized access to the slots associated with an instance.

The following restrictions apply to the use of this function:

- The *instance* argument must be a funcallable instance (it must have been returned by `ALLOCATE-INSTANCE (CLOS:FUNCALLABLE-STANDARD-CLASS)`).
- The *instance* argument cannot be an non-updated obsolete instance.
- The *location* argument must be a location of one of the directly accessible slots of the instance's class.
- The slot must be bound.

The results are undefined if any of these restrictions are violated.

Implementation dependent: only in CLISP

The second and third restrictions do not apply in **CLISP**. **CLISP**'s implementation supports non-updated obsolete instances and also supports slots with allocation `:CLASS`.

29.10.3.3. Function `CLOS:SET-FUNCALLABLE-INSTANCE-FUNCTION`

Syntax

`(CLOS:SET-FUNCALLABLE-INSTANCE-FUNCTION` *funcallable-instance function*`)`

Arguments

funcallable-instance

a funcallable instance (it must have been returned by `ALLOCATE-INSTANCE (CLOS:FUNCALLABLE-STANDARD-CLASS)`).

function
A function.

Values
The values returned by this generic function is unspecified.

Purpose
This function is called to set or to change the function of a funcallable instance. After `CLOS:SET-FUNCALLABLE-INSTANCE-FUNCTION` is called, any subsequent calls to *`funcallable-instance`* will run the new function.

29.10.3.4. Generic Function `CLOS:SLOT-VALUE-USING-CLASS`

Syntax
(`CLOS:SLOT-VALUE-USING-CLASS` *`class object slot`*)

Arguments

class
a class metaobject - the class of the *`object`* argument

object
an object

slot
an effective slot definition metaobject

Values
an object

Purpose
This generic function implements the behavior of the `SLOT-VALUE` function. It is called by `SLOT-VALUE` with the class of *`object`* as its first argument and the pertinent effective slot definition metaobject as its third argument.
The generic function `CLOS:SLOT-VALUE-USING-CLASS` returns the value contained in the given slot of the given object. If the slot is unbound, `SLOT-UNBOUND` is called.
The results are undefined if the *`class`* argument is not the class of the *`object`* argument, or if the *`slot`* argument does not appear among the set of effective slots associated with the *`class`* argument.

Methods

(CLOS:SLOT-VALUE-USING-CLASS (*class* STANDARD-CLASS) *object* (*slot* CLOS:STANDARD-EFFECTIVE-SLOT-DEFINITION))
(CLOS:SLOT-VALUE-USING-CLASS (*class* CLOS:FUNCALLABLE-STANDARD-CLASS) *object* (*slot* CLOS:STANDARD-EFFECTIVE-SLOT-DEFINITION))

These methods implement the full behavior of this generic function for slots with allocation `:INSTANCE` and `:CLASS`. If the supplied slot has an allocation other than `:INSTANCE` or `:CLASS` an `ERROR` is `SIGNAL`ed.

Overriding these methods is permitted, but may require overriding other methods in the standard implementation of the slot access protocol.

(CLOS:SLOT-VALUE-USING-CLASS (*class* BUILT-IN-CLASS) *object slot*)

This method `SIGNAL`s an `ERROR`.

29.10.3.5. Generic Function (SETF CLOS:SLOT-VALUE-USING-CLASS)

Syntax

`((SETF CLOS:SLOT-VALUE-USING-CLASS) new-value class object slot)`

Arguments

new-value
an object

class
a class metaobject - the class of the *object* argument.

object
an object

slot
an effective slot definition metaobject.

Value

The *new-value* argument.

Purpose

The generic function `(SETF CLOS:SLOT-VALUE-USING-CLASS)` implements the behavior of the `(SETF SLOT-VALUE)` function. It is called by `(SETF SLOT-VALUE)` with the class of *`object`* as its second argument and the pertinent effective slot definition metaobject as its fourth argument.
The generic function `(SETF CLOS:SLOT-VALUE-USING-CLASS)` sets the value contained in the given slot of the given object to the given new value; any previous value is lost.
The results are undefined if the *`class`* argument is not the class of the *`object`* argument, or if the *`slot`* argument does not appear among the set of effective slots associated with the *`class`* argument.

Methods

`((SETF CLOS:SLOT-VALUE-USING-CLASS)` *`new-value`* `(`*`class`* `STANDARD-CLASS)` *`object`* `(`*`slot`* `CLOS:STANDARD-EFFECTIVE-SLOT-DEFINITION))`
`((SETF CLOS:SLOT-VALUE-USING-CLASS)` *`new-value`* `(`*`class`* `CLOS:FUNCALLABLE-STANDARD-CLASS)` *`object`* `(`*`slot`* `CLOS:STANDARD-EFFECTIVE-SLOT-DEFINITION))`

These methods implement the full behavior of this generic function for slots with allocation `:INSTANCE` and `:CLASS`. If the supplied slot has an allocation other than `:INSTANCE` or `:CLASS` an `ERROR` is `SIGNAL`ed.
Overriding these methods is permitted, but may require overriding other methods in the standard implementation of the slot access protocol.

`((SETF CLOS:SLOT-VALUE-USING-CLASS)` *`new-value`* `(`*`class`* `BUILT-IN-CLASS)` *`object slot`*`)`

This method `SIGNAL`s an `ERROR`.

29.10.3.6. Generic Function `CLOS:SLOT-BOUNDP-USING-CLASS`

Syntax

`(CLOS:SLOT-BOUNDP-USING-CLASS` *`class object slot`*`)`

Arguments

`class`

a class metaobject - the class of the `object` argument.

object
: an object

slot
: an effective slot definition metaobject.

Value

`BOOLEAN`

Purpose

This generic function implements the behavior of the `SLOT-BOUNDP` function. It is called by `SLOT-BOUNDP` with the class of `object` as its first argument and the pertinent effective slot definition metaobject as its third argument.

The generic function `CLOS:SLOT-BOUNDP-USING-CLASS` tests whether a specific slot in an instance is bound.

The results are undefined if the `class` argument is not the class of the `object` argument, or if the `slot` argument does not appear among the set of effective slots associated with the `class` argument.

Methods

`(CLOS:SLOT-BOUNDP-USING-CLASS (class STANDARD-CLASS) object (slot CLOS:STANDARD-EFFECTIVE-SLOT-DEFINITION))`

`(CLOS:SLOT-BOUNDP-USING-CLASS (class CLOS:FUNCALLABLE-STANDARD-CLASS) object (slot CLOS:STANDARD-EFFECTIVE-SLOT-DEFINITION))`

These methods implement the full behavior of this generic function for slots with allocation `:INSTANCE` and `:CLASS`. If the supplied slot has an allocation other than `:INSTANCE` or `:CLASS` an `ERROR` is `SIGNAL`ed.

Overriding these methods is permitted, but may require overriding other methods in the standard implementation of the slot access protocol.

`(CLOS:SLOT-BOUNDP-USING-CLASS (class BUILT-IN-CLASS) object slot)`

This method `SIGNAL`s an `ERROR`.

Remarks. In cases where the class metaobject class does not distinguish unbound slots, true should be returned.

29.10.3.7. Generic Function CLOS:SLOT-MAKUNBOUND-USING-CLASS

Syntax

`(CLOS:SLOT-MAKUNBOUND-USING-CLASS` ***`class object slot`***`)`

Arguments

`class`
: a class metaobject - the class of the *`object`* argument.

`object`
: an object

`slot`
: an effective slot definition metaobject.

Value

The *`object`* argument.

Purpose

This generic function implements the behavior of the `SLOT-MAKUNBOUND` function. It is called by `SLOT-MAKUNBOUND` with the class of *`object`* as its first argument and the pertinent effective slot definition metaobject as its third argument.

The generic function `CLOS:SLOT-MAKUNBOUND-USING-CLASS` restores a slot in an object to its unbound state. The interpretation of “restoring a slot to its unbound state” depends on the class metaobject class.

The results are undefined if the *`class`* argument is not the class of the *`object`* argument, or if the *`slot`* argument does not appear among the set of effective slots associated with the *`class`* argument.

Methods

`(CLOS:SLOT-MAKUNBOUND-USING-CLASS (`*`class`*` STANDARD-CLASS)` ***`object`*** **`(`*`slot`*` CLOS:STANDARD-EFFECTIVE-SLOT-DEFINITION))`**

`(CLOS:SLOT-MAKUNBOUND-USING-CLASS (`*`class`*` CLOS:FUNCALLABLE-STANDARD-CLASS)` ***`object`*** **`(`*`slot`*` CLOS:STANDARD-EFFECTIVE-SLOT-DEFINITION))`**

These methods implement the full behavior of this generic function for slots with allocation `:INSTANCE` and `:CLASS`. If the supplied slot has an allocation other than `:INSTANCE` or `:CLASS` an `ERROR` is `SIGNAL`ed.

Overriding these methods is permitted, but may require overriding other methods in the standard implementation of the slot access protocol.

(CLOS:SLOT-MAKUNBOUND-USING-CLASS (*class* BUILT-IN-CLASS) *object slot*)

This method `SIGNAL`s an `ERROR`.

29.11. Dependent Maintenance

It is convenient for portable metaobjects to be able to memoize information about other metaobjects, portable or otherwise. Because class and generic function metaobjects can be reinitialized, and generic function metaobjects can be modified by adding and removing methods, a means must be provided to update this memoized information.

The dependent maintenance protocol supports this by providing a way to register an object which should be notified whenever a class or generic function is modified. An object which has been registered this way is called a ***dependent*** of the class or generic function metaobject. The dependents of class and generic function metaobjects are maintained with `CLOS:ADD-DEPENDENT` and `CLOS:REMOVE-DEPENDENT`. The dependents of a class or generic function metaobject can be accessed with `CLOS:MAP-DEPENDENTS`. Dependents are notified about a modification by calling `CLOS:UPDATE-DEPENDENT`. (See the specification of `CLOS:UPDATE-DEPENDENT` for detailed description of the circumstances under which it is called.)

To prevent conflicts between two portable programs, or between portable programs and the implementation, portable code must not register metaobjects themselves as dependents. Instead, portable programs which need to record a metaobject as a dependent, should encapsulate that metaobject in some other kind of object, and record that object as the dependent. The results are undefined if this restriction is violated.

This example shows a general facility for encapsulating metaobjects before recording them as dependents. The facility defines a basic kind of encapsulating object: an updater. Specializations of the basic class can be defined with appropriate special updating behavior. In this way, information about the updating required is associated with each updater rather than with the metaobject being updated.

Updaters are used to encapsulate any metaobject which requires updating when a given class or generic function is modified. The function `record-updater` is called to both create an updater and add it to the dependents of the class or generic function. Methods on the generic function `CLOS:UPDATE-DEPENDENT`, specialized to the specific class of updater do the appropriate update work.

```
(defclass updater ()
  ((dependent :initarg :dependent :reader dependent)))

(defun record-updater (class dependee dependent &REST ini
  (let ((updater (apply #'make-instance class :dependent d
                        initargs)))
    (add-dependent dependee updater)
    updater))
```

A `flush-cache-updater` simply flushes the cache of the dependent when it is updated.

```
(defclass flush-cache-updater (updater) ())

(defmethod update-dependent (dependee (updater flush-cache
                              &REST args)
  (declare (ignore args))
  (flush-cache (dependent updater)))
```

29.11.1. Protocol

29.11.1.1. Generic Function CLOS:UPDATE-DEPENDENT

Syntax

(CLOS:UPDATE-DEPENDENT *metaobject* *dependent* &REST *initargs*)

Arguments

metaobject
: a class or generic function metaobject - the metaobject being reinitialized or otherwise modified.

dependent
: an object - the dependent being updated.

initargs
: a list of the initialization arguments for the metaobject redefinition.

Values

The values returned by this generic function is unspecified.

Purpose

This generic function is called to update a dependent of *metaobject*.

When a class or a generic function is reinitialized each of its dependents is updated. The *initargs* argument to CLOS:UPDATE-DEPENDENT is the set of initialization arguments received by REINITIALIZE-INSTANCE.

When a method is added to a generic function, each of the generic function's dependents is updated. The *initargs* argument is a list of two elements: the symbol ADD-METHOD, and the method that was added.

When a method is removed from a generic function, each of the generic function's dependents is updated. The *initargs* argument is a list of two elements: the symbol REMOVE-METHOD, and the method that was removed.

In each case, CLOS:MAP-DEPENDENTS is used to call CLOS:UPDATE-DEPENDENT on each of the dependents. So, for example, the update of a generic function's dependents when a method is added could be performed by the following code:

```
(CLOS:MAP-DEPENDENTS generic-function
                     #'(lambda (dep)
                         (CLOS:UPDATE-DEPENDENT generic-f
                                                dep
                                                'add-method
                                                new-method)))
```

Remarks. See Section 29.11, “Dependent Maintenance” for remarks about the use of this facility.

29.11.1.2. Generic Function `CLOS:ADD-DEPENDENT`

Syntax

`(CLOS:ADD-DEPENDENT` *`metaobject dependent`*`)`

Arguments

`metaobject`
: a class or generic function metaobject

`dependent`
: an object

Values

The values returned by this generic function is unspecified.

Purpose

This generic function adds *`dependent`* to the dependents of *`metaobject`*. If *`dependent`* is already in the set of dependents it is not added again (no `ERROR` is `SIGNAL`ed).

The generic function `CLOS:MAP-DEPENDENTS` can be called to access the set of dependents of a class or generic function. The generic function `CLOS:REMOVE-DEPENDENT` can be called to remove an object from the set of dependents of a class or generic function.

The effect of calling `CLOS:ADD-DEPENDENT` or `CLOS:REMOVE-DEPENDENT` while a call to `CLOS:MAP-DEPENDENTS` on the same class or generic function is in progress is unspecified.

The situations in which `CLOS:ADD-DEPENDENT` is called are not specified.

Methods

(CLOS:ADD-DEPENDENT (*class* STANDARD-CLASS) *dependent*)

No behavior is specified for this method beyond that which is specified for the generic function.
This method cannot be overridden unless the following methods are overridden as well:

- `CLOS:REMOVE-DEPENDENT (STANDARD-CLASS T)`
- `CLOS:MAP-DEPENDENTS (STANDARD-CLASS T)`

(CLOS:ADD-DEPENDENT (*class* CLOS:FUNCALLABLE-STANDARD-CLASS) *dependent*)

No behavior is specified for this method beyond that which is specified for the generic function.
This method cannot be overridden unless the following methods are overridden as well:

- `CLOS:REMOVE-DEPENDENT (CLOS:FUNCALLABLE-STANDARD-CLASS T)`
- `CLOS:MAP-DEPENDENTS (CLOS:FUNCALLABLE-STANDARD-CLASS T)`

(CLOS:ADD-DEPENDENT (*generic-function* STANDARD-GENERIC-FUNCTION) *dependent*)

No behavior is specified for this method beyond that which is specified for the generic function.
This method cannot be overridden unless the following methods are overridden as well:

- `CLOS:REMOVE-DEPENDENT (STANDARD-GENERIC-FUNCTION T)`
- `CLOS:MAP-DEPENDENTS (STANDARD-GENERIC-FUNCTION T)`

Remarks. See Section 29.11, “Dependent Maintenance” for remarks about the use of this facility.

29.11.1.3. Generic Function CLOS:REMOVE-DEPENDENT

Syntax

(CLOS:REMOVE-DEPENDENT *metaobject dependent*)

Arguments

metaobject
: a class or generic function metaobject

dependent
: an object

Values

The values returned by this generic function is unspecified.

Purpose

This generic function removes *dependent* from the dependents of *metaobject*. If *dependent* is not one of the dependents of *metaobject*, no ERROR is SIGNALed.

The generic function CLOS:MAP-DEPENDENTS can be called to access the set of dependents of a class or generic function. The generic function CLOS:ADD-DEPENDENT can be called to add an object from the set of dependents of a class or generic function. The effect of calling CLOS:ADD-DEPENDENT or CLOS:REMOVE-DEPENDENT while a call to CLOS:MAP-DEPENDENTS on the same class or generic function is in progress is unspecified.

The situations in which CLOS:REMOVE-DEPENDENT is called are not specified.

Methods

(CLOS:REMOVE-DEPENDENT (*class* STANDARD-CLASS) *dependent*)

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

- CLOS:ADD-DEPENDENT (STANDARD-CLASS T)
- CLOS:MAP-DEPENDENTS (STANDARD-CLASS T)

(CLOS:REMOVE-DEPENDENT (*class* CLOS:FUNCALLABLE-STANDARD-CLASS) *dependent*)

No behavior is specified for this method beyond that which is specified for the generic function.

This method cannot be overridden unless the following methods are overridden as well:

- `CLOS:ADD-DEPENDENT (CLOS:FUNCALLABLE-STANDARD-CLASS T)`
- `CLOS:MAP-DEPENDENTS (CLOS:FUNCALLABLE-STANDARD-CLASS T)`

`(CLOS:REMOVE-DEPENDENT (`*`class`* `STANDARD-GENERIC-FUNCTION)` *`dependent`*`)`

No behavior is specified for this method beyond that which is specified for the generic function.
This method cannot be overridden unless the following methods are overridden as well:

- `CLOS:ADD-DEPENDENT (STANDARD-GENERIC-FUNCTION T)`
- `CLOS:MAP-DEPENDENTS (STANDARD-GENERIC-FUNCTION T)`

Remarks. See Section 29.11, “Dependent Maintenance” for remarks about the use of this facility.

29.11.1.4. Generic Function `CLOS:MAP-DEPENDENTS`

Syntax

`(CLOS:MAP-DEPENDENTS` *`metaobject function`*`)`

Arguments

`metaobject`
: a class or generic function metaobject.

`function`
: a function which accepts one argument.

Values

The values returned by this generic function is unspecified.

Purpose

This generic function applies *`function`* to each of the dependents of *`metaobject`*. The order in which the dependents are processed is not specified, but *`function`* is applied to each dependent once and only once. If, during the mapping, `CLOS:ADD-DEPENDENT` or `CLOS:REMOVE-DEPENDENT` is called to alter the dependents of *metaobject*, it is not specified whether the newly added or removed dependent will have *function* applied to it.

Methods

(CLOS:MAP-DEPENDENTS (*metaobject* STANDARD-CLASS) *function*)

No behavior is specified for this method beyond that which is specified for the generic function.
This method cannot be overridden unless the following methods are overridden as well:

- `CLOS:ADD-DEPENDENT (STANDARD-CLASS T)`
- `CLOS:REMOVE-DEPENDENT (STANDARD-CLASS T)`

(CLOS:MAP-DEPENDENTS (*metaobject* CLOS:FUNCALLABLE-STANDARD-CLASS) *function*)

No behavior is specified for this method beyond that which is specified for the generic function.
This method cannot be overridden unless the following methods are overridden as well:

- `CLOS:ADD-DEPENDENT (CLOS:FUNCALLABLE-STANDARD-CLASS T)`
- `CLOS:REMOVE-DEPENDENT (CLOS:FUNCALLABLE-STANDARD-CLASS T)`

(CLOS:MAP-DEPENDENTS (*metaobject* STANDARD-GENERIC-FUNCTION) *function*)

No behavior is specified for this method beyond that which is specified for the generic function.
This method cannot be overridden unless the following methods are overridden as well:

- `CLOS:ADD-DEPENDENT (STANDARD-GENERIC-FUNCTION T)`
- `CLOS:REMOVE-DEPENDENT (STANDARD-GENERIC-FUNCTION T)`

Remarks. See Section 29.11, "Dependent Maintenance" for remarks about the use of this facility.

29.12. Deviations from [AMOP]

This section lists the differences between the [AMOP] and the **CLISP** implementation thereof.

Not implemented in CLISP

- The generic function `CLOS:MAKE-METHOD-LAMBDA` is not implemented. See Section 29.5.3.2, "Generic Function Invocation Protocol".

Features implemented differently in CLISP

- The class precedence list of `CLOS:FUNCALLABLE-STANDARD-OBJECT` is different. See Section 29.2.2, "Inheritance Structure of Metaobject Classes".
- The `DEFCLASS` macro passes default values to `CLOS:ENSURE-CLASS`. See Section 29.3.1, "Macro `DEFCLASS`".
- The `DEFGENERIC` macro passes default values to `ENSURE-GENERIC-FUNCTION`. See Section 29.5.3.1, "Macro `DEFGENERIC`".
- The class `CLOS:FORWARD-REFERENCED-CLASS` is implemented differently. See Implementation of class `CLOS:FORWARD-REFERENCED-CLASS` in **CLISP**.
- The function `CLOS:GENERIC-FUNCTION-ARGUMENT-PRECEDENCE-ORDER` `SIGNAL`s an `ERROR` if the generic function has no lambda list.

Extensions specific to CLISP

- The Meta-Object Protocol is applicable to classes of type `STRUCTURE-CLASS`. The default superclass for `STRUCTURE-CLASS` instances is `STRUCTURE-OBJECT`. Structure classes do not support multiple inheritance and reinitialization. See Section 29.3.5.1, "Initialization of class metaobjects". See also Section 8.2, "The structure Meta-Object Protocol.".
- The `DEFGENERIC` macro supports user-defined options. See User-defined options.

- The class `METHOD` is subclassable. See Section 29.2.2, “Inheritance Structure of Metaobject Classes”.
- Slot names like `NIL` and `T` are allowed. See Section 29.4.2.1.1, “Generic Function `CLOS:SLOT-DEFINITION-NAME`”.
- The `CLOS:VALIDATE-SUPERCLASS` method is more permissive by default and does not need to be overridden in some “obvious” cases. See Section 29.3.6.7, “Generic Function `CLOS:VALIDATE-SUPERCLASS`”.
- New generic function `CLOS:COMPUTE-DIRECT-SLOT-DEFINITION-INITARGS`. It can sometimes be used when overriding `CLOS:DIRECT-SLOT-DEFINITION-CLASS` is cumbersome.
- New generic function `CLOS:COMPUTE-EFFECTIVE-SLOT-DEFINITION-INITARGS`. It can sometimes be used when overriding `CLOS:EFFECTIVE-SLOT-DEFINITION-CLASS` is cumbersome.
- New function `CLOS:COMPUTE-EFFECTIVE-METHOD-AS-FUNCTION`. It can be used in overriding methods of `CLOS:COMPUTE-DISCRIMINATING-FUNCTION`.
- The generic function `CLOS:ENSURE-GENERIC-FUNCTION-USING-CLASS` accepts a `:DECLARE` keyword.
- The functions `CLOS:FUNCALLABLE-STANDARD-INSTANCE-ACCESS` and `CLOS:STANDARD-INSTANCE-ACCESS` support non-updated obsolete instances and also support slots with allocation `:CLASS`.
- The existence of the function `CLOS:CLASS-DIRECT-SUBCLASSES` does not prevent otherwise unreferenced classes from being garbage-collected.

Chapter 30. Gray streams

Table of Contents

30.1. Overview

This interface permits the definition of new classes of streams, and programming their behavior by defining methods for the elementary stream operations. It is based on the proposal STREAM-DEFINITION-BY-USER:GENERIC-FUNCTIONS of David N. Gray to X3J13 and is supported by most **Common Lisp** implementations currently in use.

All symbols defined by this interface, starting with the prefix `FUNDAMENTAL-` or `STREAM-`, are exported from the package **“GRAY”** and `EXT:RE-EXPORT`ed from **“EXT”**.

Defined classes

`GRAY:FUNDAMENTAL-STREAM`
: This is a superclass of all user-defined streams. It is a subclass of `STREAM` and of `STANDARD-OBJECT`. Its metaclass is `STANDARD-CLASS`.

`GRAY:FUNDAMENTAL-INPUT-STREAM`
: This is a superclass of all user-defined input `STREAM`s. It is a subclass of `GRAY:FUNDAMENTAL-STREAM`. The built-in function `INPUT-STREAM-P` returns true on instances of this class. This means that when you define a new stream class capable of doing input, you have to make it a subclass of `GRAY:FUNDAMENTAL-INPUT-STREAM`.

`GRAY:FUNDAMENTAL-OUTPUT-STREAM`
: This is a superclass of all user-defined output `STREAM`s. It is a subclass of `GRAY:FUNDAMENTAL-STREAM`. The built-in function `OUTPUT-STREAM-P` returns true on instances of this class. This means that when you define a new stream class capable of doing output, you have to make it a subclass of `GRAY:FUNDAMENTAL-OUTPUT-STREAM`.

`GRAY:FUNDAMENTAL-CHARACTER-STREAM`
: This is a superclass of all user-defined streams whose `STREAM-ELEMENT-TYPE` is `CHARACTER`. It is a subclass of `GRAY:FUNDAMENTAL-STREAM`. It defines a method on `STREAM-ELEMENT-TYPE` that returns `CHARACTER`.

`GRAY:FUNDAMENTAL-BINARY-STREAM`
: This is a superclass of all user-defined streams whose `STREAM-ELEMENT-TYPE` is a subtype of `INTEGER`. It is a subclass of `GRAY:FUNDAMENTAL-STREAM`. When you define a subclass of

`GRAY:FUNDAMENTAL-BINARY-STREAM`, you have to provide a method on `STREAM-ELEMENT-TYPE`.

`GRAY:FUNDAMENTAL-CHARACTER-INPUT-STREAM`

This is a convenience class inheriting from both `GRAY:FUNDAMENTAL-CHARACTER-STREAM` and `GRAY:FUNDAMENTAL-INPUT-STREAM`.

`GRAY:FUNDAMENTAL-CHARACTER-OUTPUT-STREAM`

This is a convenience class inheriting from both `GRAY:FUNDAMENTAL-CHARACTER-STREAM` and `GRAY:FUNDAMENTAL-OUTPUT-STREAM`.

`GRAY:FUNDAMENTAL-BINARY-INPUT-STREAM`

This is a convenience class inheriting from both `GRAY:FUNDAMENTAL-BINARY-STREAM` and `GRAY:FUNDAMENTAL-INPUT-STREAM`.

`GRAY:FUNDAMENTAL-BINARY-OUTPUT-STREAM`

This is a convenience class inheriting from both `GRAY:FUNDAMENTAL-BINARY-STREAM` and `GRAY:FUNDAMENTAL-OUTPUT-STREAM`.

General generic functions defined on streams

`(STREAM-ELEMENT-TYPE` *`stream`*`)`

Returns the stream's element type, normally a subtype of `CHARACTER` or `INTEGER`.
The method for `GRAY:FUNDAMENTAL-CHARACTER-STREAM` returns `CHARACTER`.

`((SETF STREAM-ELEMENT-TYPE)` *`new-element-type stream`*`)`

Changes the stream's element type.
The default method `SIGNAL`s an `ERROR`.
This function is a **CLISP** extension (see Section 21.3.1, “Function `STREAM-ELEMENT-TYPE`”).

`(CLOSE` *`stream`* `&KEY :ABORT)`

Closes the stream and flushes any associated buffers.
When you define a primary method on this function, do not forget to `CALL-NEXT-METHOD`.

`(OPEN-STREAM-P` *`stream`*`)`

Returns true before the stream has been closed, and `NIL` after the stream has been closed.
You do not need to add methods to this function.

`(GRAY:STREAM-POSITION` *`stream position`*`)`

Just like `FILE-POSITION`, but `NIL` *`position`* means inquire.

You must define a method for this function.

generic functions for character input

(GRAY:STREAM-READ-CHAR *stream*)
: If a character was pushed back using `GRAY:STREAM-UNREAD-CHAR`, returns and consumes it. Otherwise returns and consumes the next character from the stream. Returns `:EOF` if the `end-of-stream` is reached.
You must define a method for this function.

(GRAY:STREAM-UNREAD-CHAR *stream char*)
: Pushes *`char`*, which must be the last character read from the *`stream`*, back onto the front of the *`stream`*.
You must define a method for this function.

(GRAY:STREAM-READ-CHAR-NO-HANG *stream*)
: Returns a character or `:EOF`, like `GRAY:STREAM-READ-CHAR`, if that would return immediately. If `GRAY:STREAM-READ-CHAR`'s value is not available immediately, returns `NIL` instead of waiting.
The default method simply calls `GRAY:STREAM-READ-CHAR`; this is sufficient for streams whose `GRAY:STREAM-READ-CHAR` method never blocks.

(GRAY:STREAM-PEEK-CHAR *stream*)
: If a character was pushed back using `GRAY:STREAM-UNREAD-CHAR`, returns it. Otherwise returns the next character from the stream, avoiding any side effects `GRAY:STREAM-READ-CHAR` would do. Returns `:EOF` if the `end-of-stream` is reached.
The default method calls `GRAY:STREAM-READ-CHAR` and `GRAY:STREAM-UNREAD-CHAR`; this is sufficient for streams whose `GRAY:STREAM-READ-CHAR` method has no side-effects.

(GRAY:STREAM-LISTEN *stream*)
: If a character was pushed back using `GRAY:STREAM-UNREAD-CHAR`, returns it. Otherwise returns the next character from the stream, if already available. If no character is available immediately, or if `end-of-stream` is reached, returns `NIL`.
The default method calls `GRAY:STREAM-READ-CHAR-NO-HANG` and `GRAY:STREAM-UNREAD-CHAR`; this is sufficient for streams whose `GRAY:STREAM-READ-CHAR` method has no side-effects.

(GRAY:STREAM-READ-CHAR-WILL-HANG-P *stream*)
: Returns `NIL` if `GRAY:STREAM-READ-CHAR` will return immediately. Otherwise it returns true.

The default method calls GRAY:STREAM-READ-CHAR-NO-HANG and GRAY:STREAM-UNREAD-CHAR; this is sufficient for streams whose GRAY:STREAM-READ-CHAR method has no side-effects.
This function is a **CLISP** extension (see EXT:READ-CHAR-WILL-HANG-P).

(GRAY:STREAM-READ-CHAR-SEQUENCE *stream sequence* &OPTIONAL [*start* [*end*]])

Fills the subsequence of *sequence* specified by :START and :END with characters consecutively read from *stream*. Returns the index of the first element of *sequence* that was not updated (= *end*, or < *end* if the stream reached its end).
sequence is an ARRAY of CHARACTERs, i.e. a STRING. *start* is a nonnegative INTEGER and defaults to 0. *end* is a nonnegative INTEGER or NIL and defaults to NIL, which stands for (LENGTH *sequence*).
The default method repeatedly calls GRAY:STREAM-READ-CHAR; this is always sufficient if speed does not matter.
This function is a **CLISP** extension (see EXT:READ-CHAR-SEQUENCE).

(GRAY:STREAM-READ-LINE *stream*)

Reads a line of characters, and return two values: the line (a STRING, without the terminating #\Newline character), and a BOOLEAN value which is true if the line was terminated by end-of-stream instead of #\Newline.
The default method repeatedly calls GRAY:STREAM-READ-CHAR; this is always sufficient.

(GRAY:STREAM-CLEAR-INPUT *stream*)

Clears all pending interactive input from the *stream*, and returns true if some pending input was removed.
The default method does nothing and returns NIL; this is sufficient for non-interactive streams.

generic functions for character output

(GRAY:STREAM-WRITE-CHAR *stream char*)

Writes *char*.
You must define a method for this function.

(GRAY:STREAM-LINE-COLUMN *stream*)

Returns the column number where the next character would be written (0 stands for the first column), or NIL if that is not meaningful for this stream.

You must define a method for this function.

(GRAY:STREAM-START-LINE-P *stream*)

Returns true if the next character would be written at the start of a new line.

The default method calls GRAY:STREAM-LINE-COLUMN and compares its result with 0; this is sufficient for streams whose GRAY:STREAM-LINE-COLUMN never returns NIL.

(GRAY:STREAM-WRITE-CHAR-SEQUENCE *stream sequence* &OPTIONAL [*start* [*end*]])

Outputs the subsequence of *sequence* specified by :START and :END to *stream*.

sequence is an ARRAY of CHARACTERs, i.e. a STRING. *start* is a nonnegative INTEGER and defaults to 0. *end* is a nonnegative integer or NIL and defaults to NIL, which stands for (LENGTH *sequence*).

The default method repeatedly calls GRAY:STREAM-WRITE-CHAR; this is always sufficient if speed does not matter.

This function is a **CLISP** extension (see EXT:WRITE-CHAR-SEQUENCE).

(GRAY:STREAM-WRITE-STRING *stream string* &OPTIONAL [*start* [*end*]])

Outputs the subsequence of *string* specified by :START and :END to *stream*. Returns *string*.

string is a string. *start* is a nonnegative integer and default to 0. *end* is a nonnegative integer or NIL and defaults to NIL, which stands for (LENGTH *string*).

The default method calls GRAY:STREAM-WRITE-CHAR-SEQUENCE; this is always sufficient.

(GRAY:STREAM-TERPRI *stream*)

Outputs a #\Newline character.

The default method calls GRAY:STREAM-WRITE-CHAR; this is always sufficient.

(GRAY:STREAM-FRESH-LINE *stream*)

Possibly outputs a #\Newline character, so as to ensure that the next character would be written at the start of a new line. Returns true if it did output a #\Newline character.

The default method calls GRAY:STREAM-START-LINE-P and then GRAY:STREAM-TERPRI if necessary; this is always sufficient.

(GRAY:STREAM-FINISH-OUTPUT *stream*)

Ensures that any buffered output has reached its destination, and then returns.

The default method does nothing.

(GRAY:STREAM-FORCE-OUTPUT *stream*)
: Brings any buffered output on its way towards its destination, and returns without waiting until it has reached its destination.
The default method does nothing.

(GRAY:STREAM-CLEAR-OUTPUT *stream*)
: Attempts to discard any buffered output which has not yet reached its destination.
The default method does nothing.

(GRAY:STREAM-ADVANCE-TO-COLUMN *stream column*)
: Ensures that the next character will be written at least at *column*.
The default method outputs an appropriate amount of space characters; this is sufficient for non-proportional output.

generic functions for binary input

(GRAY:STREAM-READ-BYTE *stream*)
: Returns and consumes the next integer from the stream.
Returns `:EOF` if the `end-of-stream` is reached.
You must define a method for this function.

(GRAY:STREAM-READ-BYTE-LOOKAHEAD *stream*)
: To be called only if *stream*'s `STREAM-ELEMENT-TYPE` is `(UNSIGNED-BYTE 8)` or `(SIGNED-BYTE 8)`. Returns `T` if `GRAY:STREAM-READ-BYTE` would return immediately with an `INTEGER` result. Returns `:EOF` if the `end-of-stream` is already known to be reached. If `GRAY:STREAM-READ-BYTE`'s value is not available immediately, returns `NIL` instead of waiting.
You must define a method for this function.
This function is a **CLISP** extension (see `EXT:READ-BYTE-LOOKAHEAD`).

(GRAY:STREAM-READ-BYTE-WILL-HANG-P *stream*)
: To be called only if *stream*'s `STREAM-ELEMENT-TYPE` is `(UNSIGNED-BYTE 8)` or `(SIGNED-BYTE 8)`. Returns `NIL` if `GRAY:STREAM-READ-BYTE` will return immediately. Otherwise it returns true.
The default method calls `GRAY:STREAM-READ-BYTE-LOOKAHEAD`; this is always sufficient.
This function is a **CLISP** extension (see `EXT:READ-BYTE-WILL-HANG-P`).

(GRAY:STREAM-READ-BYTE-NO-HANG *stream*)
: To be called only if *stream*'s `STREAM-ELEMENT-TYPE` is `(UNSIGNED-BYTE 8)` or `(SIGNED-BYTE 8)`. Returns an `INTEGER`

or :EOF, like GRAY:STREAM-READ-BYTE, if that would return immediately. If GRAY:STREAM-READ-BYTE's value is not available immediately, returns NIL instead of waiting.
The default method calls GRAY:STREAM-READ-BYTE if GRAY:STREAM-READ-BYTE-LOOKAHEAD returns true; this is always sufficient.
This function is a **CLISP** extension (see EXT:READ-BYTE-NO-HANG).

(GRAY:STREAM-READ-BYTE-SEQUENCE *stream sequence* &OPTIONAL [*start* [*end* [*no-hang* [*interactive*]]]])

Fills the subsequence of *sequence* specified by :START and :END with integers consecutively read from *stream*. Returns the index of the first element of *sequence* that was not updated (= *end*, or < *end* if the stream reached its end).
sequence is an ARRAY of INTEGERs. *start* is a nonnegative INTEGER and defaults to 0. *end* is a nonnegative INTEGER or NIL and defaults to NIL, which stands for (LENGTH *sequence*). If *no-hang* is true, the function should avoid blocking and instead fill only as many elements as are immediately available. If *no-hang* is false and *interactive* is true, the function can block for reading the first byte but should avoid blocking for any further bytes.
The default method repeatedly calls GRAY:STREAM-READ-BYTE; this is always sufficient if speed does not matter.
This function is a **CLISP** extension (see EXT:READ-BYTE-SEQUENCE).

generic functions for binary output

(GRAY:STREAM-WRITE-BYTE *stream integer*)

Writes *integer*.
You must define a method for this function.

(GRAY:STREAM-WRITE-BYTE-SEQUENCE *stream sequence* &OPTIONAL [*start* [*end* [*no-hang* [*interactive*]]]])

Outputs the subsequence of *sequence* specified by :START and :END to *stream*
sequence is an ARRAY of INTEGERs. *start* is a nonnegative INTEGER and defaults to 0. *end* is a nonnegative INTEGER or NIL and defaults to NIL, which stands for (LENGTH *sequence*). If *no-hang* is true, the function should avoid blocking and instead output only as many elements as it can immediately proceed. If *no-hang* is

false and *`interactive`* is true, the function can block for writing the first byte but should avoid blocking for any further bytes.
The default method repeatedly calls `GRAY:STREAM-WRITE-BYTE`; this is always sufficient if speed does not matter.
This function is a **CLISP** extension (see `EXT:WRITE-BYTE-SEQUENCE`).

30.2. Class `EXT:FILL-STREAM`

As an example of the use of **"GRAY"** `STREAM`s, **CLISP** offers an additional class, `EXT:FILL-STREAM`. An instance of this class is a "formatting" `STREAM`, which makes the final output to the underlying stream look neat: indented and filled. An instance of `EXT:FILL-STREAM` is created like this:

```
(MAKE-INSTANCE 'EXT:FILL-STREAM :stream stream
               [:text-indent symbol-or-number]
               [:sexp-indent symbol-or-number-or-function
```

where

`stream`
: is the target `STREAM` where the output actually goes.

`symbol-or-number`
: is the variable whose value is the `INTEGER` text indentation or the indentation itself (defaults to 0).

`symbol-or-number-or-function`
: When `FORMAT` writes an S-expression to a `EXT:FILL-STREAM` using `~S`, and the expression's printed representation does not fit on the current line, it is printed on separate lines, ignoring the prescribed text indentation and preserving spacing. When this argument is non-`NIL`, the S-expression **is** indented by:

  `T`
  : the text indentation above;

  `SYMBOL`
  : `SYMBOL-VALUE` is the indentation;

  `INTEGER`
  : the indentation itself;

  `FUNCTION`

called with one argument, the text indentation, and the value is used as S-expression indentation; thus `IDENTITY` is equivalent to `T` above.

Defaults to `CUSTOM:*FILL-INDENT-SEXP*`, whose initial value is `1+`.

Warning

Note that, due to buffering, one must call `FORCE-OUTPUT` when done with the `EXT:FILL-STREAM` (and before changing the indent variable). The former is done automatically by the macro `(with-fill-stream (fill target-stream ...) ...)`.

Example 30.1. Example of `EXT:FILL-STREAM` usage

```
(defvar *my-indent-level*)
(with-output-to-string (out)
  (let ((*print-right-margin* 20)
        (*print-pretty* t)
        (*my-indent-level* 2))
    (with-fill-stream (fill out :text-indent '*my-indent-
      (format fill "~%this is some long sentence which wi
      (force-output fill)
      (let ((*my-indent-level* 5))
        (format fill "~%and    properly indented to the le
                :TEXT-INDENT 'symbol 'integer))
      (format fill "~%Don't forget  to call ~S on it, and
              'force-output 'with-fill-stream '(defun foo
⇒ "
  this is some long
  sentence which
  will be broken at
  spaces
     and properly
     indented to
     the level
     specified by
     the :TEXT-INDENT
     argument which
     can be a
```

```
SYMBOL
     or an INTEGER
     - cool!
  Don't forget to
  call FORCE-OUTPUT
  on it, and/or use
WITH-FILL-STREAM
  Pretty formatting
  of the
  S-expressions
  printed with ~S
  is preserved:
(DEFUN FOO (X Y Z)
 (IF X (+ Y Z)
  (* Y Z)))
"
```

Part III. Extensions Specific to CLISP

Table of Contents

Chapter 31. Platform Independent Extensions

Table of Contents

31.1. Customizing CLISP Process Initialization and Termination

31.1.1. Cradle to Grave

What is done when

1. **Initialization**

   a. Parse command line arguments until the first positional argument (see :SCRIPT in Section 31.2, “Saving an Image”).

   b. Load the memory image.

   c. Install internal signal handlers.

   d. Initialize time variables.

   e. Initialize locale-dependent encodings.

   f. Initialize stream variables.

   g. Initialize pathname variables.

   h. Initialize **“FFI”**.

   i. Initialize modules.

   j. Run all functions in CUSTOM:*INIT-HOOKS*.

   k. Say “hi”, unless suppressed by -q.

   l. Load RC file, unless suppressed by -norc.

2. **The actual work**

   Handle command line options: file loading and/or compilation, form evaluation, script execution, read-eval-print loop.

3. **Finalization (executed even on abnormal exit due to `kill`)**

   a. Unwind the `STACK`, executing cleanup forms in `UNWIND-PROTECT`.

   b. Run all functions in `CUSTOM:*FINI-HOOKS*`.

   c. Call `FRESH-LINE` on the standard streams.

   d. Say “bye” unless suppressed by `-q`.

   e. Wait for a keypress if requested by `-w`.

   f. Close all open `FILE-STREAM`s.

   g. Finalize modules.

   h. Close all open DLLs.

31.1.2. Customizing Initialization

`CUSTOM:*INIT-HOOKS*` is run like this:

```
(MAPC #'FUNCALL CUSTOM:*INIT-HOOKS*)
```

31.1.2.1. The difference between `CUSTOM:*INIT-HOOKS*` and init function

- `CUSTOM:*INIT-HOOKS*` are *always* run regardless of the command line options before even the banner is printed.
- The init function is run *only* if the read-eval-print loop is ever entered and just before the first prompt is printed.

31.1.3. Customizing Termination

CUSTOM:*FINI-HOOKS* is run like this:

```
(MAPC #'FUNCALL CUSTOM:*FINI-HOOKS*)
```

31.2. Saving an Image

The function (EXT:SAVEINITMEM &OPTIONAL (*filename* "lispinit.mem") &KEY :KEEP-GLOBAL-HANDLERS :QUIET :INIT-FUNCTION :LOCKED-PACKAGES :START-PACKAGE :EXECUTABLE :NORC :SCRIPT :DOCUMENTATION) saves the running **CLISP**'s memory to the file *filename*; extension #P".mem" is recommended (when *filename* does not have an extension, #P".mem" extension is automatically added unless the file being created is an executable).

:QUIET
: If this argument is not NIL, the startup banner and the good-bye message will be suppressed, as if by -q.
This is **not** recommended for interactive application delivery, please *append* your banner to ours (using init function) instead of *replacing* it.

:NORC
: If this argument is not NIL, the RC file loading will be suppressed, as if by -norc.

:INIT-FUNCTION
: This argument specifies a function that will be executed at startup of the saved image, before entering the standard read-eval-print loop (but after all other initialization, see Section 31.1.1, “Cradle to Grave”); thus, if you want to avoid the read-eval-print loop, you have to call EXT:EXIT at the end of the init function yourself (this does not prevent CUSTOM:*FINI-HOOKS* from being run).
See the manual for passing command line arguments to this function.
See also CUSTOM:*INIT-HOOKS* and CUSTOM:*FINI-HOOKS*.

:SCRIPT
: This options determines the handling of positional arguments when the image is invoked.

- If it is `T`, then the first positional argument is the script name and the rest is placed into `EXT:*ARGS*`, as described in Section 32.6.2, "Scripting with **CLISP**".
- It it is `NIL`, then all positional arguments are placed into `EXT:*ARGS*` to be handled by the init function.

This option defaults to `T` when init function is `NIL` and to `NIL` when init function is non-`NIL`.

`:DOCUMENTATION`

The description of what this image does, printed by the `-help-image` olption.
Defaults to `(DOCUMENTATION init function 'FUNCTION)`

`:LOCKED-PACKAGES`

This argument specifies the packages to lock before saving the image; this is convenient for application delivery, when you do not want your users to mess up your product. This argument defaults to `CUSTOM:*SYSTEM-PACKAGE-LIST*`.

`:START-PACKAGE`

This argument specifies the starting value of `*PACKAGE*` in the image being saved, and defaults to the current value of `*PACKAGE*`.

`:KEEP-GLOBAL-HANDLERS`

When non-`NIL`, the currently established global handlers (either with `EXT:SET-GLOBAL-HANDLER` or with `-on-error`) are inherited by the image. Defaults to `NIL`, so that

```
$ clisp -i myfile -x '(EXT:SAVEINITMEM)'
```

will produce an image without any global handlers inherited from the batch mode of the above command.

`:EXECUTABLE`

When non-`NIL`, the saved file will be an standalone executable. In this case, the `#P".mem"` extension is not added. On **Win32** and **Cygwin** the extension `#P".exe"` is added instead.

You can use this memory image with the `-M` option. On **UNIX** systems, you may compress it with GNU **gzip** to save disk space.

Image Portability

Memory images are **not** portable across different platforms (in contrast with platform-independent `#P".fas"` files). They are **not** even portable across linking sets: image saved using the **full** linking set cannot be used with the **base** runtime:

```
$ clisp -K full -x '(EXT:SAVEINITMEM)'
$ clisp -K base -M lispinit.mem
base/lisp.run: initialization file `lispinit.mem' was
```

31.3. Quitting CLISP

The functions

```
(EXT:EXIT &OPTIONAL status)
(EXT:QUIT &OPTIONAL status)
(EXT:BYE &OPTIONAL status)
```

- all synonymous - terminate **CLISP**. If *`status`* is non-`NIL`, **CLISP** aborts with the supplied numeric error *`status`*, i.e., the OS environment is informed that the **CLISP** session did not succeed.

Final delimiters also terminate **CLISP**.

31.4. Internationalization of CLISP

31.4.1. The Language

Internationalization (“i18n”)
: preparing a program so that it can use multiple national languages and national cultural conventions without requiring further source code changes.

Localization (“l10n”)
: providing the data - mostly textual translations - necessary for an internationalized program to work in a particular language and with particular cultural conventions.

CLISP is internationalized, and is localized for the languages English, German, French, Spanish, Dutch, Russian, and Danish. **CLISP** also supports internationalized Lisp programs, through GNU gettext, see Section 33.2, “Internationalization of User Programs”.

User programs can also be internationalized, see Section 33.2, “Internationalization of User Programs”.

31.4.1. The Language

> ### Warning
>
> The facilities described in this section will work only for the languages for which **CLISP** itself is already localized.

The language **CLISP** uses to communicate with the user can be one of

`ENGLISH`

`DEUTSCH` (i.e., German)

`FRANÇAIS` (i.e., French)

`ESPAÑOL` (i.e., Spanish)

`NEDERLANDS` (i.e., Dutch)

`РУССКИЙ` (i.e. Russian)

`DANSK` (i.e., Danish)

This is controlled by the `SYMBOL-MACRO` ***`CUSTOM:*CURRENT-LANGUAGE*`***, which can be set at run time as well as using the `-L` start-up option. If you wish to change the locale directory at run time too, you can do that by setting `CUSTOM:*CURRENT-LANGUAGE*` to a `CONS` cell, whose `CAR` is the language (a `SYMBOL`, one of the above), and whose `CDR` is the new locale directory.

More languages can be defined through the macro ***`I18N:DEFLANGUAGE`***: `(I18N:DEFLANGUAGE` *`language`*`)`. For such an additional language to take effect, you must install the corresponding message catalog, or translate the messages yourself, using GNU gettext and Emacs (or XEmacs) po-mode.

This works only for strings. For arbitrary language-dependent Lisp objects, you define one through the macro ***I18N:DEFINTERNATIONAL***: `(I18N:DEFINTERNATIONAL symbol &OPTIONAL (default-language T))` and add language-dependent values through the macro ***I18N:DEFLOCALIZED***: `(I18N:DEFLOCALIZED symbol language value-form)` (One such form for each language. Languages without an assigned value will be treated like the default-language.) You can then access the localized value by calling ***I18N:LOCALIZED***: `(I18N:LOCALIZED symbol &OPTIONAL language)`

31.5. Encodings

31.5.1. Introduction

An “encoding” describes the correspondence between `CHARACTER`s and raw bytes during input/output via `STREAM`s with `STREAM-ELEMENT-TYPE` `CHARACTER`.

An `EXT:ENCODING` is an object composed of the following facets:

character set

This denotes both the set of `CHARACTER`s that can be represented and passed through the I/O channel, and the way these characters translate into raw bytes, i.e., the map between sequences of `CHARACTER` and `(UNSIGNED-BYTE 8)` in the form of `STRING`s and `(VECTOR (UNSIGNED-BYTE 8))` as well as character and byte `STREAM`s. In this context, for example, `CHARSET:UTF-8` and

CHARSET:UCS-4 are considered different, although they can represent the same set of characters.

line terminator mode

This denotes the way newline characters are represented.

EXT:ENCODINGs are also TYPEs. As such, they represent the set of characters encodable in the character set. In this context, the way characters are translated into raw bytes is ignored, and the line terminator mode is ignored as well. TYPEP and SUBTYPEP can be used on encodings:

```
(SUBTYPEP CHARSET:UTF-8 CHARSET:UTF-16)
⇒ T ;
⇒ T
(SUBTYPEP CHARSET:UTF-16 CHARSET:UTF-8)
⇒ T ;
⇒ T
(SUBTYPEP CHARSET:ASCII CHARSET:ISO-8859-1)
⇒ T ;
⇒ T
(SUBTYPEP CHARSET:ISO-8859-1 CHARSET:ASCII)
⇒ NIL ;
⇒ T
```

31.5.2. Character Sets

Platform Dependent: Only in CLISP built without compile-time flag UNICODE

Only one character set is understood: the platform's native (8-bit) character set. See Chapter 13, *Characters [CLHS-13]*.

Platform Dependent: Only in CLISP built with compile-time flag UNICODE

The following character sets are supported, as values of the corresponding (constant) symbol in the **“CHARSET”** package:

Symbols in package “CHARSET”

1. UCS-2 ≡ UNICODE-16 ≡ UNICODE-16-BIG-ENDIAN, the 16-bit basic multilingual plane of the **UNICODE** character set. Every character is represented as two bytes.
2. UNICODE-16-LITTLE-ENDIAN

3. `UCS-4` ≡ `UNICODE-32` ≡ `UNICODE-32-BIG-ENDIAN`, the 21-bit **UNICODE** character set. Every character is represented as four bytes. This encoding is used by **CLISP** internally.
4. `UNICODE-32-LITTLE-ENDIAN`
5. `UTF-8`, the 21-bit **UNICODE** character set. Every character is represented as one to four bytes. **ASCII** characters represent themselves and need one byte per character. Most Latin/Greek/Cyrillic/Hebrew characters need two bytes per character. Most other characters need three bytes per character, and the rarely used remaining characters need four bytes per character. This is therefore, in general, the most space-efficient encoding of all of Unicode.
6. `UTF-16`, the 21-bit **UNICODE** character set. Every character in the 16-bit basic multilingual plane is represented as two bytes, and the rarely used remaining characters need four bytes per character. This character set is only available on platforms with GNU libc or GNU libiconv.
7. `UTF-7`, the 21-bit **UNICODE** character set. This is a stateful 7-bit encoding. Not all **ASCII** characters represent themselves. This character set is only available on platforms with GNU libc or GNU libiconv.
8. `JAVA`, the 21-bit **UNICODE** character set. **ASCII** characters represent themselves and need one byte per character. All other characters of the basic multilingual plane are represented by `\u`*`nnnn`* sequences (*`nnnn`* a hexadecimal number) and need 6 bytes per character. The remaining characters are represented by `\u`*`xxxx`*`\u`*`yyyy`* and need 12 bytes per character. While this encoding is very comfortable for editing Unicode files using only **ASCII**-aware tools and editors, it cannot faithfully represent all **UNICODE** text. Only text which does not contain `\u` (backslash followed by lowercase Latin u) can be faithfully represented by this encoding.
9. `ASCII`, the well-known US-centric 7-bit character set (American Standard Code for Information Interchange - **ASCII**).
10.

    `ISO-8859-1`, an extension of the ASCII character set, suitable for the Afrikaans, Albanian, Basque, Breton, Catalan, Cornish, Danish, Dutch, English, Færoese, Finnish, French, Frisian, Galician, German, Greenlandic, Icelandic, Irish, Italian, Latin,

Luxemburgish, Norwegian, Portuguese, Ræto-Romanic, Scottish, Spanish, and Swedish languages.
This encoding has the nice property that

```
(LOOP :for i :from 0 :to CHAR-CODE-LIMIT :for c =
  :always (OR (NOT (TYPEP c CHARSET:ISO-8859-1))
              (EQUALP (EXT:CONVERT-STRING-TO-BYTES
                      (VECTOR i)))))
⇒ T
```

i.e., it is compatible with **CLISP** `CODE-CHAR`/`CHAR-CODE` in its own domain.

11. `ISO-8859-2`, an extension of the ASCII character set, suitable for the Croatian, Czech, German, Hungarian, Polish, Slovak, Slovenian, and Sorbian languages.
12. `ISO-8859-3`, an extension of the ASCII character set, suitable for the Esperanto and Maltese languages.
13. `ISO-8859-4`, an extension of the ASCII character set, suitable for the Estonian, Latvian, Lithuanian and Sami (Lappish) languages.
14. `ISO-8859-5`, an extension of the ASCII character set, suitable for the Bulgarian, Byelorussian, Macedonian, Russian, Serbian, and Ukrainian languages.
15. `ISO-8859-6`, suitable for the Arabic language.
16. `ISO-8859-7`, an extension of the ASCII character set, suitable for the Greek language.
17. `ISO-8859-8`, an extension of the ASCII character set, suitable for the Hebrew language (without punctuation).
18. `ISO-8859-9`, an extension of the ASCII character set, suitable for the Turkish language.
19. `ISO-8859-10`, an extension of the ASCII character set, suitable for the Estonian, Icelandic, Inuit (Greenlandic), Latvian, Lithuanian, and Sami (Lappish) languages.
20. `ISO-8859-13`, an extension of the ASCII character set, suitable for the Estonian, Latvian, Lithuanian, Polish and Sami (Lappish) languages.
21. `ISO-8859-14`, an extension of the ASCII character set, suitable for the Irish Gælic, Manx Gælic, Scottish Gælic, and Welsh languages.

22. `ISO-8859-15`, an extension of the ASCII character set, suitable for the ISO-8859-1 languages, with improvements for French, Finnish and the Euro.
23. `ISO-8859-16` an extension of the ASCII character set, suitable for the Rumanian language.
24. `KOI8-R`, an extension of the ASCII character set, suitable for the Russian language (very popular, especially on the internet).
25. `KOI8-U`, an extension of the ASCII character set, suitable for the Ukrainian language (very popular, especially on the internet).
26. `KOI8-RU`, an extension of the ASCII character set, suitable for the Russian language. This character set is only available on platforms with GNU libiconv.
27. `JIS_X0201`, a character set for the Japanese language.
28. `MAC-ARABIC`, a platform specific extension of the ASCII character set.
29. `MAC-CENTRAL-EUROPE`, a platform specific extension of the ASCII character set.
30. `MAC-CROATIAN`, a platform specific extension of the ASCII character set.
31. `MAC-CYRILLIC`, a platform specific extension of the ASCII character set.
32. `MAC-DINGBAT`, a platform specific character set.
33. `MAC-GREEK`, a platform specific extension of the ASCII character set.
34. `MAC-HEBREW`, a platform specific extension of the ASCII character set.
35. `MAC-ICELAND`, a platform specific extension of the ASCII character set.
36. `MAC-ROMAN` ≡ `MACINTOSH`, a platform specific extension of the ASCII character set.
37. `MAC-ROMANIA`, a platform specific extension of the ASCII character set.
38. `MAC-SYMBOL`, a platform specific character set.
39. `MAC-THAI`, a platform specific extension of the ASCII character set.
40. `MAC-TURKISH`, a platform specific extension of the ASCII character set.
41. `MAC-UKRAINE`, a platform specific extension of the ASCII character set.

42. `CP437`, a DOS oldie, a platform specific extension of the ASCII character set.
43. `CP437-IBM`, an IBM variant of `CP437`.
44. `CP737`, a DOS oldie, a platform specific extension of the ASCII character set, meant to be suitable for the Greek language.
45. `CP775`, a DOS oldie, a platform specific extension of the ASCII character set, meant to be suitable for some Baltic languages.
46. `CP850`, a DOS oldie, a platform specific extension of the ASCII character set.
47. `CP852`, a DOS oldie, a platform specific extension of the ASCII character set.
48. `CP852-IBM`, an IBM variant of `CP852`.
49. `CP855`, a DOS oldie, a platform specific extension of the ASCII character set, meant to be suitable for the Russian language.
50. `CP857`, a DOS oldie, a platform specific extension of the ASCII character set, meant to be suitable for the Turkish language.
51. `CP860`, a DOS oldie, a platform specific extension of the ASCII character set, meant to be suitable for the Portuguese language.
52. `CP860-IBM`, an IBM variant of `CP860`.
53. `CP861`, a DOS oldie, a platform specific extension of the ASCII character set, meant to be suitable for the Icelandic language.
54. `CP861-IBM`, an IBM variant of `CP861`.
55. `CP862`, a DOS oldie, a platform specific extension of the ASCII character set, meant to be suitable for the Hebrew language.
56. `CP862-IBM`, an IBM variant of `CP862`.
57. `CP863`, a DOS oldie, a platform specific extension of the ASCII character set.
58. `CP863-IBM`, an IBM variant of `CP863`.
59. `CP864`, a DOS oldie, meant to be suitable for the Arabic language.
60. `CP864-IBM`, an IBM variant of `CP864`.
61. `CP865`, a DOS oldie, a platform specific extension of the ASCII character set, meant to be suitable for some Nordic languages.
62. `CP865-IBM`, an IBM variant of `CP865`.
63. `CP866`, a DOS oldie, a platform specific extension of the ASCII character set, meant to be suitable for the Russian language.
64. `CP869`, a DOS oldie, a platform specific extension of the ASCII character set, meant to be suitable for the Greek language.
65. `CP869-IBM`, an IBM variant of `CP869`.

66. `CP874`, a DOS oldie, a platform specific extension of the ASCII character set, meant to be suitable for the Thai language.
67. `CP874-IBM`, an IBM variant of `CP874`.
68. `WINDOWS-1250` ≡ `CP1250`, a platform specific extension of the ASCII character set, heavily incompatible with ISO-8859-2.
69. `WINDOWS-1251` ≡ `CP1251`, a platform specific extension of the ASCII character set, heavily incompatible with ISO-8859-5, meant to be suitable for the Russian language.
70. `WINDOWS-1252` ≡ `CP1252`, a platform specific extension of the ISO-8859-1 character set.
71. `WINDOWS-1253` ≡ `CP1253`, a platform specific extension of the ASCII character set, gratuitously incompatible with ISO-8859-7, meant to be suitable for the Greek language.
72. `WINDOWS-1254` ≡ `CP1254`, a platform specific extension of the ISO-8859-9 character set.
73. `WINDOWS-1255` ≡ `CP1255`, a platform specific extension of the ASCII character set, gratuitously incompatible with ISO-8859-8, suitable for the Hebrew language. This character set is only available on platforms with GNU libc or GNU libiconv.
74. `WINDOWS-1256` ≡ `CP1256`, a platform specific extension of the ASCII character set, meant to be suitable for the Arabic language.
75. `WINDOWS-1257` ≡ `CP1257`, a platform specific extension of the ASCII character set.
76. `WINDOWS-1258` ≡ `CP1258`, a platform specific extension of the ASCII character set, meant to be suitable for the Vietnamese language. This character set is only available on platforms with GNU libc or GNU libiconv.
77. `HP-ROMAN8`, a platform specific extension of the ASCII character set.
78. `NEXTSTEP`, a platform specific extension of the ASCII character set.
79. `EUC-JP`, a multibyte character set for the Japanese language. This character set is only available on platforms with GNU libc or GNU libiconv.
80. `SHIFT-JIS`, a multibyte character set for the Japanese language. This character set is only available on platforms with GNU libc or GNU libiconv.
81. `CP932`, a Microsoft variant of `SHIFT-JIS`. This character set is only available on platforms with GNU libc or GNU libiconv.

82. `ISO-2022-JP`, a stateful 7-bit multibyte character set for the Japanese language. This character set is only available on platforms with GNU libc or GNU libiconv.
83. `ISO-2022-JP-2`, a stateful 7-bit multibyte character set for the Japanese language. This character set is only available on platforms with GNU libc 2.3 or newer or GNU libiconv.
84. `ISO-2022-JP-1`, a stateful 7-bit multibyte character set for the Japanese language. This character set is only available on platforms with GNU libiconv.
85. `EUC-CN`, a multibyte character set for simplified Chinese. This character set is only available on platforms with GNU libc or GNU libiconv.
86. `HZ`, a stateful 7-bit multibyte character set for simplified Chinese. This character set is only available on platforms with GNU libiconv.
87. `GBK`, a multibyte character set for Chinese, This character set is only available on platforms with GNU libc or GNU libiconv.
88. `CP936`, a Microsoft variant of `GBK`. This character set is only available on platforms with GNU libc or GNU libiconv.
89. `GB18030`, a multibyte character set for Chinese, This character set is only available on platforms with GNU libc or GNU libiconv.
90. `EUC-TW`, a multibyte character set for traditional Chinese. This character set is only available on platforms with GNU libc or GNU libiconv.
91. `BIG5`, a multibyte character set for traditional Chinese. This character set is only available on platforms with GNU libc or GNU libiconv.
92. `CP950`, a Microsoft variant of `BIG5`. This character set is only available on platforms with GNU libc or GNU libiconv.
93. `BIG5-HKSCS`, a multibyte character set for traditional Chinese. This character set is only available on platforms with GNU libc or GNU libiconv.
94. `ISO-2022-CN`, a stateful 7-bit multibyte character set for Chinese. This character set is only available on platforms with GNU libc or GNU libiconv.
95. `ISO-2022-CN-EXT`, a stateful 7-bit multibyte character set for Chinese. This character set is only available on platforms with GNU libc or GNU libiconv.

96. `EUC-KR`, a multibyte character set for Korean. This character set is only available on platforms with GNU libc or GNU libiconv.
97. `CP949`, a Microsoft variant of `EUC-KR`. This character set is only available on platforms with GNU libc or GNU libiconv.
98. `ISO-2022-KR`, a stateful 7-bit multibyte character set for Korean. This character set is only available on platforms with GNU libc or GNU libiconv.
99. `JOHAB`, a multibyte character set for Korean used mostly on **DOS**. This character set is only available on platforms with GNU libc or GNU libiconv.
100. `ARMSCII-8`, an extension of the ASCII character set, suitable for the Armenian. This character set is only available on platforms with GNU libc or GNU libiconv.
101. `GEORGIAN-ACADEMY`, an extension of the ASCII character set, suitable for the Georgian. This character set is only available on platforms with GNU libc or GNU libiconv.
102. `GEORGIAN-PS`, an extension of the ASCII character set, suitable for the Georgian. This character set is only available on platforms with GNU libc or GNU libiconv.
103. `TIS-620`, an extension of the ASCII character set, suitable for the Thai. This character set is only available on platforms with GNU libc or GNU libiconv.
104. `MULELAO-1`, an extension of the ASCII character set, suitable for the Laotian. This character set is only available on platforms with GNU libiconv.
105. `CP1133`, an extension of the ASCII character set, suitable for the Laotian. This character set is only available on platforms with GNU libc or GNU libiconv.
106. `VISCII`, an extension of the ASCII character set, suitable for the Vietnamese. This character set is only available on platforms with GNU libc or GNU libiconv.
107. `TCVN`, an extension of the ASCII character set, suitable for the Vietnamese. This character set is only available on platforms with GNU libc or GNU libiconv.
108. `BASE64`, encodes arbitrary byte sequences with 64 **ASCII** characters

    ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqr

as specified by MIME; 3 bytes are encoded with 4 characters, line breaks are inserted after every 76 characters.
While this is not a traditional character set (i.e., it does not map a set of characters in a natural language into bytes), it does define a map between arbitrary byte sequences and certain character sequences, so it falls naturally into the EXT:ENCODING class.

Platform Dependent: Only on GNU systems with GNU libc 2.2 or better and other systems (UNIX and Win32) on which the GNU libiconv C library has been installed

The character sets provided by the library function iconv can also be used as encodings. To create such an encoding, call EXT:MAKE-ENCODING with the character set name (a string) as the :CHARSET argument.
When an EXT:ENCODING is available both as a built-in and through iconv, the built-in is used, because it is more efficient and available across all platforms.
These encodings are not assigned to global variables, since there is no portable way to get the list of all character sets supported by iconv.
On standard-compliant **UNIX** systems (e.g., GNU systems, such as GNU/**Linux** and GNU/**Hurd**) and on systems with GNU libiconv you get this list by calling the *program*: **iconv -l**.
The reason we use only GNU libc 2.2 or GNU libiconv is that the other iconv implementations are broken in various ways and we do not want to deal with random **CLISP** crashes caused by those bugs. If your system supplies an iconv implementation which passes the GNU libiconv's test suite, please report that to <clisp-list@lists.sourceforge.net> (http://lists.sourceforge.net/lists/listinfo/clisp-list) and a future **CLISP** version will use iconv on your system.

31.5.3. Line Terminators

The line terminator mode can be one of the following three keywords:

:UNIX
: Newline is represented by the **ASCII** LF character (U000A).

:MAC

Newline is represented by the ASCII CR character (U000D).

:DOS
: Newline is represented by the ASCII CR followed by the ASCII LF.

Windows programs typically use the :DOS line terminator, sometimes they also accept :UNIX line terminators or produce :MAC line terminators.

The HTTP protocol also requires :DOS line terminators.

The line terminator mode is relevant only for output (writing to a file/pipe/socket STREAM). During input, all three kinds of line terminators are recognized. See also Section 13.8, “Treatment of Newline during Input and Output [CLHS-13.1.8]”.

31.5.4. Function EXT:MAKE-ENCODING

The function (EXT:MAKE-ENCODING &KEY :CHARSET :LINE-TERMINATOR :INPUT-ERROR-ACTION :OUTPUT-ERROR-ACTION) returns an EXT:ENCODING. The :CHARSET argument may be an encoding, a string, or :DEFAULT. The possible values for the line terminator argument are the keywords :UNIX, :MAC, :DOS.

The :INPUT-ERROR-ACTION argument specifies what happens when an invalid byte sequence is encountered while converting bytes to characters. Its value can be :ERROR, :IGNORE or a character to be used instead. The UNICODE character #\uFFFD is typically used to indicate an error in the input sequence.

The :OUTPUT-ERROR-ACTION argument specifies what happens when an invalid character is encountered while converting characters to bytes. Its value can be :ERROR, :IGNORE, a byte to be used instead, or a character to be used instead. The UNICODE character #\uFFFD can be used here only if it is encodable in the character set.

31.5.5. Function EXT:ENCODING-CHARSET

Platform Dependent: Only in CLISP built with compile-time flag UNICODE

The function (EXT:ENCODING-CHARSET *encoding*) returns the charset of the *encoding*, as a SYMBOL or a STRING.

Warning

(STRING (EXT:ENCODING-CHARSET *encoding*)) is not necessarily a valid MIME name.

31.5.6. Default encodings

Besides every file/pipe/socket STREAM containing an encoding, the following SYMBOL-MACRO places contain global EXT:ENCODINGs:

SYMBOL-MACRO CUSTOM:*DEFAULT-FILE-ENCODING*. The SYMBOL-MACRO place CUSTOM:*DEFAULT-FILE-ENCODING* is the encoding used for new file/pipe/socket STREAM, when no :EXTERNAL-FORMAT argument was specified.

Platform Dependent: Only in CLISP built with compile-time flag UNICODE

The following are SYMBOL-MACRO places.

CUSTOM:*PATHNAME-ENCODING*
: is the encoding used for pathnames in the file system. Normally, this should be a 1:1 encoding. Its line terminator mode is ignored.

CUSTOM:*TERMINAL-ENCODING*
: is the encoding used for communication with the terminal, in particular by *TERMINAL-IO*.

CUSTOM:*MISC-ENCODING*
: is the encoding used for access to environment variables, command line options, and the like. Its line terminator mode is ignored.

CUSTOM:*FOREIGN-ENCODING*
: is the encoding for strings passed through the **"FFI"** (some platforms only). If it is a 1:1 encoding, i.e. an encoding in which every character is represented by one byte, it is also used for passing characters through the **"FFI"**.

The default encoding objects are initialized according to `-E`*`domain encoding`*.

Reminder

You have to use `EXT:LETF`/`EXT:LETF*` for `SYMBOL-MACRO`s; `LET`/`LET*` will **not** work!

31.5.6.1. Default line terminator

The line terminator facet of the above `EXT:ENCODING`s is determined by the following logic: since **CLISP** understands all possible line terminators on *input* (see Section 13.8, "Treatment of Newline during Input and Output [CLHS-13.1.8]"), all that matters is what line terminator do most *other* programs expect?

Platform Dependent: UNIX platform only.
: If a non-0 `O_BINARY` **cpp** constant is defined, we assume that the OS distinguishes between text and binary files, and, since the encodings are relevant only for text files, we thus use `:DOS`; otherwise the default is `:UNIX`.

Platform Dependent: Win32 platform only.
: Since most **Win32** programs expect CRLF, the default line terminator is `:DOS`.

This boils down to the following code in `src/encoding.d`:

```
#if defined(WIN32) || (defined(UNIX) && (O_BINARY != 0))
```

Default line terminator on Cygwin

Both of the above tests pass on **Cygwin**, so the default line terminator is `:DOS`. If you so desire, you can change it in your RC file.

31.5.7. Converting between strings and byte vectors

Encodings can also be used to convert directly between strings and their corresponding byte vector representation according to that encoding.

(EXT:CONVERT-STRING-FROM-BYTES *vector encoding* &KEY :START :END)
: converts the subsequence of *vector* (a `(VECTOR (UNSIGNED-BYTE 8))`) from *start* to *end* to a `STRING`, according to the given *encoding*, and returns the resulting string.

(EXT:CONVERT-STRING-TO-BYTES *string encoding* &KEY :START :END)
: converts the subsequence of *string* from *start* to *end* to a `(VECTOR (UNSIGNED-BYTE 8))`, according to the given *encoding*, and returns the resulting byte vector.

31.6. Generic streams

This interface is **CLISP**-specific and now obsolete. Please use the Gray streams interface instead.

Generic streams are user programmable streams. The programmer interface:

(gstream:make-generic-stream *controller*)
: returns a generic stream.

(gstream:generic-stream-controller *stream*)
: returns a private object to which generic stream methods dispatch. The typical usage is to retrieve the object originally provided by the user in `gstream:make-generic-stream`.

(gstream:generic-stream-p *stream*)

determines whether a stream is a generic stream, returning `T` if it is, `NIL` otherwise.

In order to specify the behavior of a generic stream, the user must define **CLOS** methods on the following **CLOS** generic functions. The function `gstream:generic-stream-`*`xyz`* corresponds to the **Common Lisp** function *`xyz`*. They all take a controller and some number of arguments.

`(gstream:generic-stream-read-char `*`controller`*`)`
: Returns and consumes the next character, `NIL` at end of file. Takes one argument, the controller object.

`(gstream:generic-stream-peek-char `*`controller`*`)`
: Returns the next character, `NIL` at end of file. A second value indicates whether the side effects associated with consuming the character were executed: `T` means that a full `READ-CHAR` was done, `NIL` means that no side effects were done. Takes one argument, the controller object.

`(gstream:generic-stream-read-byte `*`controller`*`)`
: Returns and consumes the next integer, `NIL` at end of file. Takes one argument, the controller object.

`(gstream:generic-stream-read-char-will-hang-p `*`controller`*`)`
: This generic function is used to query the stream's input status. It returns `NIL` if `gstream:generic-stream-read-char` and `gstream:generic-stream-peek-char` will certainly return immediately. Otherwise it returns true.

`(gstream:generic-stream-write-char `*`controller char`*`)`
: The first argument is the controller object. The second argument is the character to be written.

`(gstream:generic-stream-write-byte `*`controller by`*`)`
: The first argument is the controller object. The second argument is the integer to be written.

`(gstream:generic-stream-write-string `*`controller string start length`*`)`
: Writes the subsequence of *`string`* starting from *`start`* of length *`length`*. The first argument is the controller object.

`(gstream:generic-stream-clear-input `*`controller`*`)`
`(gstream:generic-stream-clear-output `*`controller`*`)`
`(gstream:generic-stream-finish-output `*`controller`*`)`
`(gstream:generic-stream-force-output `*`controller`*`)`
`(gstream:generic-stream-close `*`controller`*`)`
: Take one argument, the controller object.

31.7. Weak Objects

Recall two terms: An object is called “"alive"” as long as it can be retrieved by the user or program, through any kind of references, starting from global and local variables. (Objects that consume no heap storage, also known as “"immediate objects"”, such as `CHARACTER`s, `FIXNUM`s, and `SHORT-FLOAT`s, are alive indefinitely.) An object is said to be garbage-collected when its storage is reclaimed, at some moment after it becomes “"dead"”.

31.7.1. Weak Pointers

A `EXT:WEAK-POINTER` is an object holding a reference to a given object, without keeping the latter from being garbage-collected.

Weak Pointer API

`(EXT:MAKE-WEAK-POINTER` *`value`*`)`
: returns a fresh `EXT:WEAK-POINTER` referring to *`value`*.

`(EXT:WEAK-POINTER-P` *`object`*`)`
: returns true if the *`object`* is of type `EXT:WEAK-POINTER`.

`(EXT:WEAK-POINTER-VALUE` *`weak-pointer`*`)`
: returns two values: The original value and `T`, if the value has not yet been garbage-collected, else `NIL` and `NIL`. It is `SETF`-able: you can change the value that the weak pointer points to.

Weak pointers are useful for notification-based communication protocols between software modules, e.g. when a change to an object *x* requires a notification to an object *y*, as long as *y* is alive.

31.7.2. Weak Lists

A `EXT:WEAK-LIST` is an ordered collection of references to objects that does **not** keep the objects from being garbage-collected. It is semantically equivalent to a list of `EXT:WEAK-POINTER`s, however with a more efficient in-memory representation than a plain list of `EXT:WEAK-POINTER`s would be.

Weak List API

`(EXT:MAKE-WEAK-LIST list)`
: creates a `EXT:WEAK-LIST` pointing to each of the elements in the given *`list`*.

`(EXT:WEAK-LIST-P object)`
: returns true if the *`object`* is of type `EXT:WEAK-LIST`.

`(EXT:WEAK-LIST-LIST weak-list)`
: returns a `LIST` of those objects from the *`weak-list`* that are still alive.

`(SETF (EXT:WEAK-LIST-LIST weak-list) list)`
: replaces the list of objects stored by the *`weak-list`*.

Weak lists are useful for notification based communication protocols between software modules, e.g. when a change to an object x requires a notification to objects k_1, k_2, ..., as long as such a particular k_n is alive.

A `EXT:WEAK-LIST` with a single element is semantically equivalent to a single `EXT:WEAK-POINTER`.

31.7.3. Weak “And” Relations

A weak “and” relation is an ordered collection of references to objects, that does **not** keep the objects from being garbage-collected, and which allows access to all the objects as long as all of them are still alive. As soon as one of them is garbage-collected, the entire collection of objects becomes empty.

Weak “And” Relation API

`(EXT:MAKE-WEAK-AND-RELATION list)`

creates a EXT:WEAK-AND-RELATION between the objects in the given *list*.

(EXT:WEAK-AND-RELATION-P *object*)

returns true if the *object* is of type EXT:WEAK-AND-RELATION.

(EXT:WEAK-AND-RELATION-LIST *weak-and-relation*)

returns the list of objects stored in the *weak-and-relation*. The returned list must not be destructively modified.

EXT:WEAK-AND-RELATIONs are useful to model relations between objects that become worthless when one of the objects dies.

A EXT:WEAK-AND-RELATION with a single element is semantically equivalent to a EXT:WEAK-POINTER.

31.7.4. Weak “Or” Relations

A weak “or” relation is an ordered collection of references to objects, that keeps all objects from being garbage-collected as long as one of them is still alive. In other words, each of them keeps all others among them from being garbage-collected. When all of them are unreferenced, the collection of objects becomes empty.

Weak “Or” Relation API

(EXT:MAKE-WEAK-OR-RELATION *list*)

creates a EXT:WEAK-OR-RELATION between the objects in the given *list*.

(EXT:WEAK-OR-RELATION-P *object*)

returns true if the *object* is of type EXT:WEAK-OR-RELATION.

(EXT:WEAK-OR-RELATION-LIST *weak-or-relation*)

returns the list of objects stored in the *weak-or-relation*. The returned list must not be destructively modified.

EXT:WEAK-OR-RELATIONs are useful to model relations between objects that do not become worthless when one of the objects dies.

A EXT:WEAK-OR-RELATION with a single element is semantically equivalent to a EXT:WEAK-POINTER.

31.7.5. Weak Associations

A weak association is a mapping from an object called *key* to an object called *value*, that exists as long as the key is alive. In other words, as long as the key is alive, it keeps the value from being garbage-collected.

Weak Association API

(EXT:MAKE-WEAK-MAPPING *key* *value*)
: creates a EXT:WEAK-MAPPING.

(EXT:WEAK-MAPPING-P *object*)
: returns true if the object is of type EXT:WEAK-MAPPING.

(EXT:WEAK-MAPPING-PAIR *weak-mapping*)
: returns three values: the original key, the original value, and T, if the key has not yet been garbage-collected, else NIL, NIL, NIL.

(EXT:WEAK-MAPPING-VALUE *weak-mapping*)
: returns the value, if the key has not yet been garbage-collected, else NIL.

(SETF (EXT:WEAK-MAPPING-VALUE *weak-mapping*) *value*)
: replaces the value stored in the *weak-mapping*. It has no effect when the key has already been garbage-collected.

Weak associations are useful to supplement objects with additional information that is stored outside of the object.

31.7.6. Weak “And” Mappings

A weak “and” mapping is a mapping from a tuple of objects called *keys* to an object called *value*, that does **not** keep the keys from being garbage-collected and that exists as long as all keys are alive. As soon as one of the keys is garbage-collected, the entire mapping goes away.

Weak “And” Mapping API

(EXT:MAKE-WEAK-AND-MAPPING *keys* *value*)
: creates a EXT:WEAK-AND-MAPPING between the *keys* objects in the given list and the given *value*. The *keys* list must be non-empty.

(EXT:WEAK-AND-MAPPING-P *object*)
: returns true if the *object* is of type EXT:WEAK-AND-MAPPING.

(EXT:WEAK-AND-MAPPING-PAIR *weak-and-mapping*)

returns three values: the list of keys, the value, and T, if none of the keys have been garbage-collected, else NIL, NIL, NIL. The returned keys list must not be destructively modified.

(EXT:WEAK-AND-MAPPING-VALUE *weak-and-mapping*)

returns the value, if none of the keys have been garbage-collected, else NIL.

(SETF (EXT:WEAK-AND-MAPPING-VALUE *weak-and-mapping*) *value*)

replaces the value stored in the *weak-and-mapping*. It has no effect when some key has already been garbage-collected.

EXT:WEAK-AND-MAPPINGs are useful to model properties of sets of objects that become worthless when one of the objects dies.

A EXT:WEAK-AND-MAPPING with a single key is semantically equivalent to a weak association.

31.7.7. Weak “Or” Mappings

A weak “or” mapping is a mapping from a tuple of objects called *keys* to an object called *value*, that keeps all keys and the value from being garbage-collected as long as one of the keys is still alive. In other words, each of the keys keeps all others among them and the value from being garbage-collected. When all of them are unreferenced, the entire mapping goes away.

Weak “Or” Mapping API

(EXT:MAKE-WEAK-OR-MAPPING *keys value*)

creates a EXT:WEAK-OR-MAPPING between the *keys* objects in the given list and the given *value*. The *keys* list must be non-empty.

(EXT:WEAK-OR-MAPPING-P *object*)

returns true if the *object* is of type EXT:WEAK-OR-MAPPING.

(EXT:WEAK-OR-MAPPING-PAIR *weak-or-mapping*)

returns three values: the list of keys, the value, and T, if the keys have not yet been garbage-collected, else NIL, NIL, NIL. The returned keys list must not be destructively modified.

(EXT:WEAK-OR-MAPPING-VALUE *weak-or-mapping*)

returns the value, if the keys have not yet been garbage-collected, else NIL.

(SETF (EXT:WEAK-OR-MAPPING-VALUE *weak-or-mapping*) *value*)
: replaces the value stored in the `weak-or-mapping`. It has no effect when the keys have already been garbage-collected.

`EXT:WEAK-OR-MAPPING`s are useful to model properties of sets of objects that do not become worthless when one of the objects dies.

A `EXT:WEAK-OR-MAPPING` with a single key is semantically equivalent to a weak association.

31.7.8. Weak Association Lists

A weak association list is an ordered collection of pairs, each pair being built from an object called *key* and an object called *value*. The lifetime of each pair depends on the type of the weak association list:

:KEY
: The pair exists as long as the *key* is not garbage-collected. As long as the *key* is alive, it prevents the *value* from being garbage-collected.

:VALUE
: The pair exists as long as the *value* is not garbage-collected. As long as the *value* is alive, it prevents the *key* from being garbage-collected.

:KEY-AND-VALUE
: The pair exists as long as the *key* and the *value* are alive.

:KEY-OR-VALUE
: The pair exists as long as the *key* or the *value* are alive. As long as the *key* is alive, it prevents the *value* from being garbage-collected, and as long as the *value* is alive, it prevents the *key* from being garbage-collected.

In other words, each pair is:

:KEY
: a `EXT:WEAK-MAPPING` from the *key* to the *value*,

:VALUE
: a `EXT:WEAK-MAPPING` from the *value* to the *key*,

:KEY-AND-VALUE
: a `EXT:WEAK-AND-RELATION` of the *key* and the *value*,

:KEY-OR-VALUE

a `EXT:WEAK-OR-RELATION` of the *`key`* and the *`value`*.

Weak Association List API

`(EXT:MAKE-WEAK-ALIST :type :initial-contents)`
: creates a `EXT:WEAK-ALIST`. The *`type`* argument must be one of the four aforementioned types; the default is `:KEY`. The *`initial-contents`* argument must be an association list.

`(EXT:WEAK-ALIST-P object)`
: returns true if the *`object`* is of type `EXT:WEAK-ALIST`.

`(EXT:WEAK-ALIST-TYPE weak-alist)`
: returns the type of the *`weak-alist`*.

`(EXT:WEAK-ALIST-CONTENTS weak-alist)`
: returns an association list that corresponds to the current contents of the *`weak-alist`*.

`(SETF (EXT:WEAK-ALIST-CONTENTS weak-alist) contents)`
: replaces the contents of a *`weak-alist`*. The *`contents`* argument must be an association list.

`(EXT:WEAK-ALIST-ASSOC item weak-alist [:test] [:test-not] [:key])`
: is equivalent to `(ASSOC item (EXT:WEAK-ALIST-CONTENTS weak-alist) [:test] [:test-not] [:key])`.

`(EXT:WEAK-ALIST-RASSOC item weak-alist [:test] [:test-not] [:key])`
: is equivalent to `(RASSOC item (EXT:WEAK-ALIST-CONTENTS weak-alist) [:test] [:test-not] [:key])`.

`(EXT:WEAK-ALIST-VALUE item weak-alist [:test] [:test-not])`
: is equivalent to `(CDR (EXT:WEAK-LIST-ASSOC item weak-alist [:test] [:test-not]))`.

`(SETF (EXT:WEAK-ALIST-VALUE item weak-alist [:test] [:test-not]) value)`
: replaces the value stored for *`item`* in a *`weak-alist`*. When a pair with the given *`item`* as key does not exist or has already been garbage-collected, a new pair is added to the association list.

Weak associations lists are useful to supplement objects with additional information that is stored outside of the object, when the number of such objects is known to be small.

31.7.9. Weak Hash Tables

A weak `HASH-TABLE` is an unordered collection of pairs, each pair being built from an object called *`key`* and an object called *`value`*. There can be only one pair with a given *`key`* in a weak `HASH-TABLE`. The lifetime of each pair depends on the type of the weak `HASH-TABLE`

`:KEY`
: The pair exists as long as the *`key`* is not garbage-collected. As long as the *`key`* is alive, it prevents the *`value`* from being garbage-collected.

`:VALUE`
: The pair exists as long as the *`value`* is not garbage-collected. As long as the *`value`* is alive, it prevents the *`key`* from being garbage-collected.

`:KEY-AND-VALUE`
: The pair exists as long as the *`key`* and the *`value`* are alive.

`:KEY-OR-VALUE`
: The pair exists as long as the *`key`* or the *`value`* are alive. As long as the *`key`* is alive, it prevents the *`key`* from being garbage-collected, and as long as the *`value`* is alive, it prevents the *`key`* from being garbage-collected.

In other words, each pair is:

`:KEY`
: a `EXT:WEAK-MAPPING` from the *`key`* to the *`value`*,

`:VALUE`
: a `EXT:WEAK-MAPPING` from the *`value`* to the *`key`*,

`:KEY-AND-VALUE`
: a `EXT:WEAK-AND-RELATION` of the *`key`* and the *`value`*,

`:KEY-OR-VALUE`
: a `EXT:WEAK-OR-RELATION` of the *`key`* and the *`value`*.

See also Section 18.1.1, “Function `MAKE-HASH-TABLE`”.

Weak `HASH-TABLE`s are useful to supplement objects with additional information that is stored outside of the object. This data structure scales up without performance degradation when the number of pairs is big.

Weak `HASH-TABLE`s are also useful to implement canonicalization tables.

31.8. Finalization

Calling `(EXT:FINALIZE object function)` has the effect that when the specified object is being garbage-collected, `(FUNCALL function object)` will be executed.

Calling `(EXT:FINALIZE object function guardian)` has a similar effect, but only as long as the *`guardian`* has not been garbage-collected: when *`object`* is being garbage-collected, `(FUNCALL function object guardian)` will be executed. If the *`guardian`* is garbage-collected before *`object`* is, nothing happens.

Note

The time when “the *`object`* is being garbage-collected” is not defined deterministically. (Actually, it might possibly never occur.) It denotes a moment at which no references to *`object`* exist from other Lisp objects. When the *`function`* is called, *`object`* (and possibly *`guardian`*) enter the “arena of live Lisp objects” again.

No finalization request will be executed more than once.

31.9. The Prompt

CLISP prompt consists of 3 mandatory parts: “start”, “body”, and “finish”; and 2 optional parts: “break”, which appears only during debugging (after `BREAK` or `ERROR`), and “step”, which appears only during `STEP`ping. Each part is controlled by a custom variable, which can be either a `STRING` or a `FUNCTION` of no arguments returning a `STRING` (if it is something else - or if the return value was not a `STRING` - it is printed with `PRINC`). In the order of invocation:

`CUSTOM:*PROMPT-START*`
: Defaults to an empty string.

`CUSTOM:*PROMPT-STEP*`
: Used only during `STEP`ping. Defaults to “`Step n `”, where *n* is the stepping level as returned by `EXT:STEP-LEVEL`.

CUSTOM:*PROMPT-BREAK*
Used only inside break loop (during debugging). Defaults to "`Break n `", where *n* is the break level as returned by EXT:BREAK-LEVEL.

CUSTOM:*PROMPT-BODY*
Defaults to "`package[n]`" where *package* is the shortest (nick) name (as returned by EXT:PACKAGE-SHORTEST-NAME) of the current package *PACKAGE* if it is **not** the same as it was in the beginning (determined by EXT:PROMPT-NEW-PACKAGE) or if it does not contain symbol T, (it is assumed that in the latter case you would want to keep in mind that your current package is something weird); and *n* is the index of the current prompt, kept in EXT:*COMMAND-INDEX*;

CUSTOM:*PROMPT-FINISH*
Defaults to "> ".

To facilitate your own custom prompt creation, the following functions and variables are available:

EXT:BREAK-LEVEL
This FUNCTION returns current BREAK/ERROR level.

EXT:STEP-LEVEL
This FUNCTION returns current STEP level.

EXT:PROMPT-NEW-PACKAGE
This FUNCTION returns *PACKAGE* or NIL if the current package is the same as it was initially.

EXT:PACKAGE-SHORTEST-NAME
This FUNCTION takes one argument, a PACKAGE, and returns its shortest name or nickname.

EXT:*COMMAND-INDEX*
contains the current prompt number; it is your responsibility to increment it (this variable is bound to 0 before saving the memory image).

31.10. Maximum ANSI CL compliance

Some [ANSI CL standard] features are turned off by default for convenience and for backwards compatibility. They can be switched on, all at once, by setting the SYMBOL-MACRO CUSTOM:*ANSI* to T, or they can be switched on individually. Setting CUSTOM:*ANSI* to T implies the following:

1. Setting CUSTOM:*PRINT-PATHNAMES-ANSI* to T.
2. Setting CUSTOM:*PRINT-SPACE-CHAR-ANSI* to T.
3. Setting CUSTOM:*COERCE-FIXNUM-CHAR-ANSI* to T.
4. Setting CUSTOM:*SEQUENCE-COUNT-ANSI* to T.
5. Setting CUSTOM:*MERGE-PATHNAMES-ANSI* to T.
6. Setting CUSTOM:*PARSE-NAMESTRING-ANSI* to T.
7. Setting CUSTOM:*FLOATING-POINT-CONTAGION-ANSI* to T.
8. Setting CUSTOM:*FLOATING-POINT-RATIONAL-CONTAGION-ANSI* to T.
9. Setting CUSTOM:*PHASE-ANSI* to T.
10. Setting CUSTOM:*LOOP-ANSI* to T.
11. Setting CUSTOM:*PRINT-EMPTY-ARRAYS-ANSI* to T.
12. Setting CUSTOM:*PRINT-UNREADABLE-ANSI* to T.
13. Setting CUSTOM:*DEFUN-ACCEPT-SPECIALIZED-LAMBDA-LIST* to NIL

Note

If you run **CLISP** with the -ansi switch or set the SYMBOL-MACRO CUSTOM:*ANSI* to T and then save memory image, then all subsequent invocations of **CLISP** with this image will be as if with -ansi (regardless whether you actually supply the -ansi switch). You can always set the SYMBOL-MACRO CUSTOM:*ANSI* to NIL, or invoke **CLISP** with the -traditional switch, reversing the above settings, i.e.,

1. Setting CUSTOM:*PRINT-PATHNAMES-ANSI* to NIL.
2. Setting CUSTOM:*PRINT-SPACE-CHAR-ANSI* to NIL.
3. Setting CUSTOM:*COERCE-FIXNUM-CHAR-ANSI* to NIL.
4. Setting CUSTOM:*SEQUENCE-COUNT-ANSI* to NIL.
5. Setting CUSTOM:*MERGE-PATHNAMES-ANSI* to NIL.
6. Setting CUSTOM:*PARSE-NAMESTRING-ANSI* to NIL.
7. Setting CUSTOM:*FLOATING-POINT-CONTAGION-ANSI* to NIL.
8. Setting CUSTOM:*FLOATING-POINT-RATIONAL-CONTAGION-ANSI* to NIL.
9. Setting CUSTOM:*PHASE-ANSI* to NIL.
10. Setting CUSTOM:*LOOP-ANSI* to NIL.
11. Setting CUSTOM:*PRINT-EMPTY-ARRAYS-ANSI* to NIL.
12. Setting CUSTOM:*PRINT-UNREADABLE-ANSI* to NIL.

13. Setting `CUSTOM:*DEFUN-ACCEPT-SPECIALIZED-LAMBDA-LIST*` to `T`

31.11. Additional Fancy Macros and Functions

CLISP comes with some extension macros, mostly defined in the file `macros3.lisp` and loaded from the file `init.lisp` during **make**:

31.11.1. Macro EXT:ETHE

`(EXT:ETHE value-type form)` enforces a type check in both interpreted and compiled code.

31.11.2. Macros EXT:LETF & EXT:LETF*

These macros are similar to `LET` and `LET*`, respectively, except that they can bind places, even places with multiple values. Example:

```
(letf (((values a b) form)) ...)
```

is equivalent to

```
(multiple-value-bind (a b) form ...)
```

while

```
(letf (((first l) 7)) ...)
```

is approximately equivalent to

```
 (LET* ((#:g1 l) (#:g2 (first #:g1)))
   (UNWIND-PROTECT (PROGN (SETF (first #:g1) 7) ...)
      (SETF (first #:g1) #:g2)))
```

31.11.3. Macro EXT:MEMOIZED

(EXT:MEMOIZED *form*) memoizes the primary value of *form* from its first evaluation.

31.11.4. Macro EXT:WITH-COLLECT

Similar to the LOOP's collect construct, except that it is looks more "Lispy" and can appear arbitrarily deep. It defines local macros (with MACROLET) which collect objects given to it into lists, which are then returned as multiple values. E.g.,

```
(ext:with-collect (c0 c1)
  (dotimes (i 10) (if (oddp i) (c0 i) (c1 i))))
⇒ (1 3 5 7 9) ;
⇒ (0 2 4 6 8)
```

returns two LISTs (1 3 5 7 9) and (0 2 4 6 8) as multiple values.

31.11.5. Macro EXT:WITH-GENSYMS

Similar to its namesake from Paul Graham's book “On Lisp”, this macro is useful for writing other macros:

```
(with-gensyms ("FOO-" bar baz zot) ...)
```

expands to

```
(let ((bar (gensym "FOO-BAR-"))
      (baz (gensym "FOO-BAZ-"))
      (zot (gensym "FOO-ZOT-")))
  ...)
```

31.11.6. Function EXT:REMOVE-PLIST

Similar to REMOVE and REMF, this function removes some properties from a property list. It is non-destructive and thus can be used on &REST arguments to remove some keyword parameters, e.g.,

```
(defmacro with-foo ((&KEY foo1 foo2) &BODY body)
  `(... ,foo1 ... ,foo2 ... ,@body))
(defmacro with-foo-bar ((&REST opts &KEY bar1 bar2
                         &ALLOW-OTHER-KEYS)
                        &BODY body)
  `(with-foo (,@(remove-plist opts :bar1 :bar2)
     ... ,bar1 ... ,bar2 ... ,@body)))
(defun foo-bar ()
  (with-foo-bar (:bar1 1 :foo2 2) ...))
```

here `WITH-FOO` does not receive the `:BAR1 1` argument from `FOO-BAR`.

31.11.7. Macros EXT:WITH-HTML-OUTPUT and EXT:WITH-HTTP-OUTPUT

Defined in `inspect.lisp`, these macros are useful for the rudimentary **HTTP** server defined there.

31.11.8. Function EXT:OPEN-HTTP and macro EXT:WITH-HTTP-INPUT

Defined in `clhs.lisp`, they allow downloading data over the Internet using the **HTTP** protocol. `(EXT:OPEN-HTTP url &KEY :IF-DOES-NOT-EXIST)` opens a socket connection to the *url* host, sends the **GET** request, and returns two values: the `SOCKET:SOCKET-STREAM` and content length. `(EXT:WITH-HTTP-INPUT (`*`variable`*` url) &BODY`

`body)` binds *`variable`* to the `SOCKET:SOCKET-STREAM` returned by `EXT:OPEN-HTTP` and executes the *body*. `(EXT:WITH-HTTP-INPUT ((`*`variable contents`*`) url) &BODY body)` additionally binds *`contents`* to the content length.

`EXT:OPEN-HTTP` will check `CUSTOM:*HTTP-PROXY*` on startup and parse the environment variable `HTTP_PROXY` if `CUSTOM:*HTTP-PROXY*` is `NIL`.

31.11.9. Function `EXT:BROWSE-URL`

Function `(EXT:BROWSE-URL url &KEY :BROWSER :OUT)` calls a browser on the URL. *`browser`* (defaults to `CUSTOM:*BROWSER*`) should be a valid keyword in the `CUSTOM:*BROWSERS*` association list. `:OUT` specifies the stream where the progress messages are printed (defaults to `*STANDARD-OUTPUT*`).

31.11.10. Variable `CUSTOM:*HTTP-PROXY*`

If you are behind a proxy server, you will need to set `CUSTOM:*HTTP-PROXY*` to a `LIST` `(name:password host port)`. By default, the environment variable `http_proxy` is used, the expected format is `"name:password@host:port"`. If no `#\@` is present, *`name`* and *`password`* are `NIL`. If no `#\:` is present, *`password`* (or *`port`*) are `NIL`.

Use function `(EXT:HTTP-PROXY &OPTIONAL (STRING (EXT:GETENV "http_proxy")))` to reset `CUSTOM:*HTTP-PROXY*`.

31.12. Customizing CLISP behavior

The user-customizable variables and functions are located in the package **"CUSTOM"** and thus can be listed using `(APROPOS "" "CUSTOM")`:

`CUSTOM:*ANSI*`	`CUSTOM:*APPLYHOOK*`
`CUSTOM:*APROPOS-DO-MORE*`	`CUSTOM:*APROPOS-MATCHER*`
`CUSTOM:*BREAK-ON-WARNINGS*`	`CUSTOM:*BROWSER*`
`CUSTOM:*BROWSERS*`	`CUSTOM:CLHS-ROOT`

CUSTOM:*PRINT-EMPTY-ARRAYS-ANSI*
CUSTOM:*PRINT-INDENT-LISTS*
CUSTOM:*PRINT-SYMBOL-PACKAGE-PREFIX-SHORTEST*
CUSTOM:*PRINT-PATHNAMES-ANSI*
CUSTOM:*PRINT-PRETTY-FILL*
CUSTOM:*PRINT-RPARS*
CUSTOM:*PRINT-SPACE-CHAR-ANSI*
CUSTOM:*PRINT-UNREADABLE-ANSI*
CUSTOM:*PROMPT-BODY*
CUSTOM:*PROMPT-BREAK*
CUSTOM:*PROMPT-FINISH*
CUSTOM:*PROMPT-START*
CUSTOM:*PROMPT-STEP*
CUSTOM:*REPORT-ERROR-PRINT-BACKTRACE*
CUSTOM:*SEQUENCE-COUNT-ANSI*
CUSTOM:*SOURCE-FILE-TYPES*
CUSTOM:*SUPPRESS-CHECK-REDEFINITION*
CUSTOM:*SYSTEM-PACKAGE-LIST*
CUSTOM:*TERMINAL-ENCODING*
CUSTOM:*TRACE-INDENT*
CUSTOM:*USER-COMMANDS*
CUSTOM:*USER-MAIL-ADDRESS*
CUSTOM:*WARN-ON-FLOATING-POINT-CONTAGION*
CUSTOM:*WARN-ON-HASHTABLE-NEEDING-REHASH-AFTER-GC*
CUSTOM:*WITH-HTML-OUTPUT-DOCTYPE*

Note

Some of these variables are platform-specific.

You should set these variables (and do whatever other customization you see fit) in the file `config.lisp` in the build directory before building **CLISP**. Alternatively, after building **CLISP**, or if you are using a binary distribution of **CLISP**, you can modify `config.lisp`, compile and load it, and then save the memory image. Finally, you can create an RC file which is loaded whenever **CLISP** is started.

31.13. Code Walker

You can use function `EXT:EXPAND-FORM` to expand all the macros, `SYMBOL-MACRO`s, etc, in a single form:

```
(EXT:EXPAND-FORM '(macrolet ((bar (x) `(print ,x)))
                    (macrolet ((baz (x) `(bar ,x)))
                      (symbol-macrolet ((z 3))
                        (baz z)))))
⇒ (locally (print 3)) ; the expansion
⇒ T                   ; indicator: some expansion has act
```

This is sometimes called a "code walker", except that a code walker would probably leave the MACROLET and SYMBOL-MACROLET forms intact and just do the expansion.

Warning

Function EXT:EXPAND-FORM expands forms by assuming the EVAL-WHEN situation :EXECUTE and is therefore unsuitable for forms that may later be passed to the compiler:

```
(EXT:EXPAND-FORM '(EVAL-WHEN (:COMPILE-TOPLEVEL) (foo
⇒ NIL ;
⇒ T
(EXT:EXPAND-FORM '(EVAL-WHEN (:LOAD-TOPLEVEL) (foo)))
⇒ NIL ;
⇒ T
```

Chapter 32. Platform Specific Extensions

Table of Contents

32.1. Random Screen Access

Platform Dependent: UNIX, Win32 platforms only.

`(SCREEN:MAKE-WINDOW)`
: returns a WINDOW-STREAM. As long as this stream is open, the terminal is in cbreak/noecho mode. `*TERMINAL-IO*` should not be used for input or output during this time. (Use `EXT:WITH-KEYBOARD` and `EXT:*KEYBOARD-INPUT*` instead.)

`(SCREEN:WITH-WINDOW . body)`
: binds ***`SCREEN:*WINDOW*`*** to a WINDOW-STREAM and executes *`body`*. The stream is guaranteed to be closed when the body is left. During its execution, `*TERMINAL-IO*` should not be used, as above.

`(SCREEN:WINDOW-SIZE window-stream)`
: returns the window's size, as two values: height (= y_{max}+1) and width (= x_{max}+1).

`(SCREEN:WINDOW-CURSOR-POSITION window-stream)`

returns the position of the cursor in the window, as two values: line (≥ 0, $\leq y_{max}$, 0 means top), column (≥ 0, $\leq x_{max}$, 0 means left margin).

(SCREEN:SET-WINDOW-CURSOR-POSITION *window-stream line column*)

sets the position of the cursor in the window.

(SCREEN:CLEAR-WINDOW *window-stream*)

clears the window's contents and puts the cursor in the upper left corner.

(SCREEN:CLEAR-WINDOW-TO-EOT *window-stream*)

clears the window's contents from the cursor position to the end of window.

(SCREEN:CLEAR-WINDOW-TO-EOL *window-stream*)

clears the window's contents from the cursor position to the end of line.

(SCREEN:DELETE-WINDOW-LINE *window-stream*)

removes the cursor's line, moves the lines below it up by one line and clears the window's last line.

(SCREEN:INSERT-WINDOW-LINE *window-stream*)

inserts a line at the cursor's line, moving the lines below it down by one line.

(SCREEN:HIGHLIGHT-ON *window-stream*)

switches highlighted output on.

(SCREEN:HIGHLIGHT-OFF *window-stream*)

switches highlighted output off.

(SCREEN:WINDOW-CURSOR-ON *window-stream*)

makes the cursor visible, a cursor block in most implementations.

(SCREEN:WINDOW-CURSOR-OFF *window-stream*)

makes the cursor invisible, in implementations where this is possible.

32.2. External Modules

Platform Dependent: UNIX, Win32 platforms only.

Modules on Win32

Everything described in the section will work verbatim on **Win32** when using **Cygwin** or **MinGW**, *except* for one thing - you will need to replace the `run` extension in `lisp.run` with the **Win32** executable extension `exe`.

For historical reasons, all examples appear to assume **UNIX** and use the `run` file type (“extension”) for the **CLISP** run-time. This does **not** mean that they will not work on **Win32**.

32.2.1. Overview

CLISP has a facility for adding external modules (written in **C**, for example). It is invoked through **clisp-link**.

A ***module*** is a piece of external code which defines extra Lisp objects, symbols and functions. A module ***name*** must consist of the characters `A-Z`, `a-z`, `_`, `0-9`. The module name “clisp” is reserved. Normally a module name is derived from the corresponding file name.

clisp-link needs a directory containing:

- `"modules.d"`
- `"modules.c"`
- `"clisp.h"`

clisp-link expects to find these files in a subdirectory `linkkit/` of the current directory. This can be overridden by the environment variable `CLISP_LINKKIT`.

clisp-link operates on **CLISP** linking sets and on module sets.

linking set

A ***linking set*** is a directory containing:

`makevars`
: some **/bin/sh** commands, setting the variables

`CC`	the C compiler
`CPPFLAGS`	flags for the C compiler, when preprocessing or compiling
`CFLAGS`	flags for the C compiler, when compiling or linking
`CLFLAGS`	flags for the C compiler, when linking
`LIBS`	libraries to use when linking (either present in the linking set directory, or system-wide)
`X_LIBS`	additional X Window System libraries to use
`RANLIB`	the ranlib command
`FILES`	the list of files needed when linking

`modules.h`
: the list of modules contained in this linking set

`modules.o`
: the compiled list of modules contained in this linking set

all the `FILES`
: listed in `makevars`

`lisp.run`
: the executable

`lispinit.mem`
: the memory image

To run a **CLISP** contained in some linking set *`directory`*, call

```
$ directory/lisp.run -M directory/lispinit.mem
```

or

```
$ clisp -K directory
```

(recommended, since it also passes `-B` to the run-time).

module set

A ***module set*** is a directory containing:

link.sh
: some **/bin/sh** commands, which prepare the directory before linking, and set the variables `NEW_FILES`, `NEW_LIBS`, `NEW_MODULES`, `TO_LOAD` and optionally `TO_PRELOAD`

and any other files
: needed by **link.sh**

In **link.sh** the module set directory is referred to as `$modulename/`.

Module set variables

The following variables should be defined in **link.sh**.

`NEW_FILES`
: the space-separated list of files that belong to the module set and will belong to every new linking set.

`NEW_LIBS`
: the space-separated list of files or **C** compiler switches that need to be passed to the **C** compiler when linking the `lisp.run` belonging to a new linking set.

`NEW_MODULES`
: the space-separated list of the module names belonging to the module set. Normally, every `#P".c"` file in the module set defines a module of its own. The module name is derived from the file name.

`TO_LOAD`

the space-separated list of Lisp files to load before building the `lispinit.mem` belonging to a new linking set.

`TO_PRELOAD` (optional)

the space-separated list of Lisp files to load into an intermediate `lispinit.mem` file, before building the `lispinit.mem` belonging to a new linking set. This variable is usually used to create (or unlock) the Lisp `PACKAGE`s which must be present when the new `#P".c"` files are initialized. E.g., the `FFI:DEF-CALL-IN` functions must reside in already defined packages; see Example 32.6, "Calling Lisp from C". You can find a live example in `modules/syscalls/preload.lisp` and `modules/syscalls/link.sh.in`.

Warning

If you are unlocking a package, you must also `DELETE` it from `CUSTOM:*SYSTEM-PACKAGE-LIST*` (see Section 31.2, "Saving an Image") here and re-add it to `CUSTOM:*SYSTEM-PACKAGE-LIST*` in one of the `TO_LOAD` files. See, e.g., `modules/i18n/preload.lisp` and `modules/i18n/link.sh.in`.

Creating linking sets

The command

```
$ clisp-link create-module-set module file1.c ...
```

creates a module set in *`module`* directory which refers (via symbolic links) to *`file1.c`* etc. The files are expected to be modules of their own.

The command

```
$ clisp-link add-module-set module source destination
```

combines a linking set in directory *`source`* and a module in directory *`module`* to a new linking set, in the directory *`destination`* which is newly created.

The command

```
$ clisp-link run source module ...
```

runs the linking set in directory *source*, with the module in directory *module* loaded. More than one module can be specified. If **CLISP** has been built with the configuration option `--with-dynamic-modules`, the loading will be performed dynamically. Otherwise - this is much slower - a temporary linking set will be created and deleted afterwards.

32.2.2. Module initialization

Each module has two initialization functions:

void `module__`*name*`__init_function_1` (struct module_t* module)

called only *once* when **CLISP** discovers while loading a memory image that there is a module present in the executable (`lisp.run`) which was not present at the time the image was saved. It can be used to create Lisp objects, e.g. functions or keywords, and is indeed used for that purpose by **modprep**.
You do **not** have to define this function yourself; **modprep** and **"FFI"** will do that for you.
If you use **"FFI"**, `(FFI:C-LINES :init-once ...)` will add code to this function.

Warning

The `PACKAGE`s must already exist and be unlocked, cf. `TO_PRELOAD`.

Warning

If you are using **modprep** **and** defining your own "init-once" function, it must call the `module__`*name*`__init_function_1__modprep` function!

void `module__`*name*`__init_function_2` (struct module_t* module)

called *every time* **CLISP** starts. It can be used to bind names to foreign addresses, since the address will be different in each invocation of **CLISP**, and is indeed used for that purpose by **“FFI”** (e.g., by `FFI:DEF-CALL-OUT`). It can also be used to set parameters of the libraries to which the module interfaces, e.g., the **`pcre`** module sets `pcre_malloc` and `pcre_free`.
You do **not** have to define this function yourself; **modprep** and **“FFI”** will do that for you.
If you use **“FFI”**, `(FFI:C-LINES :init-always ...)` will add code to this function.

`name` is the module name.

See also Section 31.1, “Customizing **CLISP** Process Initialization and Termination”.

32.2.3. Module finalization

Each module has a finalization function

void `module__`*`name`*`__fini_function` (struct module_t* module)
called before exiting **CLISP**.
You do **not** have to define this function yourself; **modprep** and **“FFI”** will do that for you.
If you use **“FFI”**, `(FFI:C-LINES :fini ...)` will add code to this function.

`name` is the module name.

See also Section 31.1, “Customizing **CLISP** Process Initialization and Termination”.

32.2.4. Function `EXT:MODULE-INFO`

Function `(EXT:MODULE-INFO &OPTIONAL` *`name verbose`*`)` allows one to inquire about what modules are available in the currently running image. When called without arguments, it returns the list of module names, starting with “clisp”. When *`name`* is supplied and names a module, 3 values are returned - *`name`*, *`subr-count`*, *`object-count`*. When *verbose* is non-NIL, the full list of module lisp function names written in **C** (Subrs) and the full list of internal lisp objects available in **C** code are additionally returned for the total of 5 values.

When *name* is :FFI, returns the list of shared libraries opened using :LIBRARY. When *verbose* is non-NIL, return the association list of DLL names and all foreign objects associated with it.

32.2.5. Function SYS::DYNLOAD-MODULES

Platform Dependent: Only when compiled with configure flag --with-dynamic-modules.

Note

Dynamic loading does not work on all operating systems (dlopen or equivalent is required).

Note

--with-dynamic-modules precludes some optimizations which are enabled by default.

Function (SYS::DYNLOAD-MODULES *filename* ({*name*}+)) loads a shared object file or library containing a number of named external **CLISP** modules.

Note

This facility *cannot* be used to access arbitrary shared libraries. To do that, use the :LIBRARY argument to FFI:DEF-CALL-OUT and FFI:DEF-C-VAR instead.

External modules for **CLISP** are shared objects (dynamic libraries) that contain the module__*name*__subr_tab variable, among others. This serves to register external functions which operate on Lisp-level structures with **CLISP**.

To use `dlopen` with modules, you should add `-fPIC` to the module's compilation options. Something like **cc -shared -o *name*.so *name*.o** may be needed to produce the shared object file.

32.2.6. Example

To link in the **"FFI"** bindings for the GNU/**Linux** operating system, the following steps are needed. (Step 1 and step 2 need not be executed in this order.)

1. **Create a new module set**

   ```
   $ clisp-link create-module-set linux /somewhere/bindir
   ```

2. **Modify the newly created `linux/link.sh`**

   a. **add `-lm` to the libraries**

      replace

      ```
      NEW_LIBS="$file_list"
      ```

      with

      ```
      NEW_LIBS="$file_list -lm"
      ```

   b. **load `linux.fas` before saving the memory image**

      replace

      ```
      TO_LOAD=''
      ```

      with

      ```
      TO_LOAD='/somewhere/bindings/linux.fas'
      ```

3. **Compile `linux.lisp`, creating `linux.c`**

   ```
   $ clisp -c /somewhere/bindings/linux.lisp
   ```

4. **Create a new linking set**

   ```
   $ clisp-link add-module-set linux base base+linux
   ```

5. **Run and try it**

   ```
   $ base+linux/lisp.run -M base+linux/lispinit.mem -x '(
   ```

32.2.7. Module tools

There are some tools to facilitate easy module writing.

32.2.7.1. Modprep

If your module is written in C, you can pre-process your sources with **modprep** in the **CLISP** distribution and define lisp functions with the `DEFUN` macro:

```
DEFUN(MY-PACKAGE:MY-FUNCTION-NAME, arg1 arg2 &KEY FOO BAR
  if (!boundp(STACK_0)) STACK_0 = fixnum(0); /* BAR */
  if (!boundp(STACK_1)) STACK_1 = fixnum(1); /* FOO */
  pushSTACK(`MY-PACKAGE::SOME-SYMBOL`); /* create a symbo
  pushSTACK(`#(:THIS :IS :A :VECTOR)`); /* some vector, c
  pushSTACK(``MY-PACKAGE::MY-FUNCTION-NAME``); /* double
  VALUES1(listof(7)); /* cons up a new list and clean up
}
```

Then `(MY-PACKAGE:MY-FUNCTION-NAME 'A 12 :FOO T)` will return (A 12 T 0 MY-PACKAGE::SOME-SYMBOL #(:THIS :IS :A :VECTOR) #<ADD-ON-SYSTEM-FUNCTION MY-PACKAGE:MY-FUNCTION-NAME>) (assuming you `EXPORT`ed `MY-FUNCTION-NAME` from **"MY-PACKAGE"**).

Another useful macros are:

DEFVAR

create a GC-visible private object

DEFFLAGSET

define a **C** function which will remove several flag arguments from the `STACK` and return the combined flag value

DEFCHECKER

define a map from **cpp** constants to lisp symbols and functions that map between them, checking that the argument is appropriate

See `modules/syscalls/calls.c` and other included modules for more examples and file **modprep** for full documentation.

Warning

If you manipulate Lisp objects, you need to watch out for GC-safety.

32.2.7.2. clisp.h

If your module is written in **C**, you will probably want to `#include "clisp.h"` to access **CLISP** objects. You will certainly need to read `"clisp.h"` and some code in included modules, but here are some important hints that you will need to keep in mind:

- Lisp objects have type object.
- Variables of this type are invalidated by lisp memory allocation (`allocate_*()` functions) - but **not** **C** allocations (`malloc` et al) - and must be saved on the `STACK` using **cpp** macros `pushSTACK()`, `popSTACK()` and `skipSTACK()`.
- Access object slots using the appropriate `TheFoo()` macro, e.g., `TheCons(my_cons)->Car`, but first check the type with `consp()`.
- Arguments are passed on the `STACK`, as illustrated in the above example.
- Wrap your system calls in `begin_system_call()`/`end_system_call()` pairs. These macros, defined in `"clisp.h"`, save and restore registers used by **CLISP** which could be clobbered by a system call.

32.2.7.3. Exporting

If your module uses **“FFI”** to interface to a **C** library, you might want to make your module package case-sensitive and use `exporting.lisp` in the **CLISP** distribution to make **“FFI”** forms and `DEFUN`, `DEFMACRO` at al export the symbols they define. See `modules/netica/`, `modules/matlab/` and `modules/bindings/` for examples.

32.2.8. Trade-offs: “FFI” vs. C modules

When deciding how to write a module: whether to use **“FFI”** or to stick with **C** and **modprep**, one has to take into account several issues:

Speed: C wins

“FFI” has a noticeable overhead: compare `RAWSOCK:HTONS` (defined in `modules/rawsock/rawsock.c`) with

```
(FFI:DEF-CALL-OUT htons (:name "htons") (:library :def
  (:arguments (s ffi:short)) (:return-type ffi:short)
```

and observe that `RAWSOCK:HTONS` is almost 3 times as fast (this really does compare the **“FFI”** overhead to the normal lisp function call because `htons` is computationally trivial). This difference will matter only if you call a simple function very many times, in which case it would make sense to put the loop itself into **C**.

Portability: C wins

First of all, **“FFI”** is **not** as widely ported as **CLISP**, so it is possible that you will face a platform where **CLISP** runs but **“FFI”** is not present.
Second, it is much easier to handle portability in **C**: observe the alternative implementations of `htonl` et al in `modules/rawsock/rawsock.c`.
Third, certain **C** structures have different layout on different platforms, and functions may take 64-bit arguments on some platforms and 32-bit arguments on others; so the **“FFI”** code has to track those differences, while **C** will mostly take care of these things for you.

Code size: “FFI” wins

You need to type much fewer characters with **“FFI”**, and, if you use the `:LIBRARY` argument to `FFI:DEF-CALL-OUT` and `FFI:DEF-C-VAR`, you do not need to leave your **CLISP** session to try out your code. This is a huge advantage for rapid prototyping.

UI: C wins

To produce a nice lispy UI (using `&OPTIONAL` and `&KEY`word arguments etc), you will need to write wrappers to your `FFI:FOREIGN-FUNCTION`s, while in **C** you can do that directly. The same goes for “polymorphism”: accepting different argument types (like, e.g., `POSIX:RESOLVE-HOST-IPADDR` does) would require a lisp wrapper for `FFI:FOREIGN-FUNCTION`s.

Learning curve: unclear

If you are comfortable with **C**, you might find the **CLISP** **C** module facilities (e.g., **modprep**) very easy to use.
CLISP **“FFI”**, on the other hand, is quite high-level, so, if you are more comfortable with high-level languages, you might find it easier to write **“FFI”** forms than **C** code.

Safety: unclear

One can get a segfault either way: if your `FFI:DEF-CALL-OUT` form does not describe the function's expectations with respect to the arguments and return values (including *`ALLOCATION`*), you will probably learn that the hard way. If the module is written in **C**, all the opportunities to shoot oneself in the foot (and other body parts) are wide open (although well known to most **C** users). However, with **C**, one has to watch for GC-safety too.

Note

The granularity of the choice is *per function*: the same module can use both **modprep** and **“FFI”**.

Note

It is not a good idea to have both `foo.lisp` and `foo.c` files in a module, because if you ever add an **“FFI”** form to the former, `COMPILE-FILE` will overwrite the latter.

32.2.9. Modules included in the source distribution

A few modules come with the *source* distribution of **CLISP** (but are not necessarily built in a particular *binary* distribution).

To use modules, read `unix/INSTALL` and build **CLISP** in directory `build-dir` with, e.g.,

```
$ ./configure --with-module=pcre --with-module=clx/new-cl:
```

then run it with

```
$ ./build-dir/clisp -K full
```

This will create a **base** linking set with modules **`i18n`**, **`regexp`** and **`syscalls`** (and maybe **`readline`**); and a **full** linking set with modules **`clx/new-clx`** and **`pcre`** in addition to the 3 (or 4) base modules.

Here we list the included modules by their general theme. See Chapter 33, *Extensions Implemented as Modules* for individual module documentation.

32.2.9.1. Base Modules

The default build process includes the following modules in *both* **base** **and** **full** linking sets:

`i18n`
: Internationalization of User Programs.

`regexp`
: The POSIX Regular Expressions matching, compiling, executing.

`syscalls`
: Use some system calls in a platform-independent way.

`readline` (only when both GNU readline and "FFI" are available)
: When GNU readline and **"FFI"** are available, some advanced readline and history features are exported using this module.

The composition of the **full** linking set depends on the platform and on the vendor preferences.

32.2.9.2. Database, Directory et al

`gdbm`
: Interface to GNU DataBase Manager by Masayuki Onjo.

`berkeley-db`
: Berkeley DB from Sleepycat Software interface.

`dirkey`
: Directory Access (LDAP, **Win32** registry etc).

`postgresql`
: Access PostgreSQL from **CLISP**.

`oracle`
: Access Oracle from **CLISP**; by John Hinsdale.

32.2.9.3. Mathematics, Data Mining et al

`libsvm`
: Build Support Vector Machine models using LibSVM inside **CLISP**.

`pari`
: Interface to the computer algebra system PARI.

`matlab`
: Do matrix computations via MATLAB.

`netica`
: Work with Bayesian belief networks and influence diagrams using Netica C API.

32.2.9.4. Matching, File Processing et al

`pcre`
: The Perl Compatible Regular Expressions matching, compiling, executing.

`wildcard`
: Shell (**/bin/sh**) globbing (Pathname Matching).

`zlib`
: Compress `VECTOR`s using ZLIB.

32.2.9.5. Networking

`rawsock`
: Raw socket access.

`fastcgi`
: Access FastCGI from **CLISP**; by John Hinsdale.

32.2.9.6. Graphics

CLX
: Call Xlib functions from **CLISP**. Two implementations are supplied:

  `clx/mit-clx`, from MIT
  ftp://ftp.x.org/R5contrib/CLX.R5.02.tar.Z
  : the standard implementation

  `clx/new-clx`, by Gilbert Baumann
  : faster, with additional features, but not quite complete yet. Please try it first and use **`clx/mit-clx`** only if **`clx/new-clx`** does not work for you. **`clx/new-clx`** comes with several demos, please try them using

```
$ clisp -K full -i modules/clx/new-clx/demos/clx-d
```

  and follow the intructions.

This functionality is documented in the manual http://www.stud.uni-karlsruhe.de/~unk6/clxman/, also available in the **CLISP** source distribution as `modules/clx/clx-manual.tar.gz`.

`gtk2`
Use GTK+ and Glade to create GUI by James Bailey.

32.2.9.7. Bindings

Call the operating system functions from **CLISP**. The following platforms are supported:

`bindings/glibc`
Linux/GNU libc
`bindings/win32`
Win32

32.2.9.8. Toys and Games

`queens`
Compute the number of solutions to the *n*-queens problem on a *n*×*n* chessboard (a toy example for the users to explore the **CLISP** module system).
`modules/clx/new-clx/demos/sokoban.lisp`
a demo which comes with **`clx/new-clx`**.

32.3. The Foreign Function Call Facility

Platform Dependent: Many UNIX, Win32 platforms only.

32.3.1. Introduction

This facility, also known as “Foreign Language Interface”, allows one to call a function implemented in **C** from inside **CLISP** and to do many related things, like inspect and modify foreign memory, define a “callback” (i.e., make a lisp function available to the **C** world), etc. To use this facility, one writes a foreign function description into an ordinary Lisp file, which is then compiled and loaded as usual.

There are two basic ways to do define a foreign function:

1. Use `dlopen` and `dlsym` to get to the location of the function code in a dynamic library. To access this facility, pass the `:LIBRARY` option to `FFI:DEF-CALL-OUT` and `FFI:DEF-C-VAR`. Unfortunately, this functionality is not available on some operating systems, and, also, it offers only a part of the foreign functionality: **cpp** macros and `inline` functions cannot be accessed this way.
2. Use a somewhat less direct way: when you do not use the `:LIBRARY` argument, `COMPILE-FILE` produces a `#P".c"` file (in addition to a `#P".fas"` and a `#P".lib"`). Then you compile (with a **C** compiler) and link it into **CLISP** (statically, linking it into `lisp.a`, or dynamically, loading it into a running **CLISP** using `dlopen` and `dlsym`). This way you can use any functionality your foreign library exports, whether using ordinary functions, `inline` functions, or **cpp** macros (see Example 32.5, “Accessing **cpp** macros”).

All symbols relating to the foreign function interface are exported from the package **“FFI”**. To use them, `(USE-PACKAGE "FFI")`.

Special **“FFI”** forms may appear anywhere in the Lisp file.

32.3.2. Overview

These are the special **"FFI"** forms. We have taken a pragmatic approach: the only foreign languages we support for now are **C** and ANSI **C**.

Note

Unless specifically noted otherwise, type specification parameters are not evaluated, so that they can be compiled by `FFI:PARSE-C-TYPE` into the internal format at macroexpansion time.

High-level "FFI" forms; *`name`* is any Lisp `SYMBOL`; *`c-name`* is a `STRING`

`(FFI:DEF-C-TYPE name &OPTIONAL c-type)`
: This form makes *`name`* a shortcut for *`c-type`*. Note that *`c-type`* may already refer to *`name`*. Forward declarations of types are not possible, however.
When *`c-type`* is omitted, the type is assumed to be an integer, and its size and signedness are determined at link time, e.g., `(FFI:DEF-C-TYPE size_t)`.

`(FFI:DEF-C-VAR name {option}*)`
: This form defines a `FFI:FOREIGN-VARIABLE`. *`name`* is the Lisp name, a regular Lisp `SYMBOL`.
Options for `FFI:DEF-C-VAR`

`(:NAME c-name)`
: specifies the name as seen from **C**, as a `STRING`. If not specified, it is derived from the print name of the Lisp name.

`(:TYPE c-type)`
: specifies the variable's foreign type.

`(:READ-ONLY BOOLEAN)`
: If this option is specified and non-`NIL`, it will be impossible to change the variable's value from within Lisp (using `SETQ` or similar).

`(:ALLOC ALLOCATION)`
: This option can be either `:NONE` or `:MALLOC-FREE` and defaults to `:NONE`. If it is `:MALLOC-FREE`, any values of type FFI:C-

STRING, FFI:C-PTR, FFI:C-PTR-NULL, FFI:C-ARRAY-PTR within the foreign value are assumed to be pointers to `malloc`-allocated storage, and when `SETQ` replaces an old value by a new one, the old storage is freed using `free` and the new storage allocated using `malloc`. If it is `:NONE`, `SETQ` assumes that the pointers point to good storage (not `NULL`!) and overwrites the old values by the new ones. This is dangerous (just think of overwriting a string with a longer one or storing some data in a `NULL` pointer...) and deprecated.

`(:LIBRARY name)`
: Specifies the (optional) dynamic library which contains the variable, the default is set by `FFI:DEFAULT-FOREIGN-LIBRARY`.

`(:DOCUMENTATION string)`
: Specifies the (optional) `VARIABLE` documentation.

`(FFI:DEF-C-CONST name {option}*)`

This form defines a Lisp constant variable *name* whose value is determined at link time using an internal `FFI:FOREIGN-FUNCTION`. When the **cpp** constant is not defined, *name* is unbound.

Options for `FFI:DEF-C-CONST`

`(:NAME c-name)`
: specifies the name as seen from **C**, as a `STRING`. If not specified, it is derived from the print name of the Lisp name.

`(:TYPE c-type)`
: specifies the constant's foreign type, one of

FFI:INT

FFI:C-STRING

FFI:C-POINTER

`(:DOCUMENTATION string)`
: Specifies the (optional) `VARIABLE` documentation.

See also Example 32.5, “Accessing **cpp** macros”.

`(FFI:DEF-CALL-OUT name {option}*)`

This form defines a named call-out function (a foreign function called from Lisp: control flow temporarily leaves Lisp).

Options for `FFI:DEF-CALL-OUT`

`(:NAME c-name)`

Any Lisp function call to `#'name` is redirected to call the C function `c-name`.

`(:ARGUMENTS {(argument c-type [PARAM-MODE [ALLOCATION]])}*)`
`(:RETURN-TYPE c-type [ALLOCATION])`
: Argument list and return value, see Section 32.3.8, “Argument and result passing conventions” and Section 32.3.9, “Parameter Mode”.

`(:LANGUAGE language)`
: See Section 32.3.4, “The choice of the **C** flavor”.

`(:BUILT-IN BOOLEAN)`
: When the function is a C built-in, the full prototype will be output (unless suppressed by `FFI:*OUTPUT-C-FUNCTIONS*`).

`(:LIBRARY name)`
: Specifies the (optional) dynamic library which contains the function, the default is set by `FFI:DEFAULT-FOREIGN-LIBRARY`

`(:DOCUMENTATION string)`
: Specifies the (optional) `FUNCTION` documentation.

`(FFI:DEF-CALL-IN name {option}*)`

This form defines a named call-in function (i.e., a Lisp function called from the foreign language: control flow temporary enters Lisp)

Options for `FFI:DEF-CALL-IN`

`(:NAME c-name)`
: Any C function call to the C function `c-name` is redirected to call the **Common Lisp** function `#'name`.

`(:ARGUMENTS {(argument c-type [PARAM-MODE [ALLOCATION]])}*)`
`(:RETURN-TYPE c-type [ALLOCATION])`
: Argument list and return value, see Section 32.3.8, “Argument and result passing conventions” and Section 32.3.9, “Parameter Mode”.

`(:LANGUAGE language)`
: See Section 32.3.4, “The choice of the **C** flavor”.

`(FFI:CLOSE-FOREIGN-LIBRARY name)`

Close (unload) a shared foreign library (opened by the `:LIBRARY` argument to `FFI:DEF-CALL-OUT` or `FFI:DEF-C-VAR`).

If you modify your shared library, you need to use close it using `FFI:CLOSE-FOREIGN-LIBRARY` first. When you try to use the `FFI:FOREIGN-VARIABLE` or the `FFI:FOREIGN-FUNCTION` which resides in the library *name*, it will be re-opened automatically.

(`FFI:DEFAULT-FOREIGN-LIBRARY` *library-name*)

This macro sets the default `:LIBRARY` argument for `FFI:DEF-CALL-OUT` and `FFI:DEF-C-VAR`. *library-name* should be `NIL` (meaning use the **C** file produced by `COMPILE-FILE`), a `STRING`, or, depending on the underlying `dlsym` implementation, `:DEFAULT` or `:NEXT`.

The default is set separately in each compilation unit, so, if you are interfacing to a single library, you can set this variable in the beginning of your lisp file and omit the `:LIBRARY` argument throughout the file.

(`FFI:DEF-C-STRUCT` *name* (*symbol* *c-type*)*)

This form defines *name* to be both a `STRUCTURE-CLASS` and a foreign **C** type with the given slots. If this class representation overhead is not needed one should consider writing `(FFI:DEF-C-TYPE name (FFI:C-STRUCT {LIST | VECTOR} (symbol c-type)*))` instead. *name* is a `SYMBOL` (structure name) or a `LIST` whose `FIRST` element is the structure name and the `REST` is options. Two options are supported at this time:

Options for `FFI:DEF-C-STRUCT`

`:TYPEDEF`

means that the name of this structure is a **C** type defined with `typedef` elsewhere.

`:EXTERNAL`

means that this structure is defined in a `#P".c"` file that you include with, e.g., `(FFI:C-LINES "#include <filename.h>~%")`.

These options determine how the struct is written to the `#P".c"`.

(`FFI:DEF-C-ENUM` *name* {*symbol* | (*symbol* [*value*])}*)

This form defines *symbol*s as constants, similarly to the **C** declaration enum { *symbol* [= *value*], ... };

You can use `(FFI:ENUM-FROM-VALUE name value)` and `(FFI:ENUM-TO-VALUE name symbol)` to convert between the numeric and symbolic representations (of course, the latter function boils down to `SYMBOL-VALUE` plus a check that the *symbol* is indeed a constant defined in the `FFI:DEF-C-ENUM` *name*).

(FFI:C-LINES *format-string* {*argument*}*)

This form outputs the string (FORMAT NIL *format-string* {*argument*}*) to the C output file's top level. This is usually used to include the relevant header files, see :EXTERNAL and FFI:*OUTPUT-C-FUNCTIONS*.

When *format-string* is not a STRING, is should be a SYMBOL, and then the STRING (FORMAT NIL {*argument*}*) is added to the appropriate C function:

:INIT-ALWAYS
:INIT-ONCE
: initialization function

:FINI
: finalization function

(FFI:ELEMENT *c-place* $index_1$ ... $index_n$)

Array element: If *c-place* is of foreign type (FFI:C-ARRAY *c-type* (dim_1 ... dim_n)) and $0 \leq index_1 < dim_1$, ..., $0 \leq index_n < dim_n$, this will be the place corresponding to (AREF *c-place* $index_1$ ... $index_n$) or *c-place*[$index_1$]...[$index_n$]. It is a place of type *c-type*. If *c-place* is of foreign type (FFI:C-ARRAY-MAX *c-type* *dim*) and $0 \leq index < dim$, this will be the place corresponding to (AREF *c-place* *index*) or *c-place*[*index*]. It is a place of type *c-type*.

(FFI:DEREF *c-place*)

Dereference pointer: If *c-place* is of foreign type (FFI:C-PTR *c-type*), (FFI:C-PTR-NULL *c-type*) or (FFI:C-POINTER *c-type*), this will be the place the pointer points to. It is a place of type *c-type*. For (FFI:C-PTR-NULL *c-type*), the *c-place* may not be NULL.

(FFI:SLOT *c-place* *slot-name*)

Struct or union component: If *c-place* is of foreign type (FFI:C-STRUCT *class* ... (*slot-name* *c-type*) ...) or of type (FFI:C-UNION ... (*slot-name* *c-type*) ...), this will be of type *c-type*.

(FFI:CAST *c-place* *c-type*)

Type change: A place denoting the same memory locations as the original *c-place*, but of type *c-type*.

(FFI:OFFSET *c-place* *offset* *c-type*)

Type change and displacement: return a place denoting a memory locations displaced from the original *c-place* by an *offset* counted in bytes, with type *c-type*. This can be used to resize an array, e.g. of *c-type* (FFI:C-ARRAY uint16 *n*) via (FFI:OFFSET *c-place* 0 '(FFI:C-ARRAY uint16 *k*)).

(FFI:C-VAR-ADDRESS *c-place*)
: Return the address of *c-place* as a Lisp object of type FFI:FOREIGN-ADDRESS. This is useful as an argument to foreign functions expecting a parameter of **C** type FFI:C-POINTER.

(FFI:C-VAR-OBJECT *c-place*)
: Return the FFI:FOREIGN-VARIABLE object underlying the *c-place*. This is also an acceptable argument type to a FFI:C-POINTER declaration.

(FFI:TYPEOF *c-place*)
: returns the *c-type* corresponding to the *c-place*.

(FFI:SIZEOF *c-type*)
(FFI:SIZEOF *c-place*)
: The first form returns the size and alignment of the **C** type *c-type*, measured in bytes.
The second form returns the size and alignment of the **C** type of *c-place*, measured in bytes.

(FFI:BITSIZEOF *c-type*)
(FFI:BITSIZEOF *c-place*)
: The first form returns the size and alignment of the **C** type *c-type*, measured in bits.
The second form returns the size and alignment of the **C** type of *c-place*, measured in bits.

(FFI:FOREIGN-ADDRESS-NULL *foreign-entity*)
: This predicate returns T if the *foreign-entity* refers to the NULL address (and thus *foreign-entity* should probably not be passed to most foreign functions).

(FFI:FOREIGN-ADDRESS-UNSIGNED *foreign-entity*)
(FFI:UNSIGNED-FOREIGN-ADDRESS *number*)
: FFI:FOREIGN-ADDRESS-UNSIGNED returns the INTEGER address embodied in the Lisp object of type FFI:FOREIGN-ADDRESS, FFI:FOREIGN-POINTER, FFI:FOREIGN-VARIABLE or FFI:FOREIGN-FUNCTION.
FFI:UNSIGNED-FOREIGN-ADDRESS returns a FFI:FOREIGN-ADDRESS object pointing to the given INTEGER address.

(FFI:FOREIGN-ADDRESS *foreign-entity*)
: FFI:FOREIGN-ADDRESS is both a type name and a selector/constructor function. It is the Lisp object type corresponding to a FFI:C-POINTER external type declaration, e.g. a call-out

function with `(:RETURN-TYPE FFI:C-POINTER)` yields a Lisp object of type `FFI:FOREIGN-ADDRESS`.
The function extracts the object of type `FFI:FOREIGN-ADDRESS` living within any `FFI:FOREIGN-VARIABLE` or `FFI:FOREIGN-FUNCTION` object. If the *`foreign-entity`* already is a `FFI:FOREIGN-ADDRESS`, it returns it. If it is a `FFI:FOREIGN-POINTER` (e.g. a base foreign library address), it encapsulates it into a `FFI:FOREIGN-ADDRESS` object, as suitable for use with a FFI:C-POINTER external type declaration. It does not construct addresses out of `NUMBER`s, `FFI:UNSIGNED-FOREIGN-ADDRESS` must be used for that purpose.

`(FFI:FOREIGN-VARIABLE` *`foreign-entity c-type-internal`* `&KEY` *`name`*`)`

This constructor creates a new `FFI:FOREIGN-VARIABLE` from the given `FFI:FOREIGN-ADDRESS` or `FFI:FOREIGN-VARIABLE` and the internal **C** type descriptor (as obtained from `FFI:PARSE-C-TYPE`). *`name`*, a `STRING`, is mostly useful for documentation and interactive debugging since it appears in the printed representation of the `FFI:FOREIGN-VARIABLE` object, as in `#<FFI:FOREIGN-VARIABLE "foo" #x0ADD4E55>`. In effect, this is similar to `FFI:CAST` (or rather `(FFI:OFFSET ... 0 ...)` for places), except that it works with `FFI:FOREIGN-ADDRESS` objects and allows caching of the internal **C** types.

`(FFI:FOREIGN-FUNCTION` *`foreign-entity c-type-internal`* `&KEY` *`name`*`)`

This constructor creates a `FFI:FOREIGN-FUNCTION` from the given `FFI:FOREIGN-ADDRESS` or `FFI:FOREIGN-FUNCTION` and the internal **C** type descriptor (as obtained from `(FFI:PARSE-C-TYPE '(FFI:C-FUNCTION ...))`, in which case it is important to specify the `:LANGUAGE` because the expressions are likely to be evaluated at run time, outside the compilation unit). *`name`*, a `STRING`, is mostly useful for documentation and interactive debugging since it appears in the printed representation of the `FFI:FOREIGN-FUNCTION` object, as in `#<FFI:FOREIGN-FUNCTION "foo" #x0052B060>`. It is inherited from the given `FFI:FOREIGN-FUNCTION` object when available.

`(FFI:VALIDP` *`foreign-entity`*`)`
`(SETF (FFI:VALIDP` *`foreign-entity`*`)` *`value`*`)`

This predicate returns `NIL` if the *`foreign-entity`* (e.g. the Lisp equivalent of a FFI:C-POINTER) refers to a pointer which is invalid

(e.g., because it comes from a previous Lisp session). It returns `T` if *`foreign-entity`* can be used within the current Lisp process (thus it returns `T` for all non-foreign arguments).
You can invalidate a foreign object using `(SETF FFI:VALIDP)`.
You cannot resurrect a zombie, nor can you kill a non-foreign object.

`(FFI:FOREIGN-POINTER foreign-entity)`

`FFI:FOREIGN-POINTER` returns the `FFI:FOREIGN-POINTER` associated with the Lisp object of type `FFI:FOREIGN-ADDRESS`, `FFI:FOREIGN-POINTER`, `FFI:FOREIGN-VARIABLE` or `FFI:FOREIGN-FUNCTION`.

`(FFI:SET-FOREIGN-POINTER foreign-entity {foreign-entity | :COPY})`

`FFI:SET-FOREIGN-POINTER` changes the `FFI:FOREIGN-POINTER` associated with the Lisp object of type `FFI:FOREIGN-ADDRESS`, `FFI:FOREIGN-VARIABLE` or `FFI:FOREIGN-FUNCTION` to that of the other entity. With `:COPY`, a fresh `FFI:FOREIGN-POINTER` is allocated. The original *`foreign-entity`* still points to the same object and is returned. This is particularly useful with `(SETF FFI:VALIDP)`, see Example 32.10, "Controlling validity of resources".

`(FFI:WITH-FOREIGN-OBJECT (variable c-type [initarg]) body)`
`(FFI:WITH-C-VAR (variable c-type [initarg]) body)`

These forms allocate space on the **C** execution stack, bind respectively a `FFI:FOREIGN-VARIABLE` object or a local `SYMBOL-MACRO` to *`variable`* and execute *`body`*.
When *`initarg`* is not supplied, they allocate space only for `(FFI:SIZEOF c-type)` bytes. This space is filled with zeroes. E.g., using a *`c-type`* of FFI:C-STRING or even `(FFI:C-PTR (FFI:C-ARRAY uint8 32))` (!) both allocate space for a single pointer, initialized to `NULL`.
When *`initarg`* is supplied, they allocate space for an arbitrarily complex set of structures rooted in *`c-type`*. Therefore, FFI:C-ARRAY-MAX, `#()` and `""` are your friends for creating a pointer to the empty arrays:

```
(with-c-var (v '(c-ptr (c-array-max uint8 32)) #())
  (setf (element (deref v) 0) 127) v)
```

`c-type` is evaluated, making creation of variable sized buffers easy:

```
(with-c-var (fv `(c-array uint8 ,(length my-vector)) m
  (print fv))
```

(FFI:FOREIGN-VALUE FFI:FOREIGN-VARIABLE)
(SETF (FFI:FOREIGN-VALUE FFI:FOREIGN-VARIABLE) ...)

This functions converts the reference to a **C** data structure which the `FFI:FOREIGN-VARIABLE` describes, to Lisp. Such a reference is typically obtained from `FFI:ALLOCATE-SHALLOW`, `FFI:ALLOCATE-DEEP`, `FFI:FOREIGN-ALLOCATE` or via a `(FFI:C-POINTER` ***`c-type`***`)` **C** type description. Alternatively, macros like `FFI:WITH-C-PLACE` or `FFI:WITH-C-VAR` and the concept of foreign place hide many uses of this function.

The `SETF` form performs conversion from Lisp to **C**, following to the `FFI:FOREIGN-VARIABLE`'s type description.

(FFI:WITH-FOREIGN-STRING (*foreign-address char-count byte-count string* &KEY *encoding null-terminated-p start end*) &BODY *body*)

This forms converts a Lisp *`string`* according to the *`encoding`*, allocating space on the **C** execution stack. *`encoding`* can be any `EXT:ENCODING`, e.g. `CHARSET:UTF-16` or `CHARSET:UTF-8`, whereas `CUSTOM:*FOREIGN-ENCODING*` must be an **ASCII**-compatible encoding.

`body` is then executed with the three variables *`foreign-address`*, *`char-count`* and *`byte-count`* respectively bound to an untyped `FFI:FOREIGN-ADDRESS` (as known from the FFI:C-POINTER foreign type specification) pointing to the stack location, the number of `CHARACTER`s of the Lisp *`string`* that were considered and the number of `(UNSIGNED-BYTE 8)` bytes that were allocated for it on the **C** stack.

When *`null-terminated-p`* is true, which is the default, a variable number of zero bytes is appended, depending on the encoding, e.g. 2 for `CHARSET:UTF-16`, and accounted for in *`byte-count`*, and *`char-count`* is incremented by one.

The `FFI:FOREIGN-ADDRESS` object bound to *`foreign-address`* is invalidated upon the exit from the form.

A stupid example (a quite costly interface to `mblen`):

```
(with-foreign-string (fv elems bytes string
                      :encoding charset:jis... :null-t
                      :end 5)
 (declare (ignore fv elems))
 (format t "This string would take ~D bytes." bytes))
```

(FFI:PARSE-C-TYPE *c-type*)
(FFI:DEPARSE-C-TYPE *c-type-internal*)

Convert between the external (`LIST`) and internal (`VECTOR`) **C** type representations (used by `DESCRIBE`).

Note

Although you can memoize a *`c-type-internal`* (see Section 31.11.3, "Macro EXT:MEMOIZED" - but do not expect type redefinitions to work across memoization!), you cannot serialize it (write to disk) because deserialization loses object identity.

(FFI:ALLOCATE-SHALLOW *c-type* &KEY :COUNT :READ-ONLY)
(FFI:ALLOCATE-DEEP *c-type contents* &KEY :COUNT :READ-ONLY)
(FFI:FOREIGN-FREE *foreign-entity* &KEY :FULL)
(FFI:FOREIGN-ALLOCATE *c-type-internal* &KEY :INITIAL-CONTENTS :COUNT :READ-ONLY)

Macro `FFI:ALLOCATE-SHALLOW` allocates `(FFI:SIZEOF c-type)` bytes on the **C** heap and zeroes them out (like `calloc`).
When `:COUNT` is supplied, *`c-type`* is substituted with `(FFI:C-ARRAY c-type count)`, except when *`c-type`* is `CHARACTER`, in which case `(FFI:C-ARRAY-MAX CHARACTER count)` is used instead. When `:READ-ONLY` is supplied, the Lisp side is prevented from modifying the memory contents. This can be used as an indication that some foreign side is going to fill this memory (e.g. via `read`).
Returns a `FFI:FOREIGN-VARIABLE` object of the actual *`c-type`*, whose address part points to the newly allocated memory.
`FFI:ALLOCATE-DEEP` will call **C** `malloc` as many times as necessary to build a structure on the **C** heap of the given *`c-type`*, initialized from the given *`contents`*.
E.g., `(FFI:ALLOCATE-DEEP 'FFI:C-STRING "ABCDE")` performs 2 allocations: one for a **C** pointer to a string, another for the contents of that string. This would be useful in conjunction with a char** **C** type declaration. `(FFI:ALLOCATE-SHALLOW 'FFI:C-STRING)` allocates room for a single pointer (probably 4 bytes).
`(FFI:ALLOCATE-DEEP 'CHARACTER "ABCDEF" :count 10)` allocates and initializes room for the type `(FFI:C-ARRAY-MAX`

CHARACTER 10), corresponding to char* or, more specifically, char [10] in C.

Function FFI:FOREIGN-FREE deallocates memory at the address held by the given *foreign-entity*. If :FULL is supplied and the argument is of type FFI:FOREIGN-VARIABLE, recursively frees the whole complex structure pointed to by this variable.

If given a FFI:FOREIGN-FUNCTION object that corresponds to a **CLISP** callback, deallocates it. Callbacks are automatically created each time you pass a Lisp function via the **"FFI"**.

Use (SETF FFI:VALIDP) to disable further references to this address from Lisp. This is currently not done automatically. If the given pointer is already invalid, FFI:FOREIGN-FREE (currently) SIGNALs an ERROR. This may change to make it easier to integrate with EXT:FINALIZE.

Function FFI:FOREIGN-ALLOCATE is a lower-level interface as it requires an internal C type descriptor as returned by FFI:PARSE-C-TYPE.

(FFI:WITH-C-PLACE (*variable foreign-entity*) *body*)

Create a place out of the given FFI:FOREIGN-VARIABLE object so operations on places (e.g. FFI:CAST, FFI:DEREF, FFI:SLOT etc.) can be used within *body*. FFI:WITH-C-VAR appears as a composition of FFI:WITH-FOREIGN-OBJECT and FFI:WITH-C-PLACE.

Such a place can be used to access memory referenced by a *foreign-entity* object:

```
(setq foo (allocate-deep '(c-array uint8 3) rgb))
(with-c-place (place foo) (element place 0))
```

FFI:*OUTPUT-C-FUNCTIONS*
FFI:*OUTPUT-C-VARIABLES*

CLISP will write the extern declarations for foreign functions (defined with FFI:DEF-CALL-OUT) and foreign variables (defined with FFI:DEF-C-VAR) into the output #P".c" (when the Lisp file is compiled with COMPILE-FILE) *unless* these variables are NIL. They are NIL by default, so the extern declarations are **not** written; you are encouraged to use FFI:C-LINES to include the appropriate C headers. Set these variables to non-NIL if the headers are not available or not usable.

FFI:*FOREIGN-GUARD*

When this variable is non-NIL at compile time, **CLISP** will guard the **C** statements in the output file with **cpp** conditionals to take advantage of GNU autoconf feature detection. E.g.,

```
(eval-when (compile) (setq *foreign-guard* t))
(def-call-out some-function (:name "function_name") ..
```

will produce

```
# if defined(HAVE_FUNCTION_NAME)
  register_foreign_function((void*)&function_name,"fur
# endif
```

and will compile and link on any system.
This is mostly useful for product delivery when you want your module to build on any system even if some features will not be available.
`FFI:*FOREIGN-GUARD*` is initialized to `NIL` for backwards compatibility.

Low-level "FFI" forms

(`FFI:MEMORY-AS` *foreign-address c-type-internal* `&OPTIONAL` *offset*)
(`SETF` (`FFI:MEMORY-AS` *foreign-address c-type-internal* `&OPTIONAL` *offset*) *value*)

This accessor is useful when operating with untyped foreign pointers (`FFI:FOREIGN-ADDRESS`) as opposed to typed ones (represented by `FFI:FOREIGN-VARIABLE`). It allows to type and dereference the given pointer without the need to create an object of type `FFI:FOREIGN-VARIABLE`.
Alternatively, one could use (`FFI:FOREIGN-VALUE` (`FFI:FOREIGN-VARIABLE` *foreign-entity c-type-internal*)) (also `SETF`able).
Note that *c-type-internal* is the *internal* representation of a foreign type, thus `FFI:PARSE-C-TYPE` is required with literal names or types, e.g. `(FFI:MEMORY-AS` *foreign-address* `(FFI:PARSE-C-TYPE '(FFI:C-ARRAY uint8 3)))` or `(SETF (FFI:MEMORY-AS` *foreign-address* `(FFI:PARSE-C-TYPE 'uint32)) 0)`.

32.3.3. (Foreign) C types

Foreign C types are used in the **"FFI"**. They are **not** regular **Common Lisp** types or **CLOS** classes.

A *`c-type`* is either a predefined C type or the name of a type defined by `FFI:DEF-C-TYPE`.

the predefined C types (*`c-type`*)

`simple-c-type`

the simple C types

Lisp name	Lisp equivalent	C equivalent	ILU equivalent	Comment
`NIL`	`NIL`	void		as a result type only
`BOOLEAN`	`BOOLEAN`	int	BOOLEAN	
`CHARACTER`	`CHARACTER`	char	SHORT CHARACTER	
char	`INTEGER`	signed char		
uchar	`INTEGER`	unsigned char		
short	`INTEGER`	short		
ushort	`INTEGER`	unsigned short		
int	`INTEGER`	int		
uint	`INTEGER`	unsigned int		
long	`INTEGER`	long		
ulong	`INTEGER`	unsigned long		
uint8	`(UNSIGNED-BYTE 8)`	uint8	BYTE	
sint8	`(SIGNED-BYTE 8)`	sint8		

Lisp name	Lisp equivalent	C equivalent	ILU equivalent	Comment
uint16	(UNSIGNED-BYTE 16)	uint16	SHORT CARDINAL	
sint16	(SIGNED-BYTE 16)	sint16	SHORT INTEGER	
uint32	(UNSIGNED-BYTE 32)	uint32	CARDINAL	
sint32	(SIGNED-BYTE 32)	sint32	INTEGER	
uint64	(UNSIGNED-BYTE 64)	uint64	LONG CARDINAL	does not work on all platforms
sint64	(SIGNED-BYTE 64)	sint64	LONG INTEGER	does not work on all platforms
SINGLE-FLOAT	SINGLE-FLOAT	float		
DOUBLE-FLOAT	DOUBLE-FLOAT	double		

FFI:C-POINTER

This type corresponds to what **C** calls void*, an opaque pointer. When used as an argument, `NIL` is accepted as a FFI:C-POINTER and treated as `NULL`; when a function wants to return a `NULL` FFI:C-POINTER, it actually returns `NIL`.

`(FFI:C-POINTER` *`c-type`*`)`

This type is equivalent to what **C** calls *`c-type`* *: a pointer to a single item of the given *`c-type`*. It differs from `(FFI:C-PTR-NULL` *`c-type`*`)` (see below) in that no conversion to and from Lisp will occur (beyond the usual one of the **C** `NULL` pointer to or from Lisp `NIL`). Instead, an object of type `FFI:FOREIGN-VARIABLE` is used to represent the foreign place. It is assimilable to a typed pointer.

FFI:C-STRING

This type corresponds to what **C** calls char*, a zero-terminated string. Its Lisp equivalent is a string, without the trailing zero character.

`(FFI:C-STRUCT` *`class`* `(`*`ident`*$_1$ *`c-type`*$_1$`) ... (`*`ident`*$_n$ *`c-type`*$_n$`))`

This type is equivalent to what **C** calls struct { `c-type`$_1$ `ident`$_1$; ...; `c-type`$_n$ `ident`$_n$; }. Its Lisp equivalent is: if `class` is `VECTOR`, a `SIMPLE-VECTOR`; if `class` is `LIST`, a proper list; if `class` is a symbol naming a structure or **CLOS** class, an instance of this class, with slots of names `ident`$_1$, ..., `ident`$_n$.
`class` may also be a `CONS` of a `SYMBOL` (as above) and a `LIST` of `FFI:DEF-C-STRUCT` options.

(FFI:C-UNION (*ident*$_1$ *c-type*$_1$) ... (*ident*$_n$ *c-type*$_n$))

This type is equivalent to what **C** calls union { `c-type`$_1$ `ident`$_1$; ...; `c-type`$_n$ `ident`$_n$; }. Conversion to and from Lisp assumes that a value is to be viewed as being of `c-type`$_1$.

(FFI:C-ARRAY *c-type* *dim*$_1$)
(FFI:C-ARRAY *c-type* (*dim*$_1$ ... *dim*$_n$))

This type is equivalent to what **C** calls `c-type` [`dim`$_1$] ... [`dim`$_n$]. Note that when an array is passed as an argument to a function in **C**, it is actually passed as a pointer; you therefore have to write `(FFI:C-PTR (FFI:C-ARRAY ...))` for this argument's type.

(FFI:C-ARRAY-MAX *c-type* *maxdimension*)

This type is equivalent to what **C** calls `c-type` [`maxdimension`], an array containing up to `maxdimension` elements. The array is zero-terminated if it contains less than `maxdimension` elements. Conversion from Lisp of an array with more than `maxdimension` elements silently ignores the superfluous elements.

(FFI:C-FUNCTION (:ARGUMENTS {(*argument* *a-c-type* [*PARAM-MODE* [*ALLOCATION*]])}*) (:RETURN-TYPE *r-c-type* [*ALLOCATION*]) (:LANGUAGE *language*))

This type designates a **C** function that can be called according to the given prototype `(r-c-type (*) (a-c-type`$_1$`, ...))`. Conversion between **C** functions and Lisp functions is transparent, and `NULL`/`NIL` is recognized and accepted.

(FFI:C-PTR *c-type*)

This type is equivalent to what **C** calls `c-type` *: a pointer to a single item of the given `c-type`.

(FFI:C-PTR-NULL *c-type*)

This type is also equivalent to what **C** calls `c-type` *: a pointer to a single item of the given `c-type`, with the exception that **C** `NULL` corresponds to Lisp `NIL`.

(FFI:C-ARRAY-PTR *c-type*)

This type is equivalent to what C calls *c-type* (*)[]: a pointer to a zero-terminated array of items of the given *c-type*.

The conversion of FFI:C-STRING, (FFI:C-ARRAY CHARACTER *dim*$_1$), (FFI:C-ARRAY-MAX CHARACTER *maxdimension*), (FFI:C-ARRAY-PTR CHARACTER) is governed by CUSTOM:*FOREIGN-ENCODING* and dimensions are given in *bytes*. The conversion of CHARACTER, and as such of (FFI:C-PTR CHARACTER), or (FFI:C-PTR-NULL CHARACTER), as well as that of multi-dimensional arrays (FFI:C-ARRAY CHARACTER (*dim*$_1$ ... *dim*$_n$)), are governed by CUSTOM:*FOREIGN-ENCODING* if the latter is a 1:1 encoding, or by the **ASCII** encoding otherwise.

Note

Remember that the **C** type char is a *numeric* type and does not use CHARACTER EXT:ENCODINGs.

32.3.4. The choice of the C flavor

FFI:C-FUNCTION, FFI:DEF-CALL-IN, FFI:DEF-CALL-OUT take a :LANGUAGE argument. The *language* is either :C (denotes K&R **C**) or :STDC (denotes ANSI **C**) or :STDC-STDCALL (denotes ANSI **C** with the "stdcall" calling convention). It specifies whether the **C** function (caller or callee) has been compiled by a K&R **C** compiler or by an ANSI **C** compiler, and possibly the calling convention.

The default language is set using the macro ***FFI:DEFAULT-FOREIGN-LANGUAGE*** . If this macro has not been called in the current compilation unit (usually a file), a warning is issued and :STDC is used for the rest of the unit.

32.3.5. Foreign variables

Foreign variables are variables whose storage is allocated in the foreign language module. They can nevertheless be evaluated and modified through SETQ, just as normal variables can, except that the range of allowed values is limited according to the variable's foreign type.

Equality of foreign values.

For a foreign variable *x* the form (EQL *x* *x*) is not necessarily true, since every time *x* is evaluated its foreign value is converted to a fresh Lisp value. Ergo, (SETF (AREF *x* *n*) *y*) modifies this fresh Lisp value (immediately discarded), **not** the foreign data. Use FFI:ELEMENT et al instead, see Section 32.3.6, "Operations on foreign places".

Foreign variables are defined using FFI:DEF-C-VAR and FFI:WITH-C-VAR.

32.3.6. Operations on foreign places

A FFI:FOREIGN-VARIABLE *name* defined by FFI:DEF-C-VAR, FFI:WITH-C-VAR or FFI:WITH-C-PLACE defines a place, i.e., a form which can also be used as argument to SETF. (An "lvalue" in **C** terminology.) The following operations are available on foreign places:

FFI:ELEMENT	FFI:C-VAR-ADDRESS
FFI:DEREF	FFI:C-VAR-OBJECT
FFI:SLOT	FFI:TYPEOF
FFI:CAST	FFI:SIZEOF
FFI:OFFSET	FFI:BITSIZEOF

32.3.7. Foreign functions

Foreign functions are functions which are defined in the foreign language. There are ***named foreign functions*** (imported via FFI:DEF-CALL-OUT or created via FFI:DEF-CALL-IN) and ***anonymous foreign functions***; they arise through conversion of function pointers.

A ***call-out function*** is a foreign function called from Lisp: control flow temporarily leaves Lisp. A ***call-in function*** is a Lisp function called from the foreign language: control flow temporary enters Lisp.

The following operators define foreign functions:

`FFI:DEF-CALL-IN` `FFI:FOREIGN-FUNCTION`
`FFI:DEF-CALL-OUT`

32.3.8. Argument and result passing conventions

When passed to and from functions, allocation of arguments and results is handled as follows:

Values of ***`SIMPLE-C-TYPE`***, FFI:C-POINTER are passed on the stack, with dynamic extent. The ***`ALLOCATION`*** is effectively ignored.

Values of type FFI:C-STRING, FFI:C-PTR, FFI:C-PTR-NULL, FFI:C-ARRAY-PTR need storage. The ***`ALLOCATION`*** specifies the allocation policy:

`:NONE`
: no storage is allocated.

`:ALLOCA`
: allocation of storage on the stack, which has dynamic extent.

`:MALLOC-FREE`
: storage will be allocated via `malloc` and released via `free`.

If no ***`ALLOCATION`*** is specified, the default ***`ALLOCATION`*** is `:NONE` for most types, but `:ALLOCA` for FFI:C-STRING and FFI:C-PTR and FFI:C-PTR-NULL and FFI:C-ARRAY-PTR and for `:OUT` arguments.
The `:MALLOC-FREE` policy provides the ability to pass arbitrarily nested structures within a single conversion.

Call-out function arguments:

For arguments passed from Lisp to C:

`:MALLOC-FREE`
: Lisp allocates the storage using `malloc` and never deallocates it. The **C** function is supposed to call `free` when done with it.

`:ALLOCA`
: Lisp allocates the storage on the stack, with dynamic extent. It is freed when the **C** function returns.

`:NONE`

Lisp assumes that the pointer already points to a valid area of the proper size and puts the result value there.
This is dangerous and deprecated.

For results passed from C to Lisp:

`:MALLOC-FREE`
: Lisp calls `free` on it when done.

`:NONE`
: Lisp does nothing.

Call-in function arguments:

For arguments passed from C to Lisp:

`:MALLOC-FREE`
: Lisp calls `free` on it when done.

`:ALLOCA`

`:NONE`
: Lisp does nothing.

For results passed from Lisp to C:

`:MALLOC-FREE`
: Lisp allocates the storage using `malloc` and never deallocates it. The C function is supposed to call `free` when done with it.

`:NONE`
: Lisp assumes that the pointer already points to a valid area of the proper size and puts the result value there.
This is dangerous and deprecated.

Warning

Passing FFI:C-STRUCT, FFI:C-UNION, FFI:C-ARRAY, FFI:C-ARRAY-MAX values as arguments (not via pointers) is only possible to the extent the C compiler supports it. Most C compilers do it right, but some C compilers (such as **gcc** on **hppa**, **x86_64** and **Win32**) have problems with this. The recommended workaround is to pass pointers; this is fully supported. See also the <`clisp-list@lists.sourceforge.net`>

(http://lists.sourceforge.net/lists/listinfo/clisp-list) (`SFmail/5513622`, `Gmane/devel/10089`).

32.3.9. Parameter Mode

A function parameter's *`PARAM-MODE`* may be

`:IN` (means: read-only):
: The caller passes information to the callee.

`:OUT` (means: write-only):
: The callee passes information back to the caller on return. When viewed as a Lisp function, there is no Lisp argument corresponding to this, instead it means an additional return value. Requires *`ALLOCATION`* = `:ALLOCA`.

`:IN-OUT` (means: read-write):
: Information is passed from the caller to the callee and then back to the caller. When viewed as a Lisp function, the `:OUT` value is returned as an additional multiple value.

The default is `:IN`.

32.3.10. Examples

Example 32.1. Simple declarations and access

The **C** declaration

```
struct foo {
    int a;
    struct foo * b[100];
};
```

corresponds to

```
(def-c-struct foo
  (a int)
  (b (c-array (c-ptr foo) 100)))
```

The element access

```
struct foo f;
f.b[7].a
```

corresponds to

```
(declare (type foo f))
(foo-a (aref (foo-b f) 7)) or (slot-value (aref (slot-val
```

Example 32.2. external C variable and some accesses

```
struct bar {
    short x, y;
    char a, b;
    int z;
    struct bar * n;
};

extern struct bar * my_struct;

my_struct->x++;
my_struct->a = 5;
my_struct = my_struct->n;
```

corresponds to

```
(def-c-struct bar
  (x short)
  (y short)
  (a char)
  (b char) ; or (b character) if it represents a characte
  (z int)
  (n (c-ptr bar)))

(def-c-var my_struct (:type (c-ptr bar)))

(setq my_struct (let ((s my_struct)) (incf (slot-value s
or (incf (slot my_struct 'x))
(setq my_struct (let ((s my_struct)) (setf (slot-value s
or (setf (slot my_struct 'a) 5)
(setq my_struct (slot-value my_struct 'n))
or (setq my_struct (deref (slot my_struct 'n)))
```

Example 32.3. Calling an external function

On ANSI C systems, <stdlib.h> contains the declarations:

```
typedef struct {
  int quot;    /* Quotient */
  int rem;     /* Remainder */
} div_t;
extern div_t div (int numer, int denom);
```

This translates to

```
(def-c-struct (div_t :typedef)
  (quot int)
  (rem int))
(default-foreign-language :stdc)
(def-call-out div (:arguments (numer int) (denom int))
  (:return-type div_t))
```

Sample call from within Lisp (after running **clisp-link**):

```
(div 20 3)
⇒ #S(DIV_T :QUOT 6 :REM 2)
```

Example 32.4. Another example for calling an external function

Suppose the following is defined in a file `cfun.c`:

```
struct cfunr { int x; char *s; };
struct cfunr * cfun (int i,char *s,struct cfunr * r,int a
  int j;
  struct cfunr * r2;
  printf("i = %d\n", i);
  printf("s = %s\n", s);
  printf("r->x = %d\n", r->x);
  printf("r->s = %s\n", r->s);
  for (j = 0; j < 10; j++) printf("a[%d] = %d.\n", j, a[j
  r2 = (struct cfunr *) malloc (sizeof (struct cfunr));
  r2->x = i+5;
  r2->s = "A C string";
```

```
  return r2;
}
```

It is possible to call this function from Lisp using the file `callcfun.lisp` (do not call it `cfun.lisp` - `COMPILE-FILE` will overwrite `cfun.c`) whose contents is:

```
(DEFPACKAGE "TEST-C-CALL" (:use "COMMON-LISP" "FFI"))
(IN-PACKAGE "TEST-C-CALL")
(eval-when (compile) (setq FFI:*OUTPUT-C-FUNCTIONS* t))
(def-c-struct cfunr (x int) (s c-string))
(default-foreign-language :stdc)
(def-call-out cfun
    (:arguments (i int)
                (s c-string)
                (r (c-ptr cfunr) :in :alloca)
                (a (c-ptr (c-array int 10)) :in :alloca))
  (:return-type (c-ptr cfunr)))
(defun call-cfun ()
  (cfun 5 "A Lisp string" (make-cfunr :x 10 :s "Another L
        '#(0 1 2 3 4 5 6 7 8 9)))
```

Use the module facility:

```
$ clisp-link create-module-set cfun callcfun.c
$ cc -O -c cfun.c
$ cd cfun
$ ln -s ../cfun.o cfun.o
Add cfun.o to NEW_LIBS and NEW_FILES in link.sh.
$ cd ..
$ base/lisp.run -M base/lispinit.mem -c callcfun.lisp
$ clisp-link add-module-set cfun base base+cfun
$ base+cfun/lisp.run -M base+cfun/lispinit.mem -i callcfu
> (test-c-call::call-cfun)
i = 5
s = A Lisp string
r->x = 10
r->s = Another Lisp string
a[0] = 0.
a[1] = 1.
a[2] = 2.
a[3] = 3.
a[4] = 4.
a[5] = 5.
a[6] = 6.
```

```
a[7] = 7.
a[8] = 8.
a[9] = 9.
#S(TEST-C-CALL::CFUNR :X 10 :S "A C string")
>
$ rm -r base+cfun
```

Note that there is a memory leak here: The return value `r2` of `cfun()` is `malloc`ed but never `free`d. Specifying

```
(:return-type (c-ptr cfunr) :malloc-free)
```

is not an alternative because this would also `free(r2->x)` but `r2->x` is a pointer to static data.

The memory leak can be avoided using

```
(:return-type (c-pointer cfunr))
```

instead, in conjunction with

```
(defun call-cfun ()
  (let ((data (cfun ...)))
    (prog1 (FFI:FOREIGN-VALUE data)
      (FFI:FOREIGN-FREE data :FULL nil))))
```

Example 32.5. Accessing cpp macros

Suppose you are interfacing to a library `mylib.so` which defines macros and `inline` functions in `mylib.h`:

```
#define FOO(x)  .....
inline int bar (int x) { ... }
```

To make them available from **CLISP**, write these forms into the lisp file `my.lisp`:

```
(FFI:C-LINES "#include <mylib.h>
int my_foo (int x) { return FOO(x); }
int my_bar (int x) { return bar(x); }~%")
(FFI:DEF-CALL-OUT my-foo (:name "my_foo") (:arguments (x :
(FFI:DEF-CALL-OUT my-bar (:name "my_bar") (:arguments (x :
```

Compiling this file will produce `my.c` and `my.fas` and you have two options:

1. Compile `my.c` into `my.o` with

   ```
   $ gcc -c my.c -lmylib
   ```

   and use **clisp-link** to create a new **CLISP** linking set.

2. Add `(:library "my.dll")` to the `FFI:DEF-CALL-OUT` forms, compile `my.c` into `my.so` (or `my.dll` on **Win32**) with

   ```
   $ gcc -shared -o my.so my.c -lmylib
   ```

   and load `my.fas`.

Of course, you could have created `my1.c` containing

```
#include <mylib.h>
int my_foo (int x) { return FOO(x); }
int my_bar (int x) { return bar(x); }
```

manually, but `FFI:C-LINES` allows you to keep the definitions of `my_foo` and `my-foo` close together for easier maintenance.

Example 32.6. Calling Lisp from C

To sort an array of double-floats using the Lisp function `SORT` instead of the **C** library function `qsort`, one can use the following interface code `sort1.c`. The main problem is to pass a variable-sized array.

```
extern void lispsort_begin (int);
void* lispsort_function;
void lispsort_double (int n, double * array) {
    double * sorted_array;
    int i;
    lispsort_begin(n); /* store #'sort2 in lispsort_funct
    sorted_array = ((double * (*) (double *)) lispsort_fu
    for (i = 0; i < n; i++) array[i] = sorted_array[i];
    free(sorted_array);
}
```

This is accompanied by `sort2.lisp`:

```
(DEFPACKAGE "FFI-TEST" (:use "COMMON-LISP" "FFI"))
(IN-PACKAGE "FFI-TEST")
(eval-when (compile) (setq FFI:*OUTPUT-C-FUNCTIONS* t))
(def-call-in lispsort_begin (:arguments (n int))
  (:return-type nil)
  (:language :stdc))
(def-c-var lispsort_function (:type c-pointer))
(defun lispsort_begin (n)
  (setf (cast lispsort_function
              `(c-function
                 (:arguments (v (c-ptr (c-array double-fl
                 (:return-type (c-ptr (c-array double-flo
                               :malloc-free)))
        #'sort2))
(defun sort2 (v)
  (declare (type vector v))
  (sort v #'<))
```

To test this, use the following test file `sorttest.lisp`:

```
(eval-when (compile) (setq FFI:*OUTPUT-C-FUNCTIONS* t))
(def-call-out sort10
  (:name "lispsort_double")
  (:language :stdc)
  (:arguments (n int)
              (array (c-ptr (c-array double-float 10)) :i
```

Now try

```
$ clisp-link create-module-set sort sort2.c sorttest.c
$ cc -O -c sort1.c
$ cd sort
$ ln -s ../sort1.o sort1.o
```

Add `sort1.o` to `NEW_LIBS` and `NEW_FILES` in **link.sh**. Create a file `package.lisp` containing the form

```
(MAKE-PACKAGE "FFI-TEST" :use '("COMMON-LISP" "FFI"))
```

and add `package.lisp` to `TO_PRELOAD` in **link.sh**. Proceed:

```
$ cd ..
$ base/lisp.run -M base/lispinit.mem -c sort2.lisp sortte
```

```
$ clisp-link add-module-set sort base base+sort
$ base+sort/lisp.run -M base+sort/lispinit.mem -i sort2 s
> (sort10 10 '#(0.501d0 0.528d0 0.615d0 0.550d0 0.711d0
                0.523d0 0.585d0 0.670d0 0.271d0 0.063d0))
#(0.063d0 0.271d0 0.501d0 0.523d0 0.528d0 0.55d0 0.585d0 
$ rm -r base+sort
```

Example 32.7. Calling Lisp from C dynamically

Create a dynamic library `lispdll` (`#P".dll"` on **Win32**, `#P".so"` on **UNIX**) with the following function:

```
typedef int (*LispFunc)(int parameter);
int CallInFunc(LispFunc f) {
  return f(5)+11;
}
```

and call it from Lisp:

```
(ffi:def-call-out callout
  (:name "CallInFunc")
  (:library "lispdll.dll")
  (:arguments (function-arg
               (ffi:c-function (:arguments (number ffi:in
                               (:return-type ffi:int) (:la
  (:return-type ffi:int)
  (:language :stdc))
(defun f (x) (* x 2))
⇒ F
(callout #'f)
⇒ 21
```

Example 32.8. Variable size arguments: calling gethostname from CLISP

int **gethostname**(*name*,
namelen);

char* *name*;
size_t *namelen*;

follows a typical pattern of C "out"-parameter convention - it expects a pointer to a buffer it is going to fill. So you must view this parameter as either `:OUT` or `:IN-OUT`. Additionally, one must tell the function the size of the buffer. Here *`namelen`* is just an `:IN` parameter. Sometimes this will be an `:IN-OUT` parameter, returning the number of bytes actually filled in.

So *`name`* is actually a pointer to an array of up to *`namelen`* characters, regardless of what the poor char* C prototype says, to be used like a C string (`NULL`-termination). UNIX specifies that “host names are limited to `HOST_NAME_MAX` bytes”, which is, of course, system dependent, but it appears that 256 is sufficient.

In the present example, you can use allocation `:ALLOCA`, like you would do in C: stack-allocate a temporary.

```
(FFI:DEF-CALL-OUT gethostname
  (:arguments (name (FFI:C-PTR (FFI:C-ARRAY-MAX ffi:char :
                     :OUT :ALLOCA)
              (len ffi:int))
  (:language :stdc)
  (:return-type ffi:int))

(defun myhostname ()
  (multiple-value-bind (success name)
      ;; :OUT and :IN-OUT parameters are returned as mult:
      (gethostname 256)
    (if (zerop success) name
        (error ...)))) ;; strerror(errno)

(defvar hostname (myhostname))
```

Example 32.9. Accessing variables in shared libraries

Suppose one wants to access and modify variables that reside in shared libraries:

```
struct bar {
  double x, y;
  double out;
};
```

```
struct bar my_struct = {10.0, 20.5, 0.0};

double test_dll(struct bar *ptr)
{
  return ptr->out = ptr->out + ptr->x + ptr->y;
}
```

This is compiled to `libtest.so` (or `libtest.dll`, depending on your platform).

Use the following lisp code:

```
(USE-PACKAGE "FFI")

(FFI:DEF-C-STRUCT bar
  (x double-float)
  (y double-float)
  (out double-float))

(FFI:DEF-CALL-OUT get-own-c-float
  (:library "libtest.so")
  (:language :stdc)
  (:name "test_dll")
  (:arguments (ptr c-pointer :in :alloca))
  (:return-type double-float))

(FFI:DEF-C-VAR my-c-var (:name "my_struct")
  (:library "libtest.so") (:type (c-ptr bar)))
```

Note that `get-own-c-float` takes a FFI:C-POINTER, not a `(FFI:C-PTR bar)` as the argument.

Now you can access call `get-own-c-float` on `my-c-var`:

```
(FFI:C-VAR-ADDRESS my-c-var)
⇒ #<FOREIGN-ADDRESS #x282935D8>
(get-own-c-float (FFI:C-VAR-ADDRESS my-c-var))
⇒ 30.5d0
(get-own-c-float (FFI:C-VAR-ADDRESS my-c-var))
⇒ 61.0d0
(get-own-c-float (FFI:C-VAR-ADDRESS my-c-var))
⇒ 91.5d0
(get-own-c-float (FFI:C-VAR-ADDRESS my-c-var))
⇒ 122.0d0
```

Example 32.10. Controlling validity of resources

`FFI:SET-FOREIGN-POINTER` is useful in conjunction with `(SETF FFI:VALIDP)` to limit the extent of external resources. Closing twice can be avoided by checking `FFI:VALIDP`. All pointers depending on this resource can be disabled at once upon close by sharing their `FFI:FOREIGN-POINTER` using `FFI:SET-FOREIGN-POINTER`.

```
(def-c-type PGconn c-pointer) ; opaque pointer
(def-call-out PQconnectdb (:return-type PGconn)
  (:arguments (conninfo c-string)))
(defun sql-connect (conninfo)
  (let ((conn (PQconnectdb conninfo)))
    (unless conn (error "NULL pointer"))
    ;; may wish to use EXT:FINALIZE as well
    (FFI:SET-FOREIGN-POINTER conn :COPY)))
(defun sql-dependent-resource (conn arg1)
  (let ((res (PQxxx conn arg1)))
    (FFI:SET-FOREIGN-POINTER res conn)))
(defun sql-close (connection)
  (when (FFI:VALIDP connection)
    (PQfinish connection)
    (setf (FFI:VALIDP connection) nil)
    T))
```

Warning

Sharing `FFI:FOREIGN-POINTER` goes both ways: invalidating the dependent resource will invalidate the primary one.

Note

An alternative approach to resource management, more suitable to non-**“FFI”** modules, is implemented in the berkeley-db module, see Section 33.6.2, “Closing handles”.

Example 32.11. Float point array computations

Save this code into `sum.c`:

```
double sum (int len, double *vec) {
  int i;
  double s=0;
  for (i=0; i<len; i++) s+= vec[i];
  return s;
}
```

and compile it with

```
$ gcc -shared -o libsum.so sum.c
```

Now you can sum doubles:

```
(FFI:DEF-CALL-OUT sum (:name "sum") (:library "libsum.so")
  (:return-type double-float)
  (:arguments (len int) (vec (FFI:C-ARRAY-PTR double-floa
(sum 3 #(1d0 2d0 3d0))
⇒ 6d0
```

32.3.10.1. More examples

You can find more information and examples of the **CLISP** **“FFI”** in the following <`clisp-list@lists.sourceforge.net`> (http://lists.sourceforge.net/lists/listinfo/clisp-list) messages:

variable size values
: `SFmail/5736140`, `Gmane/general/7278`

variable length arrays
: `SFmail/4062459`, `Gmane/general/6626`

Even more examples can be found in the file `tests/ffi.tst` in the **CLISP** source distribution.

32.4. The Amiga Foreign Function Call Facility

Platform Dependent: No platform supports this currently

Another Foreign Function Interface. All symbols relating to the simple foreign function interface are exported from the package **"AFFI"**. To use them, `(USE-PACKAGE "AFFI")`.

32.4.1. Design issues

"AFFI" was designed to be small in size but powerful enough to use most library functions. Lisp files may be compiled to `#P".fas"` files without the need to load function definition files at run-time and without external **C** or linker support. memory images can be created, provided that the function libraries are opened at run-time.

Therefore, **"AFFI"** supports only primitive **C** types (integers 8, 16 and 32 bits wide, signed or unsigned, pointers) and defines no new types or classes. Foreign functions are not first-class objects (you can define a `LAMBDA` yourself), name spaces are separate.

The **"AFFI"** does no tracking of resources. Use `EXT:FINALIZE`.

32.4.2. Overview

These are the **"AFFI"** forms:

(declare-library-base *keyword-base library-name*)

(require-library-functions *library-name* [(:import {*string-name*}*)])

(open-library *base-symbol*)

(close-library *base-symbol*)

(with-open-library (*base-symbol* | *library-name*) {*form*}*)

(defflibfun *function-name base-symbol offset mask result-type* {*argument-type*}*)

(declare-library-function *function-name library-name* {*option*}*)

(flibcall *function-name* {*argument*}*)

(mlibcall *function-name* {*argument*}*)

(mem-read *address result-type* [*offset*])

(mem-write *address type value* [*offset*])

(mem-write-vector *address vector* [*offset*])

(nzero-pointer-p *value*)

Except for `with-open-library`, `declare-library-function` and `mlibcall`, all of the above are functions.

A library contains a collection of functions. The library is referred to by a symbol referred as library-base at the **"AFFI"** level. This symbol is created in the package **"AFFI"**. The link between this symbol and the OS-level library name is established by `declare-library-base`. To avoid multiple package conflicts, this and only this function requires the symbol-name to be in the **"KEYWORD"** package. The function returns the library-base.

A library may be opened by `open-library` and closed by `close-library`. An opened library must be closed. `with-open-library` is provided to automatically close the library for you, thus it is much safer to use.

A function is contained in a library. Every function is referred to by a symbol. A function is defined through `defflibfun` or `declare-library-function` by giving the function name, the library-base, an offset into the library, a mask (or `NIL`) for register-based library calls, the result type and all parameter-types. `require-library-functions` loads the complete set of functions defined in a library file. Symbols are created in the package **"AFFI"** and imported into the current package.

`flibcall` and `mlibcall` call library functions. `mlibcall` is a macro that does a few checks at macroexpansion time and allows the compiler to inline the call, not requiring the foreign function to be defined again at load or execution time. The use of this macro is advertised wherever possible.

`mem-read` reads an arbitrary address (with offset for structure references) and returns the given type.

`mem-write` writes an arbitrary address. `mem-write-vector` copies the content of a Lisp `STRING` or `(VECTOR (UNSIGNED-BYTE 8))` into memory.

`nzero-pointer-p` tests for non-`NULL` pointers in all recognized representations (`NULL`, `UNSIGNED-BYTE` and `FFI:FOREIGN-POINTER`).

32.4.3. Foreign Libraries

`declare-library-base` ought to be wrapped in an `(EVAL-WHEN (compile eval load) ...)` form and come before any function is referenced, because the library base symbol must be known.

`open-library` tries to open the library referenced by the base symbol. Therefore it must have been preceded with `declare-library-base`. The call returns `NIL` on failure. `open-library` calls nest. Every successful call must be matched by `close-library`. `with-open-library` does this for you and also allows you to specify the library by name, provided that its base has been declared. It is recommended to use this macro and to reference the library by name.

CLISP will not close libraries for you at program exit. See Section 31.1, "Customizing **CLISP** Process Initialization and Termination" and watch `AFFI::*LIBRARIES-ALIST*`.

32.4.4. (Foreign) C types

The following foreign **C** types are used in **"AFFI"**. They are **not** regular **Common Lisp** types or **CLOS** classes.

"AFFI" name	**Lisp equivalent**	**C equivalent**	**Comment**
`NIL`	`NIL`	void	as a result type for functions only
4	`(UNSIGNED-BYTE 32)`	unsigned long	
2	`(UNSIGNED-BYTE 16)`	unsigned short	
1	`(UNSIGNED-BYTE 8)`	unsigned char	
-4	`(SIGNED-BYTE 32)`	long	
-2	`(SIGNED-BYTE 16)`	short	
-1	`(SIGNED-BYTE 8)`	signed char	
0	`BOOLEAN`	BOOL	as a result type for functions only
*	opaque	void*	
`:EXTERNAL`	opaque	void*	
`STRING`	`STRING` or `VECTOR`	char*	
`:IO`	`STRING` or `VECTOR`	char*	

Objects of type STRING are copied and passed NULL-terminated on the execution stack. On return, a Lisp string is allocated and filled from the address returned (unless NULL). Functions with :IO parameters are passed the address of the Lisp string or unsigned byte vector. These are not NULL -terminated! This is useful for functions like like read-c which do not need an array at a constant address longer than the dynamic extent of the call (it is dangerous to define callback functions with :IO - or STRING - type parameters). Arguments of type INTEGER and FFI:FOREIGN-POINTER are always acceptable where a STRING or :IO type is specified.

See also CUSTOM:*FOREIGN-ENCODING*.

To meet the design goals, predefined types and objects were used. As such, pointers were represented as integers. Now that there is the FFI:FOREIGN-POINTER type, both representations may be used on input. The pointer type should be therefore considered as opaque. Use nzero-pointer-p for NULL tests.

32.4.5. Foreign functions

Foreign Functions are declared either through defflibfun or declare-library-function. The former is closer to the low-level implementation of the interface, the latter is closer to the other **"FFI"**.

defflibfun requires the library base symbol and register mask to be specified, declare-library-function requires the library name and computes the mask from the declaration of the arguments.

The value of mask is implementation-dependent.

The **"AFFI"** type 0 is only acceptable as a function result type and yields either T or NIL. The difference between * and :EXTERNAL is the following: * uses integers, :EXTERNAL uses FFI:FOREIGN-POINTER as function result-type (except from NIL for a NULL pointer) and refuses objects of type STRING or (VECTOR (UNSIGNED-BYTE 8)) as input. Thus :EXTERNAL provides some security on the input and the ability to use EXT:FINALIZE for resource-tracking on the output side.

```
(declare-library-function name library-name {option}*)
```

option ::==

`(:offset library-offset)`
`(:ARGUMENTS {(argument "AFFI"-type register)}*)`
`(:return-type "AFFI"-type)`

register ::==

`:d0` | `:d1` | … | `:d7` | `:a0` | … | `:a6`

declares a named library function for further reference through `flibcall` and `mlibcall`.

`mlibcall` should be the preferred way of calling foreign functions (when they are known at compile-time) as macroexpansion-time checks may be performed and the call can be sort of inlined.

32.4.6. Memory access

`(affi:mem-read address type offset)` can read 8, 16 and 32 bit signed or unsigned integers (**"AFFI"** types -4, -2, -1, 1, 2, 4), a pointer (*), a `NULL`-terminated string (string) or, if the type argument is of type `STRING` or `(VECTOR (UNSIGNED-BYTE 8))`, it can fill this vector. `:EXTERNAL` is not an acceptable type as no object can be created by using `affi:mem-read`.

`(affi:mem-write address type value [offset])` writes integers (**"AFFI"** type -4, -2, -1, 1, 2 and 4) or pointer values (type *), but not vectors to the specified memory address.

`(affi:mem-write-vector address vector [offset])` can write memory from the given vector (of type `STRING` or `(VECTOR (UNSIGNED-BYTE 8))`).

32.4.7. Function Definition Files

`affi:require-library-functions` will `REQUIRE` a file of name derived from the library name and with type `affi`. It may be used to import all names into the current package or only a given subset identified by string names, using the `:import` keyword (recommended use). Some definition files for standard Amiga libraries are provided. See Example 32.12, “Using a predefined library function file” below.

As affi:require-library-functions loads a global file which you, the programmer, may have not defined, you may consider declaring every function yourself to be certain what the return and argument types are. See Example 32.15, “Some sample function definitions” below.

The file read-fd.lisp defines the function make-partial-fd-file with which the provided .affi files have been prepared from the original Amiga FD files (located in the directory FD:). They must still be edited as the function cannot know whether a function accepts a *, :IO, string or :EXTERNAL argument and because files in FD: only contain a register specification, not the width of integer arguments (-4, -2, -1, 1, 2, or 4).

32.4.8. Hints

By using appropriate EVAL-WHEN forms for affi:declare-library-base and affi:require-library-functions and not using affi:flibcall, it is possible to write code that only loads library function definition files at compile-time. See Example 32.12, “Using a predefined library function file” below.

Do not rely on EXT:FINALIZE to free resources for you, as **CLISP** does not call finalizers when it exits, use UNWIND-PROTECT.

32.4.9. Caveats

You can consider the library bases being symbols in need of being imported from the package **“AFFI”** originating from a brain-damage, causing the usual symbol headaches when using foreign functions calls within macros. Luckily, even if the high-level interface (or its implementation in src/affi1.lisp) were to change, the low-level part (src/affi.d) should remain untouched as all it knows are INTEGERs and FFI:FOREIGN-POINTERs, no SYMBOLs. The difficulty is just to get the library base value at run-time. Feel free to suggest enhancements to this facility!

32.4.10. Examples

Warning

These examples are somewhat specific to the Amiga.

Example 32.12. Using a predefined library function file

```
(DEFPACKAGE "AFFI-TEST" (:use "COMMON-LISP" "AFFI"))
(IN-PACKAGE "AFFI-TEST")

;; SysBase is the conventional name for exec.library
;; It is only enforced by the file loaded by REQUIRE-LIBR
(eval-when (compile eval load)
  (declare-library-base :SysBase "exec.library")) ;keywor

;; using only MLIBCALL allows not to load definitions at 
(eval-when (compile eval)
  (require-library-functions "exec.library" :import '("Fi

(with-open-library ("exec.library")
  (print (mlibcall FindTask 0)))
```

This file can be used in interpreted and compiled mode. Compiled, it will have inlined the library function calls.

Example 32.13. Using flibcall

```
(DEFPACKAGE "AFFI-TEST" (:use "COMMON-LISP" "AFFI"))
(IN-PACKAGE "AFFI-TEST")

(eval-when (compile eval load)
  ;; keyword avoids name conflicts
  (declare-library-base :SysBase "exec.library"))

;; The load situation permits the use of flibcall
(eval-when (eval compile load)
  (require-library-functions "exec.library"))

(unless (open-library 'SysBase) (error "No library for Sy
```

```
(flibcall (if t 'FindTask 'Debug) 0)
(close-library 'SysBase)
```

Example 32.14. Be fully dynamic, defining library bases ourselves

```
(DEFPACKAGE "AFFI-TEST" (:use "COMMON-LISP" "AFFI"))
(IN-PACKAGE "AFFI-TEST")

(eval-when (compile eval load)
  (defvar mylib (declare-library-base :foobase "foo.libra
(eval-when (eval compile load)            ;eval allows mlib
  (defflibfun 'foo1 mylib -30 '#xA '* 'string)
  (defflibfun 'foo2 mylib -36 '#x21 0 * 4))

(defun foo (name)
  (when (open-library mylib)
    (list (mlibcall foo1 name) (flibcall 'foo2 name 12321
    (close-library mylib)))
```

Example 32.15. Some sample function definitions

```
(defflibfun 'FindTask 'SysBase -294 #xA '* 'string)
(eval-library-function FindTask "exec.library"
  (:offset -294)
  (:return-type *)
  (:arguments
   (name   string   :A1)))
(declare-library-function NameFromLock "dos.library"
  (:offset -402)
  (:return-type 0)
  (:arguments
   (lock   4   :D1)
   (buffer :io :D2)
   (len    4   :D3)))

(eval-when (compile eval)
  (defconstant GVF_LOCAL_ONLY (ash 1 9))
  (defflibfun 'SetVar 'DosBase -900 #x5432 0 'string 'str
(defun setvar (name value)
  (with-open-library (DosBase)
    ;; length of -1 means find length of NULL-terminated-
    (mlibcall SetVar name value -1 GVF_LOCAL_ONLY)))
```

32.5. Socket Streams

Platform Dependent: UNIX, Win32 platforms only.

32.5.1. Introduction

Sockets are used for interprocess communications by processes running on the same host as well as by processes running on different hosts over a computer network. The most common kind of sockets is Internet stream sockets, and a high-level interface to them is described here. A more low level interface that closely follows the C system calls is also available, see Section 33.17, "Raw Socket Access".

Two main varieties of sockets are interfaced to:

- "active" sockets correspond to `SOCKET:SOCKET-STREAM`s which are bidirectional `STREAM`s
- "passive" sockets correspond to `SOCKET:SOCKET-SERVER`s which are a special kind of objects that are used to allow the other side to initiate interaction with lisp.

Example 32.16. Lisp read-eval-print loop server

Here is a simple lisp read-eval-print loop server that waits for a remote connection and evaluates forms read from it:

```
(LET ((server (SOCKET:SOCKET-SERVER)))
  (FORMAT t "~&Waiting for a connection on ~S:~D~%"
          (SOCKET:SOCKET-SERVER-HOST server) (SOCKET:SOCKI
  (UNWIND-PROTECT
      ;; infinite loop, terminate with Control+C
      (LOOP (WITH-OPEN-STREAM (socket (SOCKET:SOCKET-ACCE
              (MULTIPLE-VALUE-BIND (local-host local-port
```

```
          (MULTIPLE-VALUE-BIND (remote-host remote-
            (FORMAT T "~&Connection: ~S:~D -- ~S:~D
                    remote-host remote-port local-h
        ;; loop is terminated when the remote host
        (LOOP (WHEN (EQ :eof (SOCKET:SOCKET-STATUS
              (PRINT (EVAL (READ socket)) socket)
              ;; flush everything left in socket
              (LOOP :for c = (READ-CHAR-NO-HANG soc
              (TERPRI socket))))
;; make sure server is closed
(SOCKET:SOCKET-SERVER-CLOSE server)))
```

This opens a gaping security hole!

Functions like `EXT:SHELL`, `EXT:EXECUTE`, `EXT:RUN-SHELL-COMMAND` will allow the remote host to execute arbitrary code with your permissions. While functions defined in lisp (like `EXT:RUN-SHELL-COMMAND`) can be removed (using `FMAKUNBOUND`), the built-in functions (like `EXT:SHELL` and `EXT:EXECUTE`) cannot be permanently removed from the run-time, and an experienced hacker will be able to invoke them even if you `FMAKUNBOUND` their names.

You should limit the socket server to local connections by passing string `"127.0.0.1"` as the `:INTERFACE` argument.

Example 32.17. Lisp HTTP client

Here are a couple of simple lisp **HTTP** clients that fetch a web page and a binary file, and upload a file:

```
(DEFUN wget-text (host page file &OPTIONAL (port 80))
  ;; HTTP requires the :DOS line terminator
  (WITH-OPEN-STREAM (socket (SOCKET:SOCKET-CONNECT port h
     (FORMAT socket "GET ~A HTTP/1.0~2%" page)
     ;; dump the whole thing - header+data - into the out
     (WITH-OPEN-FILE (out file :direction :output)
       (LOOP :for line = (READ-LINE socket nil nil) :whil
          :do (WRITE-LINE line out)))))
(DEFUN wget-binary (host page file &OPTIONAL (port 80))
```

```
  (WITH-OPEN-STREAM (socket (SOCKET:SOCKET-CONNECT port ho
    (FORMAT socket "GET ~A HTTP/1.0~2%" page)
    (LOOP :with content-length :for line = (READ-LINE soc
      ;; header is separated from the data with a blank li
      :until (ZEROP (LENGTH line)) :do
      (WHEN (STRING= line #1="Content-length: " :end1 #2=
        (SETQ content-length (PARSE-INTEGER line :start #2
      ;; this will not work if the server does not supply
      :finally (RETURN (LET ((data (MAKE-ARRAY content-len
                                               :element-ty
                         ;; switch to binary i/o on soc
                         (SETF (STREAM-ELEMENT-TYPE soc
                         ;; read the whole file in one
                         (EXT:READ-BYTE-SEQUENCE data so
                         (WITH-OPEN-FILE (out file :dire
                                              :ELEMENT-
                           ;; write the whole file in on
                           (EXT:WRITE-BYTE-SEQUENCE data
                         data))))))
(DEFUN wput (host page file &OPTIONAL (port 80))
  (WITH-OPEN-STREAM (socket (SOCKET:SOCKET-CONNECT port ho
    (WITH-OPEN-FILE (in file :direction :inptut :ELEMENT-
      (LET* ((length (FILE-LENGTH in))
             (data (MAKE-ARRAY length :element-type '(UNS
        ;; some servers may not understand the "Content-le
        (FORMAT socket "PUT ~A HTTP/1.0~%Content-length: ~
        (SETF (STREAM-ELEMENT-TYPE socket) '(UNSIGNED-BYTE
        (EXT:READ-BYTE-SEQUENCE data in)
        (EXT:WRITE-BYTE-SEQUENCE data socket)))
    ;; not necessary if the server understands the "Conten
    (SOCKET:SOCKET-STREAM-SHUTDOWN socket :output)
    ;; get the server response
    (LOOP :for line = (READ-LINE socket nil nil) :while li
```

32.5.2. Socket API Reference

(SOCKET:SOCKET-SERVER &OPTIONAL *port* &KEY :INTERFACE :BACKLOG)

This function creates a socket an binds a port to the socket. The server exists to watch for client connect attempts. The optional argument is the port to use (non-negative `FIXNUM`). The `:BACKLOG` parameter defines maximum length of queue of pending connections (see `listen`) and defaults to 1. The `:INTERFACE` is either a `STRING`, interpreted as the IP address that will be bound, or a socket, from whose peer the connections will be made. Default is (for backward compatibility) to bind to all local interfaces, but for security reasons it is advisable to bind to loopback `"127.0.0.1"` if you need only local connections.

(SOCKET:SOCKET-SERVER-CLOSE *socket-server*)

Closes down the server socket. Just like streams, `SOCKET:SOCKET-SERVER`s are closed at garbage-collection. You should not rely on this however, because garbage-collection times are not deterministic.

(SOCKET:SOCKET-SERVER-HOST *socket-server*)

(SOCKET:SOCKET-SERVER-PORT *socket-server*)

Returns the host mask indicating which hosts can connect to this server and the port which was bound using `SOCKET:SOCKET-SERVER`.

(SOCKET:SOCKET-WAIT *socket-server* &OPTIONAL [*seconds* [*microseconds*]])

Wait for a fixed time for a connection on the `socket-server` (a `SOCKET:SOCKET-SERVER`). Without a timeout argument, `SOCKET:SOCKET-WAIT` blocks indefinitely. When timeout is zero, poll. Returns `T` when a connection is available (i.e., `SOCKET:SOCKET-ACCEPT` will not block) and `NIL` on timeout.

(SOCKET:SOCKET-ACCEPT *socket-server* &KEY :ELEMENT-TYPE :EXTERNAL-FORMAT :BUFFERED :TIMEOUT)

Creates the server-side bidirectional `SOCKET:SOCKET-STREAM` for the connection. Waits for an attempt to connect to the server for no more than `:TIMEOUT` seconds (which may be a non-negative `REAL` or a list `(sec usec)` or a pair `(sec . usec)`). `SIGNAL`s an `ERROR` if no connection is made in that time.

(SOCKET:SOCKET-CONNECT *port* &OPTIONAL [*host*] &KEY :ELEMENT-TYPE :EXTERNAL-FORMAT :BUFFERED :TIMEOUT)

Attempts to create a client-side bidirectional `SOCKET:SOCKET-STREAM`. Blocks until the server accepts the connection, for no more than `:TIMEOUT` seconds. If it is 0, returns immediately and (probably) blocks on the next i/o operation (you can use `SOCKET:SOCKET-STATUS` to check whether it will actually block).

(SOCKET:SOCKET-STATUS *socket-stream-or-list* &OPTIONAL [*seconds* [*microseconds*]])

Checks whether it is possible to read from or write to a `SOCKET:SOCKET-STREAM` or whether a connection is available on a `SOCKET:SOCKET-SERVER` without blocking.

This is similar to `LISTEN`, which checks only one `STREAM` and only for input, and `SOCKET:SOCKET-WAIT`, which works only with `SOCKET:SOCKET-SERVER`s.
We define *`status`* for a `SOCKET:SOCKET-SERVER` or a `SOCKET:SOCKET-STREAM` to be `:ERROR` if any i/o operation will cause an `ERROR`.
Additionally, for a `SOCKET:SOCKET-SERVER`, we define *`status`* to be `T` if a connection is available, i.e., is `SOCKET:SOCKET-ACCEPT` will not block, and `NIL` otherwise.
Additionally, for a `SOCKET:SOCKET-STREAM`, we define *`status`* in the given *`direction`* (one of `:INPUT`, `:OUTPUT`, and `:IO`) to be

Possible status values for various directions:

`:INPUT` status:

`NIL`	reading will block
`:INPUT`	some input is available
`:EOF`	the stream has reached its end

`:OUTPUT` status:

`NIL`	writing will block
`:OUTPUT`	output to the stream will not block

`:IO` status:

output status	input status		
	`NIL`	**`:INPUT`**	**`:EOF`**
`NIL`	`NIL`	`:INPUT`	`:EOF`
`:OUTPUT`	`:OUTPUT`	`:IO`	`:APPEND`

Possible values of *`socket-stream-or-list`*:

`SOCKET:SOCKET-STREAM` or `SOCKET:SOCKET-SERVER`
: Returns the appropriate status, as defined above (`:IO` status for `SOCKET:SOCKET-STREAM`)

`(SOCKET:SOCKET-STREAM . direction)`
: Return the status in the specified direction

a non-empty list of the above
: Return a list of values, one for each element of the argument list (a la `MAPCAR`)

If you want to avoid consing[3] up a fresh list, you can make the elements of *`socket-stream-or-list`* to be `(socket-stream direction . x)` or `(socket-server . x)`. Then `SOCKET:SOCKET-STATUS` will destructively modify its argument and replace *`x`* or `NIL` with the status and return the modified list. You can pass this modified list to `SOCKET:SOCKET-STATUS` again.

The optional arguments specify the timeout. `NIL` means wait forever, `0` means poll.

The second value returned is the number of objects with non-`NIL` status, i.e., “actionable” objects. `SOCKET:SOCKET-STATUS` returns either due to a timeout or when this number is positive, i.e., if the timeout was `NIL` and `SOCKET:SOCKET-STATUS` did return, then the second value is positive (this is the reason `NIL` is **not** treated as an empty `LIST`, but as an invalid argument).

This is the interface to `select` (on some platforms, `poll`), so it will work on any **CLISP** `STREAM` which is based on a file descriptor, e.g., `EXT:*KEYBOARD-INPUT*` and file/pipe/socket `STREAM`s, as well as on raw sockets.

(SOCKET:SOCKET-STREAM-HOST *socket-stream*)
(SOCKET:SOCKET-STREAM-PORT *socket-stream*)

These two functions return information about the `SOCKET:SOCKET-STREAM`.

(SOCKET:SOCKET-STREAM-PEER *socket-stream* [*do-not-resolve-p*])

Given a `SOCKET:SOCKET-STREAM`, this function returns the name of the host on the opposite side of the connection and its port number; the server-side can use this to see who connected.

When the optional second argument is non-`NIL`, the hostname resolution is disabled and just the IP address is returned, without the FQDN.

The *`socket-stream`* argument can also be a raw socket.

(SOCKET:SOCKET-STREAM-LOCAL *socket-stream* [*do-not-resolve-p*])

The dual to `SOCKET:SOCKET-STREAM-PEER` - same information, host name and port number, but for the local host. The difference from `SOCKET:SOCKET-STREAM-HOST` and `SOCKET:SOCKET-STREAM-PORT` is that this function asks the OS (and thus returns the correct trusted values) while the other two are just accessors to the internal data structure, and basically return the arguments given to the function which created the *`socket-stream`*.

The *`socket-stream`* argument can also be a raw socket.

(SOCKET:SOCKET-STREAM-SHUTDOWN *socket-stream direction*)

Some protocols provide for closing the connection in one *`direction`* using `shutdown`. This function provides an interface to this **UNIX** system call. *`direction`* should be `:INPUT` or `:OUTPUT`. Note that you should still call `CLOSE` after you are done with your *socket-stream*; this is best accomplished by using WITH-OPEN-STREAM.

All SOCKET:SOCKET-STREAMs are bidirectional STREAMs (i.e., both INPUT-STREAM-P and OUTPUT-STREAM-P return T for them). SOCKET:SOCKET-STREAM-SHUTDOWN *breaks* this and turns its argument stream into an input STREAM (if *direction* is :OUTPUT) or output STREAM (if *direction* is :INPUT). Thus, the following important invariant is preserved: whenever

- a STREAM is open (i.e., OPEN-STREAM-P returns T) and
- a STREAM is an input STREAM (i.e., INPUT-STREAM-P returns T)

the STREAM can be read from (e.g., with READ-CHAR or READ-BYTE).

The *socket-stream* argument can also be a raw socket.

(SOCKET:SOCKET-OPTIONS *socket-server* &REST {*option*}*)

Query and, optionally, set socket options using getsockopt and setsockopt. An *option* is a keyword, optionally followed by the new value. When the new value is not supplied, setsockopt is not called. For each option the old (or current, if new value was not supplied) value is returned. E.g., (SOCKET:SOCKET-OPTIONS *socket-server* :SO-LINGER 1 :SO-RCVLOWAT) returns 2 values: NIL, the old value of the :SO-LINGER option, and 1, the current value of the :SO-RCVLOWAT option.

The *socket-stream* argument can also be a raw socket.

(SOCKET:STREAM-HANDLES *stream*)

Return the input and output OS file descriptors of the *stream* as multiple values. See Section 33.17, “Raw Socket Access”.

32.6. Quickstarting delivery with CLISP

This section describes three ways to turn **CLISP** programs into executable programs, which can be started as quickly as executables written in other languages.

32.6.1. Summary

UNIX
: CLISP can act as a script interpreter.

Desktop environments such as KDE, Gnome, Mac OS X or Win32.
: Files created with CLISP can be associated with the CLISP executable so that clicking on them would make CLISP execute the appropriate code.

Linux kernel with `CONFIG_BINFMT_MISC=y`
: Associate the extensions `#P".fas"` and `#P".lisp"` with CLISP; then you can make the files executable and run them from the command line.

Multi-file applications

These three techniques apply to a single `#P".lisp"` or `#P".fas"` file. If your application is made up of several `#P".lisp"` or `#P".fas"` files, you can simply concatenate them (using **cat**) into one file; the techniques then apply to that concatenated file.

Lisp-less target

These three techniques assume that the target machine has CLISP pre-installed and thus you can deliver just your own application, not CLISP itself. If you want to deliver applications without assuming anything about your target box, you have to resort to creating executable memory images.

32.6.2. Scripting with CLISP

Platform Dependent: UNIX platform only.

On UNIX, a text file (`#P".fas"` or `#P".lisp"`) can be made executable by adding a first line of the form

```
#!interpreter [interpreter-arguments]
```

and using **chmod** to make the file executable.

OS Requirements. **CLISP** can be used as a script interpreter under the following conditions:

- The `interpreter` must be the full pathname of **CLISP**. The recommended path is `/usr/local/bin/clisp`, and if **CLISP** is actually installed elsewhere, making `/usr/local/bin/clisp` be a symbolic link to the real **CLISP**.
- The `interpreter` must be a real executable, not a script. Unfortunately, in the binary distributions of **CLISP** on Solaris, **clisp** is a shell script because a **C** compiler cannot be assumed to be installed on this platform. If you do have a **C** compiler installed, build **CLISP** from the source yourself; **make install** will install **clisp** as a real executable.

- On some platforms, the first line which specifies the interpreter is limited in length:

  - max. 32 characters on SunOS 4,
  - max. 80 characters on HP-UX,
  - max. 127 characters on **Linux**.

  Characters exceeding this limit are simply cut off by the system. At least 128 characters are accepted on Solaris, IRIX, AIX, OSF/1. There is no workaround: You have to keep the interpreter pathname and arguments short.

- On Solaris and HP-UX, only the first `interpreter-arg` is passed to the `interpreter`. In order to pass more than one option (for example, `-M` and `-C`) to **CLISP**, separate them with no-break spaces instead of normal spaces. (But the separator between `interpreter` and `interpreter-arguments` must still be a normal space!) **CLISP** will split the `interpreter-arguments` both at no-break spaces and at normal spaces.

Script execution.

- The script should contain Lisp forms, except in the `#!` line.

- The file is loaded normally, through the function `LOAD` (in particular, the name of the script file, which is `$0` in **/bin/sh**, can be found in `*LOAD-TRUENAME*` and `*LOAD-PATHNAME*`).
- Before it is loaded, the variable `EXT:*ARGS*` is bound to a `LIST` of `STRING`s, representing the arguments given to the Lisp script (i.e., `$1` in **/bin/sh** becomes `(FIRST EXT:*ARGS*)` etc).
- The standard **UNIX** i/o facilities (see `<stdio.h>`) are used: `*STANDARD-INPUT*` is bound to `stdin`, `*STANDARD-OUTPUT*` to `stdout`, and `*ERROR-OUTPUT*` to `stderr`. Note Section 25.2.13.1, "Scripting and `DRIBBLE`".
- The continuable `ERROR`s will be turned into `WARNING`s (using `EXT:APPEASE-CERRORS`).
- Non-continuable `ERROR`s and **Control**+C interrupts will terminate the execution of the Lisp script with an error status (using `EXT:EXIT-ON-ERROR`).
- If you wish the script's contents to be compiled during loading, add `-c` to the *`interpreter-arguments`*.

See also the manual.

If nothing works. Another, quite inferior, alternative is to put the following into a file:

```
#!/bin/sh
exec clisp <<EOF
(lisp-form)
(another-lisp-form)
(yet-another-lisp-form)
EOF
```

The problem with this approach is that the return values of each form will be printed to `*STANDARD-OUTPUT*`. Another problem is that no user input will be available.

32.6.3. Desktop Environments

Platform Dependent: Win32, Gnome, KDE, Mac OS X desktop platforms only.

Notations

Although we use **Win32**-specific notation, these techniques work on other desktop environments as well.

There are two different ways to make **CLISP** “executables” on desktop platforms.

1. Associate the `#P".mem"` extension with **c:\clisp\clisp.exe `-M` `"%s"`**.
2. Associate the `#P".fas"` extension with **c:\clisp\clisp.exe `-i` `"%s"`**
   Alternatively, you may want to have a function `main` in your `#P".fas"` files and associate the `#P".fas"` extension with **c:\clisp\clisp.exe `-i` %s `-x` (main)**.

Then clicking on the compiled lisp file (with `#P".fas"` extension) will load the file (thus executing all the code in the file), while the clicking on a **CLISP** memory image (with `#P".mem"` extension) will start **CLISP** with the given memory image.

Note

On **Win32**, **CLISP** is distributed with a file `src/install.bat`, which runs `src/install.lisp` to create a file `clisp.lnk` on your desktop and also associates `#P".fas"`, `#P".lisp"`, and `#P".mem"` files with **CLISP**.

32.6.4. Associating extensions with CLISP via kernel

Platform Dependent: Linux platforms only.

You have to build your kernel with `CONFIG_BINFMT_MISC=y` and `CONFIG_PROC_FS=y`. Then you will have a `/proc/sys/fs/binfmt_misc/` directory and you will be able to do (as `root`; you might want to put these lines into `/etc/rc.d/rc.local`):

```
# echo ":CLISP:E::fas::/usr/local/bin/clisp:" >> /proc/sy
# echo ":CLISP:E::lisp::/usr/local/bin/clisp:" >> /proc/s
```

Then you can do the following:

```
$ cat << EOF > hello.lisp
(print "hello, world!")
EOF
$ clisp -c hello.lisp
;; Compiling file hello.lisp ...
;; Wrote file hello.lisp
0 errors, 0 warnings
$ chmod +x hello.fas
$ hello.fas

"hello, world!"
$
```

Please read `/usr/src/linux/Documentation/binfmt_misc.txt` for details.

32.7. Shell, Pipes and Printing

This section describes how CLISP can invoke external executables and communicate with the resulting processes.

32.7.1. Shell

Platform Dependent: UNIX platform only.

`(EXT:EXECUTE program arg1 arg2 ...)` executes an external program. Its name is *program* (a full pathname). It is given the `STRING`s *arg1*, *arg2*, ... as arguments.

Platform Dependent: UNIX, Win32 platforms only.

`(EXT:SHELL [command])` calls the operating system's shell, the value of the environment variable `SHELL` on **UNIX** and `COMSPEC` on **Win32**. `(EXT:SHELL)` calls the shell for interactive use. `(EXT:SHELL command)` calls the shell only for execution of the one given *command*.

Platform Dependent: UNIX, Win32 platforms only.

The functions `EXT:RUN-SHELL-COMMAND` and `EXT:RUN-PROGRAM` are the general interface to `EXT:SHELL` and the above:

`(EXT:RUN-SHELL-COMMAND command &KEY :MAY-EXEC :INDIRECTP :INPUT :OUTPUT :IF-OUTPUT-EXISTS :WAIT)` runs a shell command (including shell built-in commands, like **DIR** on **Win32** and **for/do/done** on **UNIX**).

`(EXT:RUN-PROGRAM program &KEY :MAY-EXEC :INDIRECTP :ARGUMENTS :INPUT :OUTPUT :IF-OUTPUT-EXISTS :WAIT)` runs an external program.

command

the shell command.

Platform Dependent: UNIX platform only.

The shell the command is passed to is the value of the environment variable `SHELL`, which normally is **/bin/sh**. The command should be a “simple command”; a “command list” should be enclosed in "{ ... ; }" (for **/bin/sh**) or "(...)" (for `/bin/csh`).

program

the program. The directories listed in the environment variable `PATH` will be searched for it.

`:ARGUMENTS`

a list of arguments (`STRING`s) that are given to the program.

`:INPUT`

where the program's input is to come from: either `:TERMINAL` (`stdin`, the default) or `:STREAM` (a Lisp `STREAM` to be created) or a pathname designator (an input file) or `NIL` (no input at all).

`:OUTPUT`
: where the program's output is to be sent to: either `:TERMINAL` (`stdout`, the default) or `:STREAM` (a Lisp `STREAM` to be created) or a pathname designator (an output file) or `NIL` (ignore the output).

`:IF-OUTPUT-EXISTS`
: what to do if the `:OUTPUT` file already exists. The possible values are `:OVERWRITE`, `:APPEND`, `:ERROR`, with the same meaning as for `OPEN`. The default is `:OVERWRITE`.

`:WAIT`
: whether to wait for program termination or not (this is useful when no i/o to the process is needed); the default is `T`, i.e., synchronous execution.

`:MAY-EXEC`
: pass **exec** to the underlying shell (**UNIX** only).

`:INDIRECTP`
: use a shell to run the command, e.g., `(EXT:RUN-PROGRAM "dir" :indirectp T)` will run the shell built-in command **DIR**. This argument defaults to `T` for `EXT:RUN-SHELL-COMMAND` and to `NIL` for `EXT:RUN-PROGRAM`. (**Win32** only).

If `:STREAM` was specified for `:INPUT` or `:OUTPUT`, a Lisp `STREAM` is returned. If `:STREAM` was specified for both `:INPUT` and `:OUTPUT`, three Lisp `STREAM`s are returned, as for the function `EXT:MAKE-PIPE-IO-STREAM`. Otherwise, the return value depends on the process termination status: if it ended normally (without signal, core-dump etc), its exit status is returned as an `INTEGER`, otherwise `NIL` is returned.
This use of `EXT:RUN-PROGRAM` can cause deadlocks, see `EXT:MAKE-PIPE-IO-STREAM`.

32.7.2. Pipes

Platform Dependent: UNIX, Win32 platforms only.

`(EXT:MAKE-PIPE-INPUT-STREAM` *`command`* `&KEY :ELEMENT-TYPE :EXTERNAL-FORMAT :BUFFERED)`

returns an input STREAM that will supply the output from the execution of the given operating system command.

(EXT:MAKE-PIPE-OUTPUT-STREAM *command* &KEY :ELEMENT-TYPE :EXTERNAL-FORMAT :BUFFERED)

returns an output STREAM that will pass its output as input to the execution of the given operating system command.

(EXT:MAKE-PIPE-IO-STREAM *command* &KEY :ELEMENT-TYPE :EXTERNAL-FORMAT :BUFFERED)

returns three values. The primary value is a bidirectional STREAM that will simultaneously pass its output as input to the execution of the given operating system command and supply the output from this command as input. The second and third value are the input STREAM and the output STREAM that make up the bidirectional STREAM, respectively.

Warning

These three streams must be closed individually, see CLOSE-CONSTRUCTED-STREAM:ARGUMENT-STREAM-ONLY.

Warning

Improper use of this function can lead to *deadlocks*. Use it at your own risk!
A deadlock occurs if the command and your Lisp program either both try to read from each other at the same time or both try to write to each other at the same time.
To avoid deadlocks, it is recommended that you fix a protocol between the command and your program and avoid any hidden buffering: use READ-CHAR, READ-CHAR-NO-HANG, LISTEN, SOCKET:SOCKET-STATUS instead of READ-LINE and READ on the input side, and complete every output operation by a FINISH-OUTPUT. The same precautions must apply to the called command as well.

32.7.3. Printing

The macro **`EXT:WITH-OUTPUT-TO-PRINTER`**:

```
(EXT:WITH-OUTPUT-TO-PRINTER (variable [:EXTERNAL-FORMAT])
  {declaration}*
  {form}*)
```

binds the variable `variable` to an output `STREAM` that sends its output to the printer.

32.8. Operating System Environment

Most modern operating systems support environment variables that associate strings ("variables") with other strings ("values"). These variables are somewhat similar to the `SPECIAL` variables in **Common Lisp**: their values are inherited by the processes from their parent process.

You can access your OS environment variables using the function `(EXT:GETENV &OPTIONAL string)`, where `string` is the name of the environment variable. When `string` is omitted or `NIL`, all the environment variables and their values are returned in an association list.

You can change the value of existing environment variables or create new ones using `(SETF (EXT:GETENV string) new-value)`.

Chapter 33. Extensions Implemented as Modules

Table of Contents

33.1. System Calls

The **"POSIX"** module makes some system calls available from lisp. Not all of these system calls are actually POSIX, so this package has a nickname **"OS"**.

This module is present in the **base** linking set by default.

When this module is present, `*FEATURES*` contains the symbol `:SYSCALLS`.

`(POSIX:RESOLVE-HOST-IPADDR &OPTIONAL host)`
: Returns the HOSTENT structure:

  name
  : host name

  aliases
  : `LIST` of aliases

  addr-list
  : `LIST` of IP addresses as dotted quads (for IPv4) or coloned octets (for IPv6)

  addrtype
  : `INTEGER` address type (IPv4 or IPv6)

  When *`host`* is omitted or `:DEFAULT`, return the data for the current host. When *`host`* is `NIL`, all the host database is returned as a list (this would be the contents of the `/etc/hosts` file on a **UNIX** system or `${windir}/system32/etc/hosts` on a **Win32** system). This is an interface to `gethostent`, `gethostbyname`, and `gethostbyaddr`.

`(OS:SERVICE &OPTIONAL service-name protocol)`
: A convenience function for looking up a port given the service name, such as "WWW" or "FTP". It returns the SERVICE structure (name, list of aliases, port, protocol) for the given *`service-name`* and *`protocol`*, or all services as a `LIST` if *`service-name`* is missing or `NIL`.

`(POSIX:FILE-STAT pathname &OPTIONAL link-p)`
: Return the FILE-STAT structure. *`pathname`* can be a `STREAM`, a `PATHNAME`, a `STRING` or a `NUMBER` (on a **UNIX** system, meaning file descriptor). The first slot of the structure returned is the string or the

number on which `stat`, `fstat`, or `lstat` was called. The other slots are numbers, members of the struct stat:

`dev`
: Device ID of device containing file.

`ino`
: File serial number.

`mode`
: Mode of file.

`nlink`
: Number of hard links to the file.

`uid`
: User ID of file.

`gid`
: Group ID of file.

`rdev`
: Device ID (if file is character or block special).

`size`
: For regular files, the file size in bytes. For symbolic links, the length in bytes of the pathname contained in the symbolic link. For a shared memory object, the length in bytes. For a typed memory object, the length in bytes. For other file types, the use of this field is unspecified.

`atime`
: universal time of last access.

`mtime`
: universal time of last data modification.

`ctime`
: universal time of last status change (on **Win32** - creation time).

`blksize`
: A file system-specific preferred I/O block size for this object. In some file system types, this may vary from file to file.

`blocks`
: Number of blocks allocated for this object.

All slots are read-only.
If the system does not support a particular field (e.g., **Win32** prior to 2000 does not have hard links), `NIL` (or the default, like 1 for the number of hard links for old **Win32**) is returned.

Win32 platform only.

Normally, one would expect `(POSIX:FILE-STAT "foo")` and `(POSIX:FILE-STAT (OPEN "foo"))` to return “similar” objects (`OPEN`ing a file changes its access time though). This is **not** the case on **Win32**, where `stat` works but `fstat` does **not**. Specifically, `fstat` requires an int argument of an unknown nature, and it is not clear how do deduce it from the **Win32** file handle. Therefore, instead of always failing on open `FILE-STREAM` arguments, this function calls `GetFileInformationByHandle` and fills the FILE-STAT return value based on that.

`(POSIX:SET-FILE-STAT pathname &KEY :ATIME :MTIME :MODE :UID :GID)`

Set some file attributes using `chmod`, `chown`, and `utime`.

`(POSIX:STAT-VFS pathname)`

Return a STAT-VFS structure. *`pathname`* can be a `STREAM`, a `PATHNAME`, a `STRING` or a `NUMBER` (on a **UNIX** system, meaning file descriptor). The first slot of the structure returned is the string or the number on which `statvfs` or `fstatvfs` was called. The other slots are members of the struct statvfs:

bsize
: File system block size.

frsize
: Fundamental file system block size.

blocks
: Total number of blocks on file system in units of *`frsize`*.

bfree
: Total number of free blocks.

bavail
: Number of free blocks available to non-privileged processes.

files
: Total number of file serial numbers.

ffree
: Total number of free file serial numbers.

favail
: Number of file serial numbers available to non-privileged processes.

fsid

File system ID.

`flag`
: List of platform-dependent values, such as `:READ-ONLY`.

`namemax`
: Maximum filename length.

`vol-name`
: Volume name (**Win32** only).

`fs-type`
: File system type (**Win32** only).

All slots are read-only.

`(OS:FILE-INFO` ***`pathname`*** **`&OPTIONAL all)`**
: Return the FILE-INFO structure. *`pathname`* should be a pathname designator. The 7 slots are

`attributes`
`ctime`
`atime`
`wtime`
`size`
`name`
`name-short`

When *`pathname`* is wild, returns just the first match, unless the second (optional) argument is non-`NIL`, in which case a `LIST` of objects is returned, one for each match.

`(POSIX:USER-INFO &OPTIONAL` ***`user`*** **`)`**
: Return the USER-INFO structure (name, encoded password, UID, GID, full name, home directory, shell). *`user`* should be a `STRING` (`getpwnam` is used) or an `INTEGER` (`getpwuid` is used). When *`user`* is missing or `NIL`, return all users (using `getpwent`). When *`user`* is `:DEFAULT`, return the information about the current user (using `getlogin` or `getuid`).

Platform Dependent: **UNIX** platform only.

`(POSIX:GROUP-INFO &OPTIONAL` ***`group`*** **`)`**
: Return the GROUP-INFO structure (name, GID, member `LIST`). *`group`* should be a `STRING` (`getgrnam` is used) or an `INTEGER` (`getgrgid` is used). When *`group`* is missing or `NIL`, return all groups (using `getgrent`).

Platform Dependent: **UNIX** platform only.

`(POSIX:UNAME)`
: Return a structure describing the OS, derived from `uname`.

(POSIX:SYSCONF &OPTIONAL *what*)
(POSIX:CONFSTR &OPTIONAL *what*)

Return the specified configuration parameter or a property list of all available parameters (when *what* is missing or `NIL`), by calling `sysconf` and `confstr` respectively.

(POSIX:PATHCONF *pathname* &OPTIONAL *what*)

Return the specified configuration parameter or a property list of all available parameters (when *what* is missing or `NIL`), by calling `fpathconf` on open file streams and `pathconf` on all other pathname designators.

(POSIX:RLIMIT &OPTIONAL *what*)

Return the current and the maximal limits as two values when *what* is specified or the association list of all available limits (as an RLIMIT structure) when *what* is missing or `NIL`, by calling `getrlimit`.

(SETF (POSIX:RLIMIT *what*) (VALUES cur max))
(SETF (POSIX:RLIMIT *what*) rlimit)
(SETF (POSIX:RLIMIT) rlimit-alist)

Set the limits using `setrlimit`.

1. In the first form, *cur* and *max* are numbers (or `NIL` for `RLIM_INFINITY`).
2. In the second form, *rlimit* is an RLIMIT structure.
3. In the third form, *rlimit-alist* is an association list, as returned by `(POSIX:RLIMIT)`.

(POSIX:USAGE)

Return 2 structures describing the resource usage by the lisp process and its children, using `getrusage`.

(POSIX:ERF *real*)
(POSIX:ERFC *real*)
(POSIX:J0 *real*)
(POSIX:J1 *real*)
(POSIX:JN *integer real*)
(POSIX:Y0 *real*)
(POSIX:Y1 *real*)
(POSIX:YN *integer real*)
(POSIX:GAMMA *real*)
(POSIX:LGAMMA *real*)

Compute the error functions, Bessel functions and Gamma. These functions are required by the POSIX standard and should be available in `libm.so`.

Warning

Please note that these functions do not provide lisp-style error handling and precision, and do all the computations at the `DOUBLE-FLOAT` level.

`(POSIX:BOGOMIPS)`
: Compute the BogoMips rating.

`(POSIX:LOADAVG &OPTIONAL percentp)`
: Return 1, 5, and 15 minute system load averages, retrieved by `getloadavg`. If the argument is specified and non-`NIL`, the values are returned as integer percentiles.

`(POSIX:STREAM-LOCK` *`stream lock-p`* `&KEY (:BLOCK T) (:SHARED NIL) (:START 0) (:END NIL))`
: Set or remove a file lock for the (portion of the) file associated with *`stream`*, depending on *`lock-p`*. When *`block`* is `NIL`, the call is non-blocking, and when locking fails, it returns `NIL`. When *`shared`* is non-`NIL`, then lock can be shared between several callers. Several processes can set a *`shared`* (i.e., *read*) lock, but only one can set an *`exclusive`* (i.e., *write*, or non-*`shared`*) lock. Uses `fcntl` or `LockFileEx`.

Warning

UNIX and **Win32** differ on locking 0-length files: on **Win32**, two processes can have exclusive locks on it!

Warning

Win32 locks are *mandatory*: if you lock a file, others will not be able to open it! **UNIX** locks are usually *advisory*: a process is free to ignore it, but on some **UNIX** systems one can mount some file system with *mandatory* locks.

`(POSIX:WITH-STREAM-LOCK (`*`stream`* `&REST` *`options`*`) &BODY` *`body`*`)`
: Lock the *`stream`*, execute the *`body`*, unlock the *`stream`*. Pass *`options`* to `POSIX:STREAM-LOCK`.

`(POSIX:STREAM-OPTIONS` *`stream command`* `&OPTIONAL` *`value`*`)`
: Call `fcntl`, *`command`* can be `:FD` or `:FL`.

(POSIX:MKNOD *pathname type mode*)
Create a special file using `mknod`. Use `:FIFO` to create pipes and `:SOCK` to create sockets.

(POSIX:CONVERT-MODE *mode*)
Convert between numeric, (e.g., `0644`) and symbolic (e.g., `(:RUSR :WUSR :RGRP :ROTH)`) file modes.

(UMASK *mode*)
Change process mask using `umask`.

(POSIX:COPY-FILE *source destination* &KEY :METHOD :PRESERVE :IF-EXISTS :IF-DOES-NOT-EXIST)
This is an interface to `symlink` (when *`method`* is `:SYMLINK`), `link` (when it is `:HARDLINK`), and `rename` (when it is `:RENAME`) system calls, as well as, you guessed it, a generic file copy utility (when *`method`* is `:COPY`).
Both *`source`* and *`destination`* may be wild, in which case `TRANSLATE-PATHNAME` is used.

(POSIX:DUPLICATE-HANDLE *fd1* &OPTIONAL *fd2*)
This is an interface to the `dup` system calls on **UNIX** systems and to `DuplicateHandle` system call on **Win32**.

(OS:SHORTCUT-INFO *pathname*)
Return information about a **Win32** shortcut (`#P".lnk"`) file contents in a SHORTCUT-INFO structure.

(OS:MAKE-SHORTCUT *pathname* &KEY :WORKING-DIRECTORY :ARGUMENTS :SHOW-COMMAND :ICON :DESCRIPTION :HOT-KEY :PATH)
Create (or modify the properties of an existing one) a **Win32** shortcut (`#P".lnk"`) file.

(OS:SYSTEM-INFO)
Return **Win32** system information in a SYSTEM-INFO structure.

(OS:VERSION)
Return **Win32** version information in a VERSION structure.

(OS:MEMORY-STATUS)
Return **Win32** memory status information in a MEMORY-STATUS structure.

(OS:FILE-PROPERTIES filename set &KEY :INITID &ALLOW-OTHER-KEYS)
Wrapper for the **Win32** `IPropertyStorage` functionality.

`filename`
name of a compound file (where properties are stored) or (on NTFS) name of any file (properties are stored in the filesystem). For compound files on NTFS, file storage is preferred.

set
: property set, either `:BUILT-IN` or `:USER-DEFINED`

`:INITID` ***init-id***
: set the *`init-id`*

specifier value

specifier
: the property specifier: an `INTEGER`, `KEYWORD`, `STRING` or a `LIST` of an `INTEGER` or a `KEYWORD` and a `STRING`.

INTEGER
: a property identifier

KEYWORD
: Predefined `KEYWORD` IDs are

:APPNAME	:CREATE-DTM	:LASTPRINTED	:SUBJECT
:AUTHOR	:DOC-SECURITY	:LASTSAVE-DTM	:TEMPLATE
:CHARCOUNT	:EDITTIME	:LOCALE	:THUMBNAI
:CODEPAGE	:KEYWORDS	:PAGECOUNT	:TITLE
:COMMENTS	:LASTAUTHOR	:REVNUMBER	:WORDCOUN

STRING
: string property specifier. If no match is found, the first ID >= init-id (which defaults to 2) is associated with the string and its value is replaced with new value.

(INTEGER|KEYWORD STRING)
: the first element is used as a specifier, the string is associated with this ID.

value
: the new value of the property, a suitable Lisp object, `NIL` or a `LIST` of a `KEYWORD` and the value itself. If *`value`* is `NIL`, no assignment is done. `:EMPTY` and `:NULL` correspond to the VT_EMPTY and VT_NULL data types. `KEYWORD` in the `LIST` specifies the desired type of the property being set. Supported types are

:BOOL	:I1	:LPWSTR	:UI4
:BSTR	:I2	:R4	:UI8
:DATE	:I4	:R8	:UINT
:ERROR	:I8	:UI1	
:FILETIME	:LPSTR	:UI2	

FILETIMEs are converted to/from the universal time format, while DATEs are not.

Returns the property contents before assignment as multiple values.

(POSIX:CRYPT *key salt*)

Call `crypt`, arguments are `STRING`s.

(POSIX:ENCRYPT *block decrypt-p*)

(POSIX:SETKEY *key*)

Call `encrypt` and `setkey`, respectively. *`block`* and *`key`* are of type `(VECTOR (UNSIGNED-BYTE 8) 8)`. *`decrypt-p`* is `BOOLEAN`.

(OS:PHYSICAL-MEMORY)

Return 2 values: total and available physical memory.

Platform Dependent: **UNIX**, **Win32** platforms only.

(OS:FILE-OWNER *filename*)

Return the owner of the file.

Platform Dependent: **UNIX**, **Win32** platforms only.

(OS:PRIORITY *pid* &OPTIONAL *what*)

Return the process priority, platform-dependent `INTEGER` or platform -independent `SYMBOL`, one of

`:REALTIME`	`:NORMAL`	`:IDLE`
`:HIGH`	`:BELOW-NORMAL`	
`:ABOVE-NORMAL`	`:LOW`	

On **UNIX** calls `getpriority`, on **Win32** calls `GetPriorityClass`. `SETF`able using `setpriority` and `SetPriorityClass`.

(OS:PROCESS-ID)

Return the process ID (on **UNIX** calls `getpid`, on **Win32** calls `GetCurrentProcessId`)

(POSIX:OPENLOG ident
&KEY :PID :CONS :NDELAY :ODELAY :NOWAIT :FACILITY)

calls `openlog`

(POSIX:SETLOGMASK maskpri)

calls `setlogmask`

(POSIX:SYSLOG severity facility *format-string* &REST *arguments*)

calls `syslog` on `(APPLY FORMAT NIL` *`format-string arguments`*`)`

No `%` conversion is performed, you must do all formatting in Lisp.

(POSIX:CLOSELOG)

calls `closelog`

(POSIX:KILL *pid signal*)

calls `kill`

(POSIX:GETPGRP *pid*)

calls `getpgrp`

`(POSIX:SETPGRP)`
calls `setpgrp`; on non-POSIX systems where it requires 2 arguments (legacy BSD-style), it is called as `setpgrp(0,0)`

`(POSIX:GETSID pid)`
calls `getsid`

`(POSIX:SETSID)`
calls `setsid`

`(POSIX:SETPGID pid pgid)`
calls `setpgid`

`(POSIX:ENDUTXENT)`
calls `endutxent`

`(POSIX:GETUTXENT &OPTIONAL utmpx)`
calls `getutxent`, returns a `STRUCTURE-OBJECT` of type POSIX:UTMPX, which can be passed to subsequent calls to this function and re-used.

`(POSIX:GETUTXID id)`
calls `getutxid`, the argument is filled and returned

`(POSIX:GETUTXLINE line)`
calls `getutxline`, the argument is filled and returned

`(POSIX:PUTUTXLINE utmpx)`
calls `pututxline`, the argument is filled and returned

`(POSIX:SETUTXENT)`
calls `setutxent`

`(POSIX:GETUID)`
`(SETF (POSIX:GETUID) uid)`
Call `getuid` and `setuid`.

`(POSIX:GETGID)`
`(SETF (POSIX:GETGID) gid)`
Call `getgid` and `setgid`.

`(POSIX:GETEUID)`
`(SETF (POSIX:GETEUID) uid)`
Call `geteuid` and `seteuid`.

`(POSIX:GETEGID)`
`(SETF (POSIX:GETEGID) gid)`
Call `getegid` and `setegid`.

`(OS:STRING-TIME format-string &OPTIONAL object timezone)`
When *`object`* is a `STRING`, is is parsed according to *`format-string`* by `strptime`. When it is an `INTEGER`, it is formatted according to *`format-string`* by `strftime`. *`object`* defaults to `(GET-UNIVERSAL-TIME)`.

(POSIX:MKSTEMP *filename* &KEY :DIRECTION :ELEMENT-TYPE :EXTERNAL-FORMAT :BUFFERED)
: calls `mkstemp`, returns a `FILE-STREAM`.
`:DIRECTION` should allow output.
When `mkstemp` is missing, use `tempnam`. On **Win32** use `GetTempFileName`.

(POSIX:MKDTEMP *filename*)
: calls `mkdtemp` (similar to `mkstemp` but not in POSIX), returns the namestring of a new empty temporary directory.

(POSIX:SYNC &OPTIONAL *stream*)
: calls `fsync` (`FlushFileBuffers` on **Win32**) on the file descriptor associated with *`stream`*, or `sync` when *`stream`* is not supplied

(POSIX:MAKE-XTERM-IO-STREAM &KEY title)
: When running under the X Window System, you can create a bidirectional `STREAM`, which uses a new dedicated xterm, using the function `POSIX:MAKE-XTERM-IO-STREAM`:

```
(SETQ *ERROR-OUTPUT*
      (SETQ *DEBUG-IO*
            (POSIX:MAKE-XTERM-IO-STREAM :title "clisp
```

Platform Dependent: **UNIX** platform only.

(POSIX:FFS *n*)
: Find the first bit set. Like `ffs`, but implemented in Lisp and supports `BIGNUM`s.

33.2. Internationalization of User Programs

33.2.1. The GNU gettext

GNU gettext is a set of functions, included in **CLISP** or the **C** library, which permit looking up translations of strings through message catalogs. It is also a set of tools which makes the translation maintenance easy for the translator and the program maintainer.

The GNU gettext functions are available in **CLISP** in the **"I18N"** package, which is `EXT:RE-EXPORT`ed from the **"EXT"** package.

This module is present in the **base** linking set by default.

When this module is present, `*FEATURES*` contains the symbol `:I18N`.

(`I18N:GETTEXT` *`MSGID`* `&OPTIONAL` *`DOMAIN`* *`CATEGORY`*)
: returns the translation of the message *`MSGID`*, in the given *`DOMAIN`*, depending on the given *`CATEGORY`*. *`MSGID`* should be an **ASCII** string, and is normally the English message.

(`I18N:NGETTEXT` *`MSGID`* *`msgid_plural`* *`n`* `&OPTIONAL` *`DOMAIN`* *`CATEGORY`*)
: returns the plural form of the translation for of *`MSGID`* and *`n`* in the given *`DOMAIN`*, depending on the given *`CATEGORY`*. *`MSGID`* and *`msgid_plural`* should be **ASCII** strings, and are normally the English singular and English plural variant of the message, respectively.

33.2.1.1. Domain

The *`DOMAIN`* is a string identifier denoting the program that is requesting the translation. The pathname of the message catalog depends on the *`DOMAIN`*: usually it is located at `TEXTDOMAINDIR/l/LC_MESSAGES/domain.mo`, where *`l`* is the ISO 639-2 code of the language. The notion of *`DOMAIN`* allows several Lisp programs running in the same image to request translations independently of each other.

Function `I18N:TEXTDOMAIN`. `(I18N:TEXTDOMAIN)` is a place that returns the default *`DOMAIN`*, used when no *`DOMAIN`* argument is passed to the `I18N:GETTEXT` and `I18N:NGETTEXT` functions. It is `SETF`able. `(SETF I18N:TEXTDOMAIN)` is usually used during the startup phase of a program. Note that the default *`DOMAIN`* is not saved in a memory image. The use of `(SETF I18N:TEXTDOMAIN)` is recommended only for programs that are so simple that they will never need more than one *`DOMAIN`*.

Function `I18N:TEXTDOMAINDIR`. `(I18N:TEXTDOMAINDIR DOMAIN)` is a place that returns the base directory, called `TEXTDOMAINDIR` above, where the message catalogs for the given *`DOMAIN`* are assumed to be installed. It is `SETF`able. `(SETF I18N:TEXTDOMAINDIR)` is usually used during the startup phase of a program, and should be used because only the program knows where its message catalogs are installed. Note that the `TEXTDOMAINDIR`s are not saved in a memory image.

33.2.1.2. Category

The *`CATEGORY`* argument of the `I18N:GETTEXT` and `I18N:NGETTEXT` functions denotes which *`LOCALE`* facet the result should depend on. The possible values are a platform-dependent subset of `:LC_ADDRESS`, `:LC_ALL`, `:LC_COLLATE`, `:LC_CTYPE`, `:LC_IDENTIFICATION`, `:LC_MEASUREMENT`, `:LC_MESSAGES`, `:LC_MONETARY`, `:LC_NAME`, `:LC_NUMERIC`, `:LC_PAPER`, `:LC_TELEPHONE`, `:LC_TIME` The use of these values is useful for users who have a character/time/collation/money handling set differently from the usual message handling. Note that when a *`CATEGORY`* argument is used, the message catalog location depends on the *`CATEGORY`*: it will be expected at `TEXTDOMAINDIR/ll/category/domain.mo`.

33.2.1.3. Internationalization Example

A non-internationalized program simulating a restaurant dialogue might look as follows.

prog.lisp.

```
(setq n (parse-integer (first EXT:*ARGS*)))

(format t "~A~%" "'Your command, please?', asked the waite

(format t "~@?~%"
          (if (= n 1) "a piece of cake" "~D pieces of cake
          n)
```

After being internationalized, all strings are wrapped in I18N:GETTEXT calls, and I18N:NGETTEXT is used for plurals. Also, I18N:TEXTDOMAINDIR is assigned a value; in our case, for simplicity, the current directory.

prog.lisp.

```
(setf (textdomain) "prog")
(setf (textdomaindir "prog") "./")

(setq n (parse-integer (first EXT:*ARGS*)))

(format t "~A~%"
          (gettext "'Your command, please?', asked the wa

(format t "~@?~%"
          (ngettext "a piece of cake" "~D pieces of cake"
          n)
```

For ease of reading, it is customary to define an abbreviation for the I18N:GETTEXT function. An underscore is customary.

prog.lisp.

```
(setf (textdomaindir "prog") "./")
(defun _ (msgid) (gettext msgid "prog"))

(setq n (parse-integer (first EXT:*ARGS*)))

(format t "~A~%"
          (_"'Your command, please?', asked the waiter.")

(format t "~@?~%"
          (ngettext "a piece of cake" "~D pieces of cake"
          n)
```

Now the program's maintainer creates a message catalog template through the command

```
bash$ xgettext -o prog.pot prog.lisp
```

Note

xgettext version 0.11 or higher is required here.

The message catalog template looks roughly like this.

prog.pot.

```
msgid "'Your command, please?', asked the waiter."
msgstr ""

msgid "a piece of cake"
msgid_plural "%d pieces of cake"
msgstr[0] ""
msgstr[1] ""
```

Then a French translator creates a French message catalog

prog.fr.po.

```
msgid ""
msgstr ""
"Content-Type: text/plain; charset=ISO-8859-1\n"
"Plural-Forms: nplurals=2; plural=(n > 1);\n"

msgid "'Your command, please?', asked the waiter."
msgstr "«Votre commande, s'il vous plait», dit le garçon.'

# Les gateaux allemands sont les meilleurs du monde.
msgid "a piece of cake"
msgid_plural "%d pieces of cake"
msgstr[0] "un morceau de gateau"
msgstr[1] "%d morceaux de gateau"
```

and sends it to the program's maintainer.

The program's maintainer compiles the catalog as follows:

```
bash$ mkdir -p ./fr/LC_MESSAGES
bash$ msgfmt -o ./fr/LC_MESSAGES/prog.mo prog.fr.po
```

When a user in a french *LOCALE* then runs the program

```
bash$ clisp prog.lisp 2
```

she will get the output

```
    «Votre commande, s'il vous plait», dit le garçon.
    2 morceaux de gateau
```

33.2.2. Locale

(I18N:SET-LOCALE &OPTIONAL *CATEGORY* *LOCALE*)
: This is an interface to `setlocale`.
When *LOCALE* is missing or `NIL`, return the current one.
When *CATEGORY* is missing or `NIL`, return all categories as a `LIST`.

(I18N:LOCALE-CONV)
: This is an interface to `localeconv`.
Returns a I18N:LOCALE-CONV structure.

(I18N:LANGUAGE-INFORMATION &OPTIONAL *item*)
: This is an interface to `nl_langinfo` (**UNIX**) and `GetLocaleInfo` (**Win32**).
When *item* is missing or `NIL`, return all available information as a `LIST`.

33.3. POSIX Regular Expressions

The **"REGEXP"** module implements the POSIX regular expressions by calling the standard **C** system facilities. The syntax of these regular expressions is described in many places, such as your local `<regex.h>` manual and Emacs info pages.

This module is present in the **base** linking set by default.

When this module is present, `*FEATURES*` contains the symbol `:REGEXP`.

33.3.1. Regular Expression API

(REGEXP:MATCH *pattern string* &KEY (:START 0) :END :EXTENDED :IGNORE-CASE :NEWLINE :NOSUB :NOTBOL :NOTEOL)

This macro returns as first value a `REGEXP:MATCH` structure containing the indices of the start and end of the first match for the regular expression *pattern* in *string*; or `NIL` if there is no match. Additionally, a `REGEXP:MATCH` structure is returned for every matched `"\(...\)"` group in *pattern*, in the order that the open parentheses appear in *pattern*. If *start* is non-`NIL`, the search starts at that index in *string*. If *end* is non-`NIL`, only `(SUBSEQ string start end)` is considered.

Example 33.1. REGEXP:MATCH

```
(REGEXP:MATCH "quick" "The quick brown fox jumped quic
⇒ #S(REGEXP:MATCH :START 4 :END 9)
(REGEXP:MATCH "quick" "The quick brown fox jumped quic
⇒ #S(REGEXP:MATCH :START 27 :END 32)
(REGEXP:MATCH "quick" "The quick brown fox jumped quic
⇒ NIL
(REGEXP:MATCH "\\([a-z]*\\)[0-9]*\\(bar\\)" "foo12bar"
⇒ #S(REGEXP:MATCH :START 0 :END 8) ;
⇒ #S(REGEXP:MATCH :START 0 :END 3) ;
⇒ #S(REGEXP:MATCH :START 5 :END 8)
```

(REGEXP:MATCH-START *match*)
(REGEXP:MATCH-END *match*)

Return the start and end the *match*; `SETF`-able.

(REGEXP:MATCH-STRING *string match*)

Extracts the substring of *string* corresponding to the given pair of start and end indices of *match*. The result is shared with *string*. If you want a fresh `STRING`, use `COPY-SEQ` or `COERCE` to `SIMPLE-STRING`.

(REGEXP:REGEXP-QUOTE *string* &OPTIONAL *extended*)

This function returns a regular expression `STRING` that matches exactly *string* and nothing else. This allows you to request an exact string match when calling a function that wants a regular expression.

Example 33.2. REGEXP:REGEXP-QUOTE

```
(regexp-quote "^The cat$")
⇒ "\\^The cat\\$"
```

One use of `REGEXP:REGEXP-QUOTE` is to combine an exact string match with context described as a regular expression. When *`extended`* is non-`NIL`, also quote #\+ and #\?.

(REGEXP:REGEXP-COMPILE *string* &KEY :EXTENDED :IGNORE-CASE :NEWLINE :NOSUB)

Compile the regular expression *`string`* into an object suitable for `REGEXP:REGEXP-EXEC`.

(REGEXP:REGEXP-EXEC *pattern string* &KEY (:START 0) :END :NOTBOL :NOTEOL)

Execute the *`pattern`*, which must be a compiled regular expression returned by `REGEXP:REGEXP-COMPILE`, against the appropriate portion of the *`string`*.

Negative *`end`* means `(+ (LENGTH string) end)`

Returns `REGEXP:MATCH` structures as multiple values (one for each subexpression which successfully matched and one for the whole pattern), unless `:BOOLEAN` was non-`NIL`, in which case return `T` as an indicator of success, but do not allocate anything.

(REGEXP:REGEXP-SPLIT *pattern string* &KEY (:START 0) :END :EXTENDED :IGNORE-CASE :NEWLINE :NOSUB :NOTBOL :NOTEOL)

Return a list of substrings of *`string`* (all sharing the structure with *`string`*) separated by *`pattern`* (a regular expression `STRING` or a return value of `REGEXP:REGEXP-COMPILE`)

(REGEXP:WITH-LOOP-SPLIT (*variable stream pattern* &KEY (:START 0) :END :EXTENDED :IGNORE-CASE :NEWLINE :NOSUB :NOTBOL :NOTEOL) &BODY *body*)

Read lines from *`stream`*, split them with `REGEXP:REGEXP-SPLIT` on *`pattern`*, and bind the resulting list to *`variable`*.

:EXTENDED :IGNORE-CASE :NEWLINE :NOSUB

These options control compilation of a pattern. See `<regex.h>` for their meaning.

:NOTBOL :NOTEOL

These options constrol execution of a pattern. See `<regex.h>` for their meaning.

REGEXP:REGEXP-MATCHER

A valid value for `CUSTOM:*APROPOS-MATCHER*`. This will work only when your *`LOCALE`* is `CHARSET:UTF-8` because **CLISP** uses

CHARSET:UTF-8 internally and POSIX constrains <regex.h> to use the current *LOCALE*.

33.3.2. Example

The following code computes the number of people who use a particular shell:

```
#!/usr/local/bin/clisp -C
(DEFPACKAGE "REGEXP-TEST" (:use "LISP" "REGEXP"))
(IN-PACKAGE "REGEXP-TEST")
(let ((h (make-hash-table :test #'equal :size 10)) (n 0))
  (with-open-file (f "/etc/passwd")
    (with-loop-split (s f ":")
      (incf (gethash (seventh s) h 0))))
  (with-hash-table-iterator (i h)
    (loop (multiple-value-bind (r k v) (i)
            (unless r (return))
            (format t "[~d] ~s~30t== ~5:d~%" (incf n) k v
```

For comparison, the same can be done by the following **Perl**:

```
#!/usr/local/bin/perl -w

use diagnostics;
use strict;

my $IN = $ARGV[0];
open(INF,"< $IN") || die "$0: cannot read file [$IN]: $!\
my %hash;
while (<INF>) {
  chop;
  my @all = split($ARGV[1]);
  my $shell = ($#all >= 6 ? $all[6] : "");
  if ($hash{$shell}) { $hash{$shell} ++; }
  else { $hash{$shell} = 1; }
}
my $ii = 0;
for my $kk (keys(%hash)) {
  print "[",++$ii,"] \"",$kk,"\"  --  ",$hash{$kk},"\n";
}
close(INF);
```

33.4. Advanced Readline and History Functionality

The **“READLINE”** module exports most of the GNU readline functions using **“FFI”**.

This module is present even in the **base** linking set by default on platforms where both GNU readline and **“FFI”** are available.

When this module is present, `*FEATURES*` contains the symbol `:READLINE`.

Lisp-level Functionality

`READLINE:*READLINE-INPUT-STREAM*`
: A `STREAM` (see Section 21.3.13, “Functions EXT:MAKE-BUFFERED-INPUT-STREAM and EXT:MAKE-BUFFERED-OUTPUT-STREAM ”) that receives user input using GNU readline and the standard **CLISP** prompt.

33.5. GDBM - The GNU database manager

This is an interface to the GNU DataBase Manager.

When this module is present, `*FEATURES*` contains the symbol `:GDBM`.

See `modules/gdbm/test.tst` for sample usage.

GDBM module API

`(GDBM:GDBM-VERSION)`
: Return the version string.

`(GDBM:GDBM-OPEN` *`filename`* `&KEY :BLOCKSIZE :READ-WRITE :OPTION :MODE :DEFAULT-KEY-TYPE :DEFAULT-VALUE-TYPE)`
: Open *`filename`* database file. The return value is a GDBM structure. `:READ-WRITE` can have one of following values:

  `:READER`

  `:WRITER`

`:WRCREAT`

`:NEWDB`

and `:OPTION` is one of

`:SYNC`

`:NOLOCK`

`:FAST`

CLISP can store and retrieve values of the following types:

`STRING`

`VECTOR` (meaning anything that can be `COERCE`d to `(VECTOR (UNSIGNED-BYTE 8))`)

`EXT:32BIT-VECTOR` (meaning `(VECTOR (UNSIGNED-BYTE 32))`)

`INTEGER`

`SINGLE-FLOAT`

`DOUBLE-FLOAT`

and `:DEFAULT-KEY-TYPE` and `:DEFAULT-VALUE-TYPE-TYPE` should be one of those. If not specified (or `NIL`), the `:TYPE` argument is required in the access functions below.

If *`filename`* is actually an existing GDBM structure, then it is re-opened (if it has been closed), and returned as is.

The return value is `EXT:FINALIZE`d with `GDBM-CLOSE`.

`(GDBM:GDBM-DEFAULT-KEY-TYPE` ***`db`*****`)`**

`(GDBM:GDBM-DEFAULT-VALUE-TYPE` ***`db`*****`)`**

Return the default data conversion types.

`(GDBM:GDBM-CLOSE` ***`db`*****`)`**

Close the database.

`(GDBM:GDBM-OPEN-P` ***`db`*****`)`**

Check whether *`db`* has been already closed.

Warning

Only the above functions accept closed databases, the following functions `SIGNAL`s an `ERROR` when passed a closed database.

`(GDBM:GDBM-STORE` ***`db key contents`*** **`&KEY :FLAG)`**

`db` is the GDBM structure returned by `GDBM-OPEN`. *`key`* is the key datum. *`contents`* is the data to be associated with the key. `:FLAG` can have one of following values:

`:INSERT`

:REPLACE

(GDBM:GDBM-FETCH *db key* &KEY (TYPE (GDBM:GDBM-DEFAULT-VALUE-TYPE *db*)))

Search the database. The `:TYPE` argument specifies the return type.

(GDBM:GDBM-DELETE *db key*)

Delete *key* and its contents.

(GDBM:GDBM-EXISTS *db key*)

Search data without retrieving it.

(GDBM:GDBM-FIRSTKEY *db* &KEY (TYPE (GDBM:GDBM-DEFAULT-KEY-TYPE *db*)))

Return the key of the first entry, as `:TYPE`. If the database has no entries, the return value is `NIL`.

(GDBM:GDBM-NEXTKEY *db key* &KEY (TYPE (GDBM:GDBM-DEFAULT-KEY-TYPE *db*)))

Return the key that follows *key*, as `:TYPE`, or `NIL` if there are no further entries.

(GDBM:GDBM-REORGANIZE *db*)

Reorganize database.

(GDBM:GDBM-SYNC *db*)

Syncronize the in-memory state of the database to the disk file.

(GDBM:GDBM-SETOPT *db option value*)

Set options on an already open database. *option* is one of following:

:CACHESIZE
: set the size of the internal bucket cache. (default is 100)

:FASTMODE
: `T` or `NIL` (obsolete)

:SYNCMODE
: `T` or `NIL`

:CENTFREE
: `T` or `NIL`

:COALESCEBLKS
: `T` or `NIL`

:DEFAULT-VALUE-TYPE
:DEFAULT-KEY-TYPE
: see `GDBM-OPEN`

(GDBM:GDBM-FILE-SIZE *db*)

Return the underlying file size using `lseek`.

(GDBM:DO-DB (*key db* &REST *options*) &BODY *body*)

Iterate over the database keys, *options* are passed to `GDBM-FIRSTKEY` and `GDBM-NEXTKEY`. *body* is passed to `LOOP`, so you can use all the standard loop contructs, e.g., `(do-db (k db) :collect (list k (gdbm-fetch k)))` will convert the database to an association list.

`(GDBM:WITH-OPEN-DB (`*`db filename`* `&REST` *`options`*`) &BODY` *`body`*`)`

Open the `filename`, execute the *body*, close the database.

33.6. Berkeley DB access

This interface to Berkeley DB from Sleepycat Software exports most functions in the official **C** API. Supported versions:

4.2
4.3
4.4
4.5
4.6

When this module is present, `*FEATURES*` contains the symbol `:BERKELEY-DB`.

See `modules/berkeley-db/test.tst` for sample usage.

33.6.1. Berkeley-DB Objects

Thie module exports the following opaque `STRUCTURE-OBJECT` types:

BDB:DBE
: environment handle

BDB:DB
: database handle

BDB:DBC
: cursor handle

BDB:TXN
: transaction handle

BDB:LOGC
: log cursor handle

BDB:MPOOLFILE
: memory pool file handle

BDB:DBLOCK
: lock handle

They contain the internal handle (a `FFI:FOREIGN-POINTER`), the `LIST` of parents, and the `LIST` of dependents.

33.6.2. Closing handles

`CLOSE` will close (or commit, in the case of a transaction, or put, in the case of a lock) the Berkeley-DB handle objects. garbage-collector will also call `CLOSE`. Closing an object will `CLOSE` all its dependents and remove the object itself from the dependents lists of its parents (but see `BDB:LOCK-CLOSE`).

33.6.3. Database Environment

`(BDB:DB-VERSION &OPTIONAL subsystems-too)`
: Return version information as multiple values:

1. descriptive `STRING` (from `db_version`)
2. major version number (`FIXNUM`)
3. minor version number (`FIXNUM`)

4. patch number (`FIXNUM`)

When the optional argument is non-`NIL`, returns the association list of the subsystem versions as the 5th value.

`(BDB:DBE-CREATE &KEY PASSWORD ENCRYPT HOST CLIENT-TIMEOUT SERVER-TIMEOUT)`
Create an environment handle (`db_env_create`), possibly connecting to a remote host (`DB_ENV->set_rpc_server`) and possibly using encryption with password (`DB_ENV->set_encrypt`).

`(BDB:DBE-CLOSE dbe)`
Close an environment (`DB_ENV->close`). You can also call `CLOSE`.

`(BDB:DBE-MESSAGES dbe)`
Return the verbose messages accumulated so far (requires Berkeley-DB 4.3 or better).

`(BDB:DBREMOVE dbe file database &KEY TRANSACTION AUTO-COMMIT)`
Remove a database (`DB_ENV->dbremove`).

`(BDB:DBREMOVE dbe file database newname &KEY TRANSACTION AUTO-COMMIT)`
Rename a database (`DB_ENV->dbrename`).

`(BDB:DBE-OPEN dbe &KEY FLAGS HOME JOIN INIT-CDB INIT-LOCK INIT-LOG INIT-MPOOL INIT-TXN RECOVER RECOVER-FATAL USE-ENVIRON USE-ENVIRON-ROOT CREATE LOCKDOWN PRIVATE SYSTEM-MEM THREAD MODE)`
Open an environment (`DB_ENV->open`). `:FLAGS` may be the value of a previous call to `(BDB:DBE-GET-OPTIONS dbe :OPEN)`.

`(BDB:DBE-REMOVE dbe &KEY HOME FORCE USE-ENVIRON USE-ENVIRON-ROOT)`
Destroy an environment (`DB_ENV->remove`).

`(BDB:WITH-DBE (var &KEY create options) &BODY body)`
Create an environment, execute *`body`*, close it. *`create`* is a list of options to be passed to `BDB:DBE-CREATE`, *`options`* is a list of options to be passed to `BDB:DBE-SET-OPTIONS`.

33.6.4. Environment Configuration

`(BDB:DBE-SET-OPTIONS dbe &KEY MSGFILE ERRFILE ERRPFX PASSWORD ENCRYPT LOCK-TIMEOUT TXN-TIMEOUT SHM-KEY TAS-SPINS TX-TIMESTAMP TX-MAX DATA-DIR TMP-DIR LG-BSIZE LG-DIR LG-MAX LG-REGIONMAX NCACHE CACHESIZE CACHE LK-CONFLICTS LK-DETECT LK-MAX-LOCKERS LK-MAX-LOCKS LK-MAX-OBJECTS AUTO-COMMIT CDB-ALLDB DIRECT-DB DSYNC-LOG LOG-

AUTOREMOVE LOG-INMEMORY DIRECT-LOG NOLOCKING NOMMAP NOPANIC OVERWRITE PANIC-ENVIRONMENT REGION-INIT TXN-NOSYNC TXN-WRITE-NOSYNC YIELDCPU VERB-CHKPOINT VERB-DEADLOCK VERB-RECOVERY VERB-REPLICATION VERB-WAITSFOR VERBOSE)

Set some environment options using

DB_ENV->set_flags	DB_ENV->set_timeout	DB_ENV->set_lg_regionmax
DB_ENV->set_verbose	DB_ENV->set_encrypt	DB_ENV->set_lk_conflicts
DB_ENV->set_tmp_dir	DB_ENV->set_errfile	DB_ENV->set_lk_detect
DB_ENV->set_data_dir	DB_ENV->set_msgfile	DB_ENV->set_lk_max_lockers
DB_ENV->set_tx_max	DB_ENV->set_errpfx	DB_ENV->set_lk_max_locks
DB_ENV->set_tx_timestamp	DB_ENV->set_lg_bsize	DB_ENV->set_lk_max_objects
DB_ENV->set_tas_spins	DB_ENV->set_lg_dir	DB_ENV->set_cachesize
DB_ENV->set_shm_key	DB_ENV->set_lg_max	

(BDB:DBE-GET-OPTIONS dbe &OPTIONAL *what*)

Retrieve some environment options.

Values of *what*

missing

NIL

all options as a `LIST`

:TX-TIMESTAMP

Recover to the time specified by timestamp rather than to the most current possible date (DB_ENV->get_tx_timestamp)

:TX-MAX

the number of active transactions (DB_ENV->set_tx_max)

:DATA-DIR

list of data directories (DB_ENV->get_data_dir)

:TMP-DIR

temporary directory (DB_ENV->get_tmp_dir). May be `NIL`.

:VERBOSE

the `LIST` of verbosity settings (DB_ENV->get_verbose).

:AUTO-COMMIT

:CDB-ALLDB

:DIRECT-DB

:DSYNC-LOG
:LOG-AUTOREMOVE
:LOG-INMEMORY
:DIRECT-LOG
:NOLOCKING
:NOMMAP
:NOPANIC
:OVERWRITE
:PANIC-ENVIRONMENT
:REGION-INIT
:TXN-NOSYNC
:TXN-WRITE-NOSYNC
:YIELDCPU
:VERB-CHKPOINT
:VERB-DEADLOCK
:VERB-RECOVERY
:VERB-REPLICATION
:VERB-WAITSFOR

a `BOOLEAN` indicator of whether this option is set or not (`DB_ENV->get_verbose` and `DB_ENV->get_flags`).

:LG-BSIZE

log buffer size (`DB_ENV->get_lg_bsize`).

:LG-DIR

logging directory (`DB_ENV->get_lg_dir`).

:LG-MAX

log file size (`DB_ENV->get_lg_max`).

:LG-REGIONMAX

logging region size (`DB_ENV->get_lg_regionmax`).

:NCACHE
:CACHESIZE
:CACHE

cache parameters (`DB_ENV->get_cachesize`).

:LK-CONFLICTS

lock conflicts matrix (`DB_ENV->get_lk_conflicts`).

:LK-DETECT

automatic deadlock detection (`DB_ENV->get_lk_detect`).

:LK-MAX-LOCKERS

maximum number of lockers (`DB_ENV->get_lk_max_lockers`).

:LK-MAX-LOCKS

maximum number of locks (`DB_ENV->get_lk_max_locks`).

:LK-MAX-OBJECTS

maximum number of lock objects (DB_ENV->get_lk_max_objects).

:TAS-SPINS
: the number of test-and-set spins (DB_ENV->get_tas_spins).

:SHM-KEY
: base segment ID for shared memory regions (DB_ENV->get_shm_key).

:LOCK-TIMEOUT
:TXN-TIMEOUT
: timeout values for locks or transactions in the database environment (DB_ENV->get_timeout).

:ENCRYPT
: encryption flags (DB_ENV->get_encrypt_flags).

:ERRFILE
: file descriptor or NIL (DB_ENV->get_errfile).

:MSGFILE
: file descriptor or NIL (DB_ENV->get_msgfile).

:ERRPFX
: STRING or NIL (DB_ENV->get_errpfx).

:DB-XIDDATASIZE
: the LENGTH of the globally unique (VECTOR (UNSIGNED-BYTE 8)) which must be passed to DB_TXN->prepare.

:HOME
: the home directory when open (DB_ENV->get_home).

:OPEN
: the LIST of flags passed to BDB:DBE-OPEN (DB_ENV->get_open_flags).

:CACHE
: database cache information (DB_ENV->get_cachesize).

33.6.5. Database Operations

(BDB:DB-CREATE dbe &KEY XA)
: Create a database handle (db_create).

(BDB:DB-CLOSE db &KEY NOSYNC)
: Close a database (DB->close). You can also call CLOSE.

(BDB:DB-DEL dbe key &KEY TRANSACTION AUTO-COMMIT)
: Delete items from a database (DB->del).

(BDB:DB-FD db)
: Return a file descriptor from a database (DB->fd).

(BDB:DB-GET db key &KEY ACTION AUTO-COMMIT DEGREE-2 DIRTY-READ MULTIPLE RMW TRANSACTION (ERROR T))

Get items from a database (DB->get). If :ERROR is NIL and the record is not found, no ERROR is SIGNALed, instead :NOTFOUND is returned. :ACTION should be one of

:CONSUME	:GET-BOTH
:CONSUME-WAIT	:SET-RECNO

(BDB:DB-PUT db key val &KEY AUTO-COMMIT ACTION TRANSACTION)

Store items into a database (DB->put). :ACTION should be one of

:APPEND :NODUPDATA :NOOVERWRITE

(BDB:DB-STAT db &KEY FAST-STAT TRANSACTION)

Return database statistics (DB->get_byteswapped, DB->get_type, DB->stat).

(BDB:DB-OPEN db file &KEY DATABASE TYPE MODE FLAGS CREATE DIRTY-READ EXCL NOMMAP RDONLY THREAD TRUNCATE AUTO-COMMIT TRANSACTION)

Open a database (DB->open). :TYPE should be one of

:BTREE	:RECNO
:HASH	:UNKNOWN (default)
:QUEUE	

:FLAGS may be the value of a previous call to (BDB:DB-GET-OPTIONS db :OPEN)

(BDB:DB-SYNC db)

Flush a database to stable storage (DB->sync).

(BDB:DB-TRUNCATE db &KEY TRANSACTION AUTO-COMMIT)

Empty a database (DB->truncate).

(BDB:DB-UPGRADE db file &KEY DUPSORT)

Upgrade a database (DB->upgrade).

(BDB:DB-RENAME db file database newname)

Rename a database (DB->rename).

(BDB:DB-REMOVE db file database)

Remove a database (DB->remove).

(BDB:DB-JOIN db cursor-sequence &KEY JOIN-NOSORT)

Create a specialized join cursor for use in performing equality or natural joins on secondary indices (DB->join).

(BDB:DB-KEY-RANGE db key &KEY TRANSACTION)

return an estimate of the proportion of keys that are less than, equal to, and greater than the specified key (DB->key_range). The underlying database must be of type Btree.

(BDB:DB-VERIFY db file &KEY DATABASE SALVAGE AGGRESSIVE PRINTABLE NOORDERCHK)
: Verify/salvage a database (DB->verify). :SALVAGE, if supplied, should be the output file name. :DATABASE, if supplied, will force DB_ORDERCHKONLY.

(BDB:WITH-DB (var dbe file &KEY *create options open*) &BODY body)
: Open the database, execute *body*, close it. *create* is a list of options to be passed to BDB:DB-CREATE, *options* is a list of options to be passed to BDB:DB-SET-OPTIONS, *open* is a list of options to be passed to BDB:DB-OPEN.

33.6.6. Database Configuration

(BDB:DB-SET-OPTIONS db &KEY ERRFILE MSGFILE ERRPFX PASSWORD ENCRYPTION NCACHE CACHESIZE CACHE LORDER PAGESIZE BT-MINKEY H-FFACTOR H-NELEM Q-EXTENTSIZE RE-DELIM RE-LEN RE-PAD RE-SOURCE CHKSUM ENCRYPT TXN-NOT-DURABLE DUP DUPSORT INORDER RECNUM REVSPLITOFF RENUMBER SNAPSHOT)
: Set some database options using

DB_ENV->set_errfile	DB->set_pagesize	DB->set_re_len
DB_ENV->set_msgfile	DB->set_bt_minkey	DB->set_re_pad
DB_ENV->set_errpfx	DB->set_h_ffactor	DB->set_re_source
DB->set_encrypt	DB->set_h_nelem	DB->set_flags
DB->set_cachesize	DB->set_q_extentsize	
DB->set_lorder	DB->set_re_delim	

(BDB:DB-GET-OPTIONS db &OPTIONAL *what*)
: Retrieve some database options.

Values of *what*

missing
NIL
: all options as a LIST

:FLAGS
: all flags (DB_ENV->get_flags).

:CHKSUM
:ENCRYPT
:TXN-NOT-DURABLE
:DUP
:DUPSORT
:INORDER
:RECNUM
:REVSPLITOFF
:RENUMBER
:SNAPSHOT
: a `BOOLEAN` indicator of whether this option is set or not (`DB_ENV->get_verbose` and `DB_ENV->get_flags`).

:CACHE
: database cache information (`DB->get_cachesize` or `DB_ENV->get_cachesize` if the database was created within an environment).

:ENCRYPTION
: encryption flags (`DB_ENV->get_encrypt_flags`).

:ERRFILE
: file descriptor or `NIL` (`DB_ENV->get_errfile`).

:MSGFILE
: file descriptor or `NIL` (`DB_ENV->get_msgfile`).

:ERRPFX
: `STRING` or `NIL` (`DB_ENV->get_errpfx`).

:PAGESIZE
: database page size (`DB->get_pagesize`).

:BT-MINKEY
: the minimum number of key/data pairs intended to be stored on any single `:BTREE` leaf page underlying source file (`DB->get_bt_minkey`).

:H-FFACTOR
: the desired density within the `:HASH` table (`DB->get_h_ffactor`).

:H-NELEM
: an estimate of the final size of the `:HASH` table (`DB->get_h_nelem`).

:Q-EXTENTSIZE
: the size of the extents used to hold pages in a `:QUEUE` database (`DB->get_q_extentsize`).

:RE-DELIM
: the record delimiter for `:RECNO` databases (`DB->get_re_delim`).

:RE-LEN
: database record length (DB->get_re_len).

:RE-PAD
: database record pad byte (DB->get_re_pad).

:RE-SOURCE
: the underlying source file for `:RECNO` databases (DB->get_re_source).

:LORDER
: database byte order (DB->get_lorder).

:DBNAME
: the file name and database name (DB->get_dbname)

:TRANSACTIONAL
: the indicator whether the database is transactional (DB->get_transactional).

:OPEN
: the flags passed to BDB:DB-OPEN (DB->get_open_flags).

Warning

Once you call a method for one type of access method, the handle can only be used for that type. The methods DB->get_re_delim and DB->get_re_source are for a `:RECNO` database so you *cannot* call them (by passing `:RE-DELIM` or `:RE-SOURCE` to this function) and then use the database handle to open a database of different type (e.g., `:QUEUE`).

33.6.7. Database Cursor Operations

(BDB:MAKE-DBC db &KEY DEGREE-2 DIRTY-READ WRITECURSOR TRANSACTION)
: Create a cursor handle (DB->cursor).

(BDB:DBC-CLOSE cursor)
: Close the cursor handle (DBCursor->close). You can also call CLOSE.

(BDB:DBC-COUNT cursor)
: Return count of duplicates (DBCursor->count).

(BDB:DBC-DEL cursor)
: Delete by cursor (DBCursor->del).

(BDB:DBC-DUP cursor &KEY POSITION)

Duplicate a cursor (DBCursor->dup).

(BDB:DBC-GET cursor key data action &KEY DEGREE-2 DIRTY-READ MULTIPLE (ERROR T))

Retrieve by cursor (DBCursor->get). If :ERROR is NIL and the record is not found, no ERROR is SIGNALed, :NOTFOUND or :KEYEMPTY is returned instead, as appropriate. *action* should be one of

:CURRENT	:GET-RECNO	:NEXT-DUP	:SET
:FIRST	:JOIN-ITEM	:NEXT-NODUP	:SET-RANGE
:GET-BOTH	:LAST	:PREV	:SET-RECNO
:GET-BOTH-RANGE	:NEXT	:PREV-NODUP	

(BDB:DBC-PUT cursor key data flag)

Store by cursor (DBCursor->put).

(BDB:WITH-DBC (var &REST *options*) &BODY *body*))

Open a cursor, execute *body*, close it. *options* are passed to BDB:MAKE-DBC.

33.6.8. Lock Subsystem

(BDB:LOCK-DETECT dbe action)

Perform deadlock detection (DB_ENV->lock_detect).

(BDB:LOCK-ID dbe)

Acquire a locker ID (DB_ENV->lock_id).

(BDB:LOCK-ID-FREE dbe id)

Release a locker ID (DB_ENV->lock_id_free). All associated locks should be released first.

(BDB:LOCK-GET dbe object locker mode &KEY NOWAIT)

Acquire a lock (DB_ENV->lock_get). The BDB:DBLOCK object returned by this function will **not** be released when the environment is closed. This permits long-lived locks.

(BDB:LOCK-PUT dbe lock)

Release a lock (DB_ENV->lock_put).

(BDB:LOCK-CLOSE lock)

Release a lock (DB_ENV->lock_put) using the environment with which it has been acquired. This is used to EXT:FINALIZE BDB:DBLOCK objects.

Warning

If that environment has already been closed, you are in a big trouble (segfault), so you better release your locks or do not drop them.

(BDB:LOCK-STAT dbe &KEY STAT-CLEAR)
: Return lock subsystem statistics (`DB_ENV->lock_stat`).

33.6.9. Log Subsystem

(BDB:LOG-ARCHIVE dbe &KEY ARCH-ABS ARCH-DATA ARCH-LOG ARCH-REMOVE)
: Return a list of log or database filenames (`DB_ENV->log_archive`).

(BDB:LOG-FILE dbe lsn)
: Return the name of the file containing the record named by *`lsn`* (`DB_ENV->log_file`).

(BDB:LOG-FLUSH dbe lsn)
: Flush log records to disk (`DB_ENV->log_flush`).

(BDB:LOG-PUT dbe data &KEY :FLUSH)
: Write a log record (`DB_ENV->log_put`).

(BDB:LOG-STAT dbe &KEY STAT-CLEAR)
: Logging subsystem statistics (`DB_ENV->log_stat`).

33.6.9.1. Log Cursor Operations

(BDB:LOG-CURSOR dbe)
: Create a log cursor handle (`DB_ENV->log_cursor`).

(BDB:LOGC-CLOSE logc)
: Close a log cursor handle (`DB_LOGC->close`).

(BDB:LOGC-GET logc action &KEY TYPE ERROR)
: Retrieve a log record (`DB_LOGC->get`). If `:ERROR` is `NIL` and the record is not found, no `ERROR` is `SIGNAL`ed, `:NOTFOUND` is returned instead.

Valid *`action`*s

:CURRENT
:FIRST
:LAST
:NEXT
:PREV
Retrieve the appropriate record.

DB:LSN
Retrieve the specified record, as with `DB_SET`.

Returns two values: the datum of type specified by the `:TYPE` argument and the DB:LSN value of the record retrieved (when *`action`* is a DB:LSN, it is returned unchanged).

33.6.9.2. Log Sequence Numbers

Use `EQUALP` to check similarity of BDB:LSN objects.

`(BDB:LOG-COMPARE lsn1 lsn2)`
Compare two Log Sequence Numbers (`log_compare`).

33.6.10. Memory Pool Subsystem

not implemented yet, patches are welcome

33.6.11. Replication

not implemented yet, patches are welcome

33.6.12. Sequences

not implemented yet, patches are welcome

33.6.13. Transaction Subsystem

`(BDB:TXN-BEGIN dbe &KEY DEGREE-2 PARENT DIRTY-READ NOSYNC NOWAIT SYNC)`

Begin a transaction (DB_ENV->txn_begin).

(BDB:TXN-ABORT txn)
: Abort a transaction (DB_TXN->abort).

(BDB:TXN-COMMIT txn &KEY NOSYNC SYNC)
: Commit a transaction (DB_TXN->commit).

(BDB:TXN-DISCARD txn)
: Discard a transaction (DB_TXN->discard).

(BDB:TXN-ID txn)
: Return the transaction's ID (DB_TXN->id).

(BDB:TXN-CHECKPOINT dbe &KEY KBYTE MIN FORCE)
: Checkpoint the transaction subsystem (DB_ENV->txn_checkpoint).

(BDB:TXN-PREPARE txn id)
: Initiate the beginning of a two-phase commit (DB_TXN->prepare).

(BDB:TXN-RECOVER dbe &KEY FIRST NEXT)
: Return a list of prepared but not yet resolved transactions (DB_ENV->txn_recover).

(BDB:TXN-SET-TIMEOUT txn timeout which)
: Set timeout values for locks or transactions for the specified transaction (DB_TXN->set_timeout).

(BDB:TXN-STAT dbe &KEY STAT-CLEAR)
: Transaction subsystem statistics (DB_ENV->txn_stat).

33.7. Directory Access

This module provides some directory access from lisp, in package **“LDAP”**.

When this module is present, `*FEATURES*` contains the symbol `:DIRKEY`.

3 types of directory keys may exist, depending on the compilation environment.

valid directory key types

:win32
: Win32 registry access

:gnome
: gnome-config access

:ldap
: LDAP interface via OpenLDAP or compatible

The following functions and macros are exported (please note that these features are experimental and the API may be modified in the future).

(LDAP:DIR-KEY-OPEN *dkey pathname* &KEY (:DIRECTION :INPUT) :IF-DOES-NOT-EXIST)
: Open the directory key under *dkey*, which should be either an open directory key or a valid directory key type. The meaning of the `:DIRECTION` and `:IF-DOES-NOT-EXIST` keyword arguments is the same as for `OPEN`.

(LDAP:DIR-KEY-CLOSE *dkey*)
: Close the directory key. The preferred way is to use the `LDAP:WITH-DIR-KEY-OPEN` macro.

(LDAP:WITH-DIR-KEY-OPEN (*variable dkey pathname* &REST {*option*}*) &BODY *body*)
: Open the directory key (by calling `LDAP:DIR-KEY-OPEN` on *dkey*, *pathname* and *options*), bind it to *variable*, execute *body*, then close it with `LDAP:DIR-KEY-CLOSE`.

(LDAP:DIR-KEY-TYPE *dkey*)
: Return the directory key type of the directory key

(LDAP:DIR-KEY-PATH *dkey*)
: Return the path of this directory key, which is the *pathname* argument of `LDAP:DIR-KEY-OPEN` if *dkey* was a directory key type or the concatenation of the *pathname* argument and the `ldap:dir-key-path` of *dkey*.

(LDAP:DIR-KEY-DIRECTION *dkey*)
: One of `:INPUT`, `:OUTPUT` and `:IO`, indicating the permitted operation on this key and its derivatives.

(LDAP:DIR-KEY-CLOSED-P *dkey*)
: Check whether the key has been closed. It is not an error to close a closed key.

(LDAP:DIR-KEY-SUBKEY-DELETE *dkey subkey*) (LDAP:DIR-KEY-VALUE-DELETE *dkey attribute*)
: Delete the specified subkey or attribute.

(LDAP:DIR-KEY-SUBKEY *dkey*) (LDAP:DIR-KEY-ATTRIBUTES *dkey*)
: Return the list of the subkeys or attributes.

(LDAP:DIR-KEY-VALUE *dkey attribute* &OPTIONAL *default*)
: Return the value of the specified attribute, similar to `GETHASH` and `SETF`able just like `GETHASH`.

(LDAP:DIR-KEY-INFO *dkey*)
: Return some information about the directory key. This is highly platform-dependent and will probably be removed or replaced or modified in the future.

(LDAP:WITH-DIR-KEY-SEARCH (*key-iter atribute-iter dkey pathname* &KEY :scope) &BODY *body*)

This is the main way to iterate over the subtree under the key *dkey*+*pathname*.

key-iter is a non-`NIL` symbol and is bound via `MACROLET` to a macro, each call of which returns the next subkey.

atribute-iter is a symbol and is bound, when non-`NIL`, to a macro, each call of which returns two values - the next attribute and its value.

The `:scope` keyword argument specifies the scope of the search and can be

:self
: iterate over the key itself

:level
: iterate over the children of the key

:tree
: iterate over the subtree

`LDAP:WITH-DIR-KEY-SEARCH` is used to implement `LDAP:DIR-KEY-VALUES`, `LDAP:DIR-KEY-CHILDREN` and `LDAP:DIR-KEY-DUMP-TREE` in `modules/dirkey/dirkey.lisp`.

33.8. PostgreSQL Database Access

This package offers an **"FFI"**-based interface to PostgreSQL.

The package **"SQL"** (nicknamed **"POSTGRES"** and **"POSTGRESQL"**) is case-sensitive, so you would write `(sql:PQconnectdb ...)` when you need to call `PQconnectdb()`.

When this module is present, `*FEATURES*` contains the symbol `:POSTGRESQL`.

See `modules/postgresql/test.tst` for sample usage.

Additionally, some higher level functionality is available:

(sql:pq-finish *connection*)
: `PQfinish` the *connection* and mark it as invalid

(sql:pq-clear *result*)

`PQclear` the *`result`* and mark it as invalid

`(sql:sql-error` ***`connection result format-string`*** **`&REST`** ***`arguments`*`)`**

finalize *`connection`* and *`result`* and `SIGNAL` an appropriate `ERROR`

`(sql:sql-connect &KEY host port options tty name login password)`

call `PQsetdbLogin` and return the *`connection`*

`(sql:with-sql-connection (`*`variable`* `&REST` *`options`* `&KEY` *`log`* `&ALLOW-OTHER-KEYS) &BODY` *`body`*`)`

1. bind `*sql-log*` to the *`log`* argument
2. call `sql:sql-connect` on *`option`*s and bind *`variable`* to the result
3. execute *`body`*
4. call `sql:pq-finish` on *`variable`*

`(sql:sql-transaction` ***`connection command status`*** **`&OPTIONAL (clear-p T))`**

execute the *`command`* via *`connection`*; if the status does not match *`status`*, `ERROR` is `SIGNAL`ed; if *`clear-p`* is non-`NIL` `sql:pq-clear` the *`result`*; otherwise return it

`(sql:with-sql-transaction (`*`result connection command status`*`) &BODY` *`body`*`)`

execure the *`body`* on the *`result`* of *`command`*, then `sql:pq-clear` the *`result`*

`sql:*sql-login*`

the default *`login`* argument to `sql:sql-connect` (initially set to `"postgres"`)

`sql:*sql-password*`

the default *`password`* argument to `sql:sql-connect` (initially set to `"postgres"`)

`sql:*sql-log*`

when non-`NIL`, should be a `STREAM`; `sql:sql-connect` and `sql:sql-transaction` will write to it (initially set to `NIL`)

Warning

Since `PQfinish` and `PQclear` cannot be called on the same pointer twice, one needs to track their validity (`sql:sql-`

`connect` and `sql:sql-transaction` take care of that). See Example 32.10, “Controlling validity of resources”.

33.9. Oracle Interface

The Oracle module allows a **CLISP** program to act as client to an Oracle database server. The module includes full SQL support, transactions (including auto-commit), support for most Oracle data types (LONG, BLOB, CLOB, RAW, etc.), automatic conversion between Oracle and **Common Lisp** data types, database connection caching and retry, concurrent connections to multiple databases, proper handling of Oracle errors, and more.

The module can be used to build sophisticated Oracle database applications in **Common Lisp**.

When this module is present, `*FEATURES*` contains the symbol `:ORACLE`.

33.9.1. Functions and Macros in package “ORACLE”

Access to Oracle is via these functions and macros in package **“ORACLE”**. When any Oracle function fails, the general Lisp function `ERROR` is called, with the condition string set to include the Oracle error number, the Oracle message text, and other context of the error (e.g., the text and parse location of a SQL query).

`(ORACLE:CONNECT` ***`user password server`*** **`&OPTIONAL`** ***`schema auto-commit prefetch-buffer-bytes long-len truncate-ok`*** **`)`**

Connect to an Oracle database. All subsequent operations will affect this database until the next call to `ORACLE:CONNECT`. A single program can access different Oracle schemas concurrently by repeated calls to `ORACLE:CONNECT`. Database connections are cached and re-used: if you call `ORACLE:CONNECT` again with the same *`user`*, *schema*, and *server*, the previous Oracle connection will be re-used. `ORACLE:CONNECT` may not be called inside `WITH-TRANSACTION`. Returns: `T` if a cached connection was re-used, `NIL` if a new connection was created (and cached). The meaning of the arguments is as follows:

Arguments for `ORACLE:CONNECT`

user
: Oracle user ID

password
: Password for user, or `NIL` if *user* has no password (!).

server
: Oracle server ID (SID).

schema
: Oracle default schema (default: `NIL`). If `NIL`, same as user. This allows you to log on with one user's id/password but see the database as if you were some other user.

auto-commit
: Flag: whether to commit after every operation (default: `T`). Set this to `NIL` if you intend to do transactions and call `COMMIT` explicitly. However, `WITH-TRANSACTION` is probably easier.

prefetch-buffer-bytes
: Number of bytes to cache from SQL SELECT fetches (default: 64 Kbytes) If you are very short of memory, or have a slow connection to Oracle, you can reduce this to 10k or so. Alternatively, if you have a fast connection to Oracle and regularly do large queries, you can increase throughput by increasing this value.

long-len
: Number of bytes to fetch for "long" (LONG, [BC]LOB) types. Long data that exceeds this size will raise an error, or be truncated depending on the value of *truncate-ok* (below). Setting *long-len* to zero and *truncate-ok* to `NIL` will disable long fetching entirely. If *long-len* is `NIL` or negative, defaults to 500k bytes.

truncate-ok
: Flag: if set, allow truncation of LONG columns to *long-len* bytes on fetch; otherwise, fetches of LONG columns exceeding *long-len* bytes will raise an error. Default: `NIL`.

`(ORACLE:DISCONNECT)`

Disconnect from the database currently connected. No more calls can be made until `ORACLE:CONNECT` is called again. The connection is closed and removed from the connection cache. Does nothing if there is no connection. `DISCONNECT` may not be called inside `WITH-TRANSACTION`. Returns `NIL`.

`(ORACLE:RUN-SQL` *`sql`* `&OPTIONAL` *`params is-select`*`)`

Execute a SQL statement. Must be `ORACLE:CONNECT`ed to a database. Returns the number of rows affected by the SQL operation, for non-SELECT statements, zero for SELECT statements. For destructive database operations (INSERT, UPDATE, DELETE), the results are committed to the database immediately if *`auto-commit`* when establishing the current connection; see `ORACLE:CONNECT`. The meaning of the arguments is as follows:

Arguments for `RUN-SQL`

`sql`
: Text of SQL statement, as a string. The *`sql`* statement may contain Oracle "named parameters," e.g. ":myparam" whose values will be substituted from the parameters given in *`params`*.

`params`
: A mapping of the names of the bind-parameters in the query to their values. The set of named parameters in the query must match exactly the keys mapped by *`params`*. The mapping may be passed as either (1) a hash table whose keys are the named parameters or (2) a list of pairs, ((name value) (name value) ...). Parameter values passed from Lisp are converted to the appropriate Oracle data types (see `FETCH`).

`is-select`
: Flag: whether the statement is a SELECT query. You usually do not need to set this as it is detected by default based on the SQL text. However, there are situations, such as when a SELECT query begins with comment, that you need to specify it explicitly.

`(ORACLE:DO-ROWS` *`vars`* `&BODY` *`body`*`)`

Macro which loops over a SQL SELECT result, evaluating, for each row in the result, the forms in *`body`*, binding symbols given in *`vars`* to corresponding database columns in the SELECT result. The argument *`vars`* must be a non-empty list of symbols matching a subset of the columns of an active SELECT query. If a SELECT column is an Oracle expression such as `SUBSTR(mycol, 1, 10)`, it

is recommended to use a column alias, e.g., `SELECT SUBSTR (mycol, 1, 10) AS myvar`, in which case the column alias will be used as the symbol bound to the column value.
As `DO-ROWS` expands into a `DO*` loop, it may be terminated prematurely, before all rows are fetched, by using `RETURN` anywhere in *`body`*.
It is allowed to call `ORACLE:CONNECT` in the *`body`* of the loop, but only to switch the connection to a database other than the one that was used to do the SELECT. This is useful for reading from one database while writing to another.
In *`vars`*, instead of a single symbol, a pair (*`bound-var "column-name"`*) may be specified, which will cause values from the SELECTed column or alias , *`column-name`*, to be bound to Lisp variable, *`bound-var`*. This is for unusual cases where a Lisp variable cannot be created with the same name as the column (e.g., a column named "T"), or when it is inconvenient or impossible to alias the column with `SELECT ... AS`.

`(ORACLE:FETCH &OPTIONAL result-type)`

Fetch a single row of data. Returns a row of values corresponding to the columns of an active SELECT statment. The row data is returned in one of three different forms, depending on the value of the symbol *`result-type`*:

Return values for `FETCH`

`ARRAY`
: Values will be returned in an `ARRAY` with the same number of columns as in the SELECT statement, in the same order. This is the default.

`PAIRS`
: A list of pairs, `((column, value) ...)` is be returned. The number and order of pairs is the same as the columns in the SELECT statement.

`HASH`
: A `HASH-TABLE` whose keys are the column names and whose values are the column values in the row. The SELECT columns *must be unique* and be valid Lisp symbols to use this option. If you are SELECTing an expression, you probably want to use a column alias: `SELECT <expr> AS some_alias ...`

The following data type conversions are done between Oracle datatypes and **Common Lisp** data types:

Oracle type	Converts to/from Common Lisp type
Numeric (NUMBER, INTEGER, FLOAT)	The appropriate Common Lisp numeric type (`FIXNUM`, `BIGNUM`, `FLOAT`)
String (CHAR, VARCHAR, VARCHAR2)	A Common Lisp `STRING`. Note that CHAR will be padded out to its full, fixed length as defined in Oracle; VARCHAR will be a string of variable length. Also note that Oracle has no "zero-length string" value - it returns the SQL special value `NULL` which is converted to `NIL` (see below).
DATE	A string of the form "YYYY-MM-DD HH:MM:SS" where HH is 24-hour form. If you want dates formatted differently, convert them to strings in Oracle using `SELECT TO_CHAR (mydate, '`*`template`*`') AS mydate`; the result will then be returned as a string, formatted as per *`template`*.
RAW, LONG RAW	A hexadecimal string, with two hex digits for each byte of Oracle data. Note that this means the Lisp string will be twice the size, in bytes, as the Oracle data.
"Large" types (LONG, BLOB, CLOB)	A Lisp string of (arbitrary, possibly binary) data. Note that truncation may occur; see the `ORACLE:CONNECT` parameters *`long-len`* and *`truncate-ok`*.
`NULL`	The Common Lisp value `NIL`

`(ORACLE:FETCH-ALL &OPTIONAL` ***`max-rows result-type item-type`*****`)`**

Fetch some or all the rows from a query and return result as a sequence of sequences. Arguments are all optional: *`max-rows`* limits the result to that numbers of rows; *`result-type`* is the type of sequence of the rows, either '`ARRAY` (the default) or '`LIST`; *`item-type`* is the type of sequence of the column values for each row, either '`ARRAY` (the default) or '`LIST`. Each row fetched always contains the full set of column values SELECTed.

`FETCH-ALL` is often useful in conjunction with `MAP` or `REDUCE` to iterate over an entire SELECT result to construct a single Lisp value.

`(ORACLE:PEEK &OPTIONAL` ***`result-type`*****`)`**

Peek at next row of data (without fetching it). Returns a row a la `FETCH`, except does not advance to the next row. Repeated calls to `PEEK` will thus return the same row of data. Returns `NIL` if at EOF. If data is available, returns row data just as `FETCH` (see `FETCH` for data format and conversions done). Optional argument *`result-type`* is the type of sequence of the column values for the returned row, either `ARRAY` (the default) or `LIST`. `PEEK` is a useful look-ahead for database reporting functions that may need to "break" on changes in data to print headers, summaries, etc.

`(ORACLE:COLUMNS)`

Returns information on the columns of a SELECT result, in the form of an array of SQLCOL structures, one for each result column in the most recent SELECT statement. It is not necessary to have called `FETCH` before requesting column information on the query, however the query must have been compiled and executed with `RUN-SQL`. Each SQLCOL structure has these slots:

Slots of `SQLCOL`

NAME
: The Oracle column name or the expression selected. If the query used a column alias, `SELECT` *`expr`* `AS` *`alias`*, then *`alias`* will be returned as the column name.

TYPE
: Oracle data type (VARCHAR, NUMBER, DATE, ...)

SIZE
: Oracle data length (useful mostly for character types)

SCALE
: For numeric (NUMBER) types, number of digits to right of decimal; NIL for FLOAT

PRECISION
: For numeric types, total number of significant digits (decimal digits for NUMBER, bits for FLOAT)

NULL_OK
: `T` if `NULL`s allowed, `NIL` if `NULL`s are not allowed.

To access the values of the SQLCOL structures, use the standard accessor functions, e.g., `(ORACLE:SQLCOL-NAME (elt (ORACLE:COLUMNS) 0))`

`(ORACLE:EOF)`

Returns EOF status. A SELECT query cursor is considered at EOF if the next FETCH would return no data. Must be connected to a database, and have an active SELECT statement.

(ORACLE:INSERT-ROW *table values*)

Inserts a single row into `table`. Second argument `values` is a map of column names to values: either a hash table whose keys are the column names, or a list of (name, value) pairs. Columns missing from the map will be given the default Oracle value, or `NULL`. Returns the number of rows inserted (i.e., always 1).

(ORACLE:UPDATE-ROW *table condition vals* &OPTIONAL *params*)

Updates rows in `table`. Second argument `condition` is a string expression for a WHERE clause (without the "WHERE") which determines which rows are updated. Third argument `vals` is a map of columns to be updated to their new values: a hash table whose keys are column names, or list of (name, value) pairs. Optional `params` specifies values for named parameters that may occur in `condition`, e.g., when the condition is a match on a primary key, e.g.: `"pk_column = :pk_val"`. Returns the number of rows updated.

(ORACLE:ROW-COUNT)

For SELECT statements, returns the number of rows `FETCH`ed (**not** `PEEK`ed) so far. For other statements (e.g., INSERT, UPDATE, DELETE), returns the number of rows affected by the last operation (e.g., inserted, updated, deleted). Must be connected to a database and have an active SQL statement.

(ORACLE:WITH-TRANSACTION &BODY *body*)

Evaluates the forms in *body* atomically as a database transaction, ensuring that either all the database operations done in *body* complete successfully, or none of them do. If pending (un-committed) changes exist when this macro is entered, they are *rolled back* (undone), so that the database is affected only by the subsequent updates inside *body*. Nesting of `WITH-TRANSACTION` blocks is not allowed and will raise an error. There is no effect on the status of `auto-commit` given in `ORACLE:CONNECT`; it resumes its previous state when the macro exits. The value of the `WITH-TRANSACTION` expression is that of the last form in *body*.

(ORACLE:COMMIT)

Commits (makes permanent) any pending changes to the database. The `auto-commit` parameter to `ORACLE:CONNECT` must not have been set to use this function, nor can it be called inside a WITH-TRANSACTION block. Always returns NIL.

(ORACLE:ROLLBACK)
: Rolls back (undoes and abandons) any pending changes to the database. The *auto-commit* parameter to ORACLE:CONNECT must not have been set to use this function, nor can it be called inside a WITH-TRANSACTION block. Always returns NIL.

(ORACLE:AUTO-COMMIT)
: Toggles the state of *auto-commit* initially given to ORACLE:CONNECT for the current connection. With *auto-commit* enabled, modifications to the database are committed (made permanent) after each destructive SQL operation made with calls to RUN-SQL, INSERT-ROW, UPDATE_ROW, etc. With *auto-commit* disabled, transactional integrity is under the programmer's control and is managed either by (1) explicitly calling COMMIT or ROLLBACK to commit or undo the pending operations, or (2) wrapping code blocks with database operations inside the WITH-TRANSACTION macro. AUTO-COMMIT returns the previous status of *auto-commit*. AUTO-COMMIT may not be called inside WITH-TRANSACTION.

33.9.2. Oracle Example

Below is a simple example script which uses Oracle's demo database schema, SCOTT.

```
(setf server "orcl") ; Change this to your server's SID
(oracle:connect "scott" "tiger" server)

(oracle:run-sql "SELECT deptno, dname, loc FROM dept ORDE
(oracle:do-rows (deptno dname loc)
  (format t "Dept. no is '~A', " deptno)
  (format t "Dept. name is '~A', " dname)
  (format t "Dept. loc is '~A'~%" loc))

(oracle:update-row "dept" "dname = :acctval" '(("dname" "

(oracle:run-sql "SELECT deptno, dname, loc FROM dept ORDE
(oracle:do-rows (deptno dname loc)
  (format t "Dept. no is '~A', " deptno)
  (format t "Dept. name is '~A', " dname)
  (format t "Dept. loc is '~A'~%" loc))
```

```
(oracle:update-row "dept" "dname = :acctval" '(("dname" "A
```

33.9.3. Oracle Configuration

Obviously, a working Oracle environment is required. It is recommended that you first be able to log on and use the Oracle SQL*Plus application to test your environment *before* attempting Oracle access via the **CLISP** module. At a minimum you will need to set environment variables `ORACLE_HOME` to the Oracle base directory and `LD_LIBRARY_PATH` to include `$ORACLE_HOME/lib` and possibly other directories.

33.9.4. Building the Oracle Interface

The module uses the Oracle Call Interface (OCI) **C** library. To build the module you will need the Oracle OCI headers and link libraries; as a quick check, make sure you have the file `oci.h` somewhere under `ORACLE_HOME`, probably in `$ORACLE_HOME/rdbms/demo/oci.h`.

To build the module into **CLISP**, configure with **./configure ... --with-module=oracle ...**. The **full** linking set will contain the module, so you will need to use the `-K` option to use it. You can test that you really have the Oracle-enabled **CLISP** by evaluating `(DESCRIBE 'oracle:connect)`.

Note

It may be necessary to edit file `modules/oracle/Makefile` prior to running **./configure**.

33.10. LibSVM Interface

This is an **"FFI"**-based interface to the version 2.84 of LibSVM (included in the source distribution in the directory `modules/libsvm/`, so you do not need to install it yourself).

The package **"LIBSVM"** is case-sensitive, and you do not need the `svm_` prefix for the functions described in `modules/libsvm/README`.

When this module is present, `*FEATURES*` contains the symbol `:LIBSVM`.

See `modules/libsvm/test.tst` for sample usage.

33.10.1. Types

All data is kept on the **C** side as much as possible, so these foreign types do **not** have a **CLOS** counterpart.

node
: Corresponds to svm_node, represented as a `LIST` on the lisp side.

problem
: Corresponds to svm_problem, represented as a `LIST` on the lisp side.

parameter
: Corresponds to svm_parameter, represented as a `VECTOR` on the lisp side.

model
: Corresponds to svm_model, an opaque `FFI:FOREIGN-POINTER`.

33.10.2. Functions

33.10.2.1. Functions related to problem

`(problem-l` *`problem`*`)`

Return the number of rows in the *problem* (a `FFI:FOREIGN-VARIABLE`)

`(problem-y problem &OPTIONAL (length (problem-l problem)))`

Return a `(VECTOR DOUBLE-FLOAT length)` representing the targets in the *problem* (a `FFI:FOREIGN-VARIABLE`).

`(problem-y-n problem n &OPTIONAL (length (problem-l problem)))`

Return the `DOUBLE-FLOAT` representing the *n*th target in the *problem* (a `FFI:FOREIGN-VARIABLE`).

`(problem-x problem &OPTIONAL (length (problem-l problem)))`

Return a `(VECTOR (VECTOR node) length)` representing the predictors in the *problem* (a `FFI:FOREIGN-VARIABLE`).

`(problem-x-n problem n &OPTIONAL (length (problem-l problem)))`

Return the `(VECTOR node)` representing the *n*th set of predictors in the *problem* (a `FFI:FOREIGN-VARIABLE`).

`(make-problem &KEY l y x)`

Allocate a `FFI:FOREIGN-VARIABLE` representing a model.

`(destroy-problem problem)`

Release the memory taken by the problem object and invalidate the `FFI:FOREIGN-VARIABLE` *problem*.

Warning

You *must* call this function yourself, but only *after* deallocating all model objects trained from this *problem*. See `modules/libsvm/README` for more information.

`(load-problem filename &KEY (log *STANDARD-OUTPUT*))`

Read a problem from a file in the libsvm/svmlight format. Return two values: the problem and max index (i.e., the number of columns).
Messages go to *log*.

`(save-problem filename problem &KEY (log *STANDARD-OUTPUT*))`

Write a problem into a file.
Messages go to *log*.

33.10.2.2. Functions related to model

(destroy-model *model*)
: Release the memory taken by the model object and invalidate the `FFI:FOREIGN-VARIABLE` *model*.
Calls `svm_destroy_model`.
You do **not** have to call this function yourself, it is attached to the *model* by `train` and `load-model` via `EXT:FINALIZE`.

(check-parameter *problem* *parameter*)
: Check if the *parameter* is appropriate for the *problem*.
Calls `svm_check_parameter`.

(train *problem* *parameter*)
: Train a model.
Calls `svm_train` and `check-parameter`.

(cross-validation *problem* *parameter* *n*)
: Run *n*-fold cross-validation.
Calls `svm_cross_validation` and `check-parameter`.

(save-model *filename* *model*)
: Write a model into a file.
Calls `svm_save_model`.

(load-model *filename*)
: Read a model from a file.
Calls `svm_load_model`.

(get-svm-type *model*)
: Call `svm_get_svm_type`.

(get-nr-class *model*)
: Call `svm_get_nr_class`.

(get-labels *model*)
: Call `svm_get_labels`.

(get-svr-probability *model*)
: Call `svm_get_svr_probability`.

(predict-values *model* *x*)
: Return the decision values (a `(VECTOR DOUBLE-FLOAT)`) given by *model* for *x* (a `(VECTOR node)`).
Calls `svm_predict_values`.

(predict *model* *x*)
: Call `svm_predict`.

(predict-probability *model* *x*)
: Call `svm_predict_probability`.

(check-probability-model *model*)
: Call `svm_check_probability_model`.

33.10.2.3. Functions related to parameter

`(destroy-parameter parameter)`

Release the memory taken by the parameter object and invalidate the `FFI:FOREIGN-VARIABLE` *`parameter`*.
Does **not** call `svm_destroy_param`.
You do **not** have to call this function yourself, it is attached to the *`parameter`* by `make-parameter` via `EXT:FINALIZE`.

`(make-parameter &KEY :v svm_type kernel_type degree gamma coef0 cache_size eps C nr_weight weight_label weight nu p shrinking probability)`

Allocates a new `FFI:FOREIGN-VARIABLE` of type parameter with the supplied slots.
The defaults come from vector *`v`* (such as returned by `(FFI:FOREIGN-VALUE parameter)`), if supplied, providing an easy way to copy *`parameter`*s, otherwise the defaults for **svm-train** are used.

`(parameter-alist parameter)`

Return the association list representing *`parameter`*.

33.11. Computer Algebra System PARI

This package offers an **“FFI”**-based interface to PARI.

The package **“PARI”** is case-sensitive.

When this module is present, `*FEATURES*` contains the symbol `:PARI`.

PARI objects are printed and read using a special `#Z""` syntax.

33.12. Matlab Interface

This is an interface to the Matlab C API. The package **“MATLAB”** is case-sensitive, so you would write `(matlab:engOpen ...)` when you need to call `engOpen`.

When this module is present, `*FEATURES*` contains the symbol `:MATLAB`.

Additionally, some higher level functionality is available (see `modules/matlab/test.tst` for sample usage):

`(matlab:matfile-content mf)`
: Return a `VECTOR` of `STRING`s naming the variables in file *`mf`* (opened using `matOpen`).

`matlab:*command*`
: The default argument to `engOpen`.

`matlab:*engine*`
: The currrently open Matlab engine.

`(matlab:engine)`
: Make sure `*engine*` is valid and return it.

`(matlab:with-engine (&OPTIONAL engine command) &BODY body)`
: Run the *`body`* wuth the *`engine`* bound to a Matlab engine (default `*engine*`). The engine is opened with `engOpen`, then closed with `engClose`.

`(matlab:with-MATfile (file name &OPTIONAL mode) &BODY body)`
: `matOpen` the matlab file, do the *`body`*, `matClose` it.

`(matlab:copy-lisp-to-mxArray lisp-array &OPTIONAL matlab-matrix)`
: Copy data from the 2-dimensional lisp array to the Matlab matrix.

`(matlab:copy-lisp-to-matlab lisp-array matlab-variable &KEY engine)`
: Copy the 2-dimensional lisp array to the Matlab variable (a `STRING`) in the supplied engine (defaults to `*engine*`).

`(matlab:copy-mxArray-to-lisp matlab-matrix &OPTIONAL lisp-array)`
: Copy the matlab matrix to the 2-dimensional lisp array (created anew or re-used if supplied).

`(matlab:copy-matlab-to-lisp matlab-variable &OPTIONAL lisp-array &KEY engine)`
: Copy data from the matlab variable to the 2-dimensional lisp array (created anew or re-used if supplied).

`(matlab:invert-matrix lisp-array &KEY engine)`
: Invert the lisp matrix using the specified engine.

33.13. Netica Interface

This is an interface to the Netica C API for working with Bayesian belief networks and influence diagrams.

The package **"NETICA"** is case-sensitive, e.g., you would write `(netica:GetNodeExpectedUtils_bn ...)` when you need to call `GetNodeExpectedUtils_bn`.

When this module is present, `*FEATURES*` contains the symbol `:NETICA`.

An interface to all public **C** functions is provided. Additionally, some higher level functionality is available (see `modules/netica/demo.lisp` for sample usage):

`(netica:start-netica &KEY :license :verbose)`
: Call `NewNeticaEnviron_ns` and `InitNetica_bn` and print some statistics; initialize `netica:*env*`.

`(netica:check-errors &KEY :env :clear :severity)`
: Show and, optionally, clear (`ClearError_ns`), the errors of the given severity (`ErrorSeverity_ns`) and above. You should call this function after *every* call to a Netica function. Every wrapper function in this list calls it, so you do **not** need to call it after a call to a wrapper function.

`(netica:error-message error)`
: Convert netica error to a `STRING` containing

  `ErrorCategory_ns`
  `ErrorSeverity_ns`
  `ErrorNumber_ns`
  `ErrorMessage_ns`

`(netica:close-netica &KEY :env :verbose)`
: Terminate the netica session. Sets `netica:*env*` to `NIL`.

`(netica:make-net`
`&KEY :name :comment :title :env :verbose)`
: Call `NewNet_bn`, `SetNetTitle_bn` and `SetNetComment_bn`.

`(netica:net-info net &KEY :out)`
: Print some information about the net:

  `GetNetName_bn`
  `GetNetTitle_bn`
  `GetNetComment_bn`

GetNetFileName bn
GetNetNodes bn

(netica:make-node &KEY :name :net :kind :levels :states :num-states :title :comment :parents :cpt :x :y :env :verbose)
Call NewNode bn with the given name and many other parameters.

(netica:node-info node &KEY :header :out)
Print some information about the node, preceded by the header.

(netica:get-beliefs node &KEY :env :verbose)
Call GetNodeBeliefs bn on the node.

(netica:enter-finding net node state &KEY :env :verbose)
Call EnterFinding bn using NodeNamed bn and StateNamed bn.

(netica:save-net net &KEY :file :env :verbose)
Call WriteNet bn.

(netica:read-net file &KEY :env :verbose)
Call ReadNet bn.

(netica:with-open-dne-file (var file &REST opts) &BODY body)
Call NewStreamFile ns, execute *body*, then DeleteStream ns - just like WITH-OPEN-STREAM.

netica:*verbose*
The log STREAM or NIL; the default value for the :VERBOSE argument (initially set to NIL).

netica:*license*
The license key provided by Norsys; the default value for the :LICENSE argument.

netica:*env*
The Netica environment object; the default value for the :ENV argument.

33.14. Perl Compatible Regular Expressions

This is an interface to Perl Compatible Regular Expressions.

When this module is present, *FEATURES* contains the symbol :PCRE.

PCRE module API

(PCRE:PCRE-VERSION)

Return version information as 3 values: descriptive `STRING` and 2 `FIXNUM`s: major and minor numbers.

`(PCRE:PCRE-CONFIG type)`

Return some information about the PCRE build configuration. *`type`* is one of

`:UTF8`

`:NEWLINE`

`:LINK-SIZE`

`:POSIX-MALLOC-THRESHOLD`

`:MATCH-LIMIT`

`(PCRE:PCRE-COMPILE string &KEY :STUDY :IGNORE-CASE :MULTILINE :DOTALL :EXTENDED :ANCHORED :DOLLAR-ENDONLY :EXTRA :NOTBOL :NOTEOL :UNGREADY :NOTEMPTY :NO-AUTO-CAPTURE)`

Compile a pattern, optionally study it.

`(PCRE:PATTERN-INFO pattern &OPTIONAL request)`

Return some information about the *`pattern`*, such as

`:OPTIONS`

`:SIZE`

`:CAPTURECOUNT`

`:BACKREFMAX`

`:FIRSTBYTE`

`:FIRSTTABLE`

`:LASTLITERAL`

`:NAMEENTRYSIZE`

`:NAMECOUNT`

`:NAMETABLE`

`:STUDYSIZE`

`(PCRE:PCRE-NAME-TO-INDEX pattern name)`

Convert the name of the sub-pattern to an index in the return vector.

**`(PCRE:PCRE-EXEC pattern string &KEY :WORK-SPACE :DFA :BOOLEAN :OFFSET :ANCHORED :NOTBOL :NOTEOL :NO`
`-SHORTEST :DFA-RESTART)`

Execute the compiled *`pattern`* against the *`string`* at the given *`offset`* with the given options. Returns `NIL` if no matches or a `VECTOR` of `LENGTH` `CAPTURECOUNT+1` of PCRE:MATCH structures, unless `:BOOLEAN` was non-`NIL`, in which case return `T` as an indicator of success, but do not allocate anything.

`:DFA` argument determines whether `pcre_dfa_exec` is used instead of `pcre_exec` (PCRE v6 and better).

`:WORK-SPACE` is only used for `:DFA` and defaults to 20.

(PCRE:MATCH-START *match*)
(PCRE:MATCH-END *match*)
Return the start and end of the *match*. SETF-able.

(PCRE:MATCH-SUBSTRING *match string*)
Return the substring of *string* bounded by *match*.

(PCRE:MATCH-STRINGS return-vector *string*)
Return all substrings for all matches found by `PCRE:PCRE-EXEC`.

(PCRE:MATCH-STRING return-vector which *string* &OPTIONAL *pattern*)
Return the substring that matches the given sub-pattern. If *which* is a name of the sub-pattern (as opposed to its number), *pattern* must be supplied.

(PCRE:PCRE-MATCHER *pattern*)
A valid value for `CUSTOM:*APROPOS-MATCHER*`.

33.15. The Wildcard Module

Wildcards, also called “Pathname Matching Notation”, describe sets of file names.

When this module is present, `*FEATURES*` contains the symbol `:WILDCARD`.

The **“WILDCARD”** package exports the following two symbols:

(WILDCARD:MATCH *pattern string* &KEY :START :END :case-insensitive). This function returns a non-`NIL` value if the *string* matches the *pattern*.

(WILDCARD:WILDCARD-MATCHER *pattern*). This function is a valid value for `CUSTOM:*APROPOS-MATCHER*`.

33.15.1. Wildcard Syntax

*****
Matches any zero or more characters.

?

Matches any one character.

[*string*]

Matches exactly one character that is a member of the STRING *string*. This is called a “character class”. As a shorthand, *string* may contain ranges, which consist of two characters with a dash between them. For example, the class *[a-z0-9_]* matches a lowercase letter, a number, or an underscore. You can negate a class by placing a #\! or #\^ immediately after the opening bracket. Thus, *[^A-Z@]* matches any character except an uppercase letter or an at sign.

\

Removes the special meaning of the character that follows it. This works even in character classes.

Note

Slash characters have no special significance in the wildcard matching, unlike in the shell (**/bin/sh**), in which wildcards do not match them. Therefore, a pattern *foo*bar* can match a file name `foo3/bar`, and a pattern *./sr*sc* can match a file name `./src/misc`.

33.16. ZLIB Interface

This is an **“FFI”**-based interface to the ZLIB.

When this module is present, `*FEATURES*` contains the symbol `:ZLIB`.

`(ZLIB:Z-VERSION)`

Return the string version of the underlying library.

`(ZLIB:COMPRESS source &KEY level)`

Compress the *source* VECTOR.

`(ZLIB:UNCOMPRESS source destination-length)`

Uncompress the *source* VECTOR (returned by `ZLIB:COMPRESS`). *destination-length* should be no less than the length of the uncompressed *source*.

`(ZLIB:COMPRESS-BOUND source-length)`

Return the maximum length of the return value of `ZLIB:COMPRESS`.

`(ZLIB:ERROR-STRING errno)`

Return a descriptive string for the supplied error code.

ZLIB:ZERROR

An ERROR sometimes SIGNALed by ZLIB:COMPRESS and ZLIB:UNCOMPRESS. You can find the error code and the caller using ZLIB:ZERROR-ERRNO and ZLIB:ZERROR-CALLER.

33.17. Raw Socket Access

33.17.1. Introduction

This is the raw socket interface, as described in <sys/socket.h>. Sockets are represented by their FIXNUM file descriptors.

When this module is present, *FEATURES* contains the symbol :RAWSOCK.

Try SOCKET:SOCKET-STREAM first!

For most uses of sockets, the facilities described in Section 32.5, “Socket Streams” are adequate and much more convenient than these. You are encouraged to consider SOCKET:SOCKET-STREAMs and ensure that they are not adequate for your purposes before you use raw sockets.

Do not use `EXT:MAKE-STREAM`!

You can turn such a raw socket into a usual lisp `STREAM` using `EXT:MAKE-STREAM`, but you should be *extremely* careful with such dubious actions! See the <`clisp-devel@lists.sourceforge.net`> (http://lists.sourceforge.net/lists/listinfo/clisp-devel) mailing list archives for more details. Note that `EXT:MAKE-STREAM` will duplicate the file descriptor (using `dup`), so you *still* have to `CLOSE` the original raw socket.

33.17.2. Single System Call Functions

We implement access to

(`accept` ***socket address***)

(`bind` ***socket address***)

(`connect` ***socket address***)

(`getaddrinfo` `&KEY` `node service protocol socktype family passive canonname numerichost numericserv v4mapped all addrconfig`)

(`getnameinfo` ***address*** `&KEY` `nofqdn numerichost namereqd numericserv numericscope dgram`)

(`getpeername` ***socket address***)

(`getsockname` ***socket address***)

(`htonl` ***n***)

(`htons` ***n***)

(`ntohl` ***n***)

(`ntohs` ***n***)

(`recv` ***socket buffer*** `&KEY` `start end peek oob waitall`)

(`recvfrom` ***socket buffer address*** `&KEY` `start end peek oob waitall`)

(`recvmsg` ***socket message*** `&KEY` `start end peek oob waitall`)

(`send` ***socket buffer*** `&KEY` `start end oob eor`)

(`sendmsg` ***socket message*** `&KEY` `start end oob eor`)

(`sendto` ***socket buffer address*** `&KEY` `start end oob eor`)

(`sockatmark` ***socket***)

```
(socket domain type protocol)
(socketpair domain type protocol)
```

using same-named lisp functions in package **“RAWSOCK”**. Additionally,

`(RAWSOCK:SOCK-CLOSE socket)` calls `close`.
`(RAWSOCK:SOCK-LISTEN socket &OPTIONAL (backlog SOMAXCONN))` calls `listen`.

Note

When the OS does not provide `socketpair`, it is emulated using `socket` + `connect` + `accept`.

33.17.3. Common arguments

33.17.3.1. Platform-dependent Keywords

void* ***`buffer`***
: A `(VECTOR (UNSIGNED-BYTE 8))`. The vector may be adjustable and have a fill pointer. Whenever a function accepts a *`buffer`* argument, it also accepts `:START` and `:END` keyword arguments with the usual meaning and defaults. You do not have to supply the vector length because Lisp can determine it itself, but, if you want to, you can use `:END` argument for that.

int ***`socket`***
: An `INTEGER` (returned by `socketpair` or `socket`).

int ***`family`***
int ***`domain`***
: A `NIL` (stands for `AF_UNSPEC`), `INTEGER`, or a platform-specific keyword, e.g., `:INET` stands for `AF_INET`.

int ***`type`***
: A `NIL` (stands for 0); `INTEGER`; or a platform-specific keyword, e.g., `:DGRAM` stands for `SOCK_DGRAM`.

int ***`protocol`***

A `NIL` (stands for 0); `INTEGER`; a platform-specific keyword, e.g., `:ETH_P_ARP` stands for `ETH_P_ARP`, `:IPPROTO-ICMP` stands for `IPPROTO_ICMP`; or a `STRING` (passed to `getprotobyname`).

int *flags*

This **C** argument corresponds to keyword arguments to the Lisp functions. E.g., `rawsock:send` accepts `:OOB` and `EOR` arguments, while `rawsock:recv` accepts `PEEK`, `OOB` and `WAITALL`.

struct sockaddr *address*

A `STRUCTURE-OBJECT` RAWSOCK:SOCKADDR returned by `MAKE-SOCKADDR`. You do not need to supply its length because Lisp can determine it itself.

struct msghdr *message*

A `STRUCTURE-OBJECT` RAWSOCK:MESSAGE with the following slots:

addr a SOCKADDR.

iovec a `(VECTOR (VECTOR (UNSIGNED-BYTE 8)))` (`:START` and `:END` arguments are applied to this vector)

control a `(VECTOR (UNSIGNED-BYTE 8))`

flags a `LIST`

33.17.3.1. Platform-dependent Keywords

One can extract the list of acceptable platform-dependent keywords for, e.g., socket domain, using the following code:

```
(BLOCK NIL
  (HANDLER-BIND ((TYPE-ERROR
                   (LAMBDA (c)
                     (FORMAT T "~&error: ~A~%" c)
                     (RETURN (CDDR (THIRD (TYPE-ERROR-EX
    (rawsock:socket "bad" NIL NIL)))
```

33.17.4. Return Values

The return values of the functions described in section Section 33.17.2, “Single System Call Functions” are derived from the return values of the underlying system call: if, say, the *address* argument is modified by the system call, two values are returned (in addition to the possible values coming from the return value of the system call): the (modified) *address* structure and its new size. If the system call fails, an ERROR is SIGNALed.

33.17.5. Not Implemented

We do not interface to select or poll in this module, they are already available through SOCKET:SOCKET-STATUS.

We do not interface to shutdown in this module, it is already available through SOCKET:SOCKET-STREAM-SHUTDOWN.

We do not interface to gethostbyname or gethostbyaddr in this module, they are already available through POSIX:RESOLVE-HOST-IPADDR.

33.17.6. Errors

Errors in getaddrinfo and getnameinfo are SIGNALed as CONDITIONs of type RAWSOCK:EAI using gai_strerror.

Errors in other functions are reported as the usual OS errors (using strerror).

33.17.7. High-Level Functions

Functions that do not correspond to a single system call

(RAWSOCK:SOCK-READ *socket buffer* &KEY start end)
(RAWSOCK:SOCK-WRITE *socket buffer* &KEY start end)

Call one of read/readv or write/writev (depending on whether *buffer* is a (VECTOR (UNSIGNED-BYTE 8)) or a (VECTOR (VECTOR (UNSIGNED-BYTE 8)))). Return the number of bytes read or written.
When readv and writev and not available, they are emulated by repeated calls to read and write.

On **Win32** we have to use `recv` instead of `read` and `send` instead of `write` because **Win32** `read` and `write` do not work on sockets, only on regular files.

`(RAWSOCK:PROTOCOL &OPTIONAL protocol)`

Call `getprotobyname` when *`protocol`* is a `STRING`, or call `getprotobynumber` when *`protocol`* is an `INTEGER`. Return a RAWSOCK:PROTOCOL structure object. When *`protocol`* is `NIL`, return a `LIST` of all known protocols using `setprotoent`, `getprotoent`, and `endprotoent`.

`(RAWSOCK:NETWORK &OPTIONAL network type)`

Call `getnetbyname` when *`network`* is a `STRING`, or call `getnetbynumber` when *`network`* is an `INTEGER`. Return a RAWSOCK:NETWORK structure object. When *`network`* is `NIL`, return a `LIST` of all known networks using `setnetent`, `getnetent`, and `endnetent`.

`(RAWSOCK:IF-NAME-INDEX &OPTIONAL what)`

Call `if_nametoindex` when *`network`* is a `STRING` and return an `INTEGER`; or call `if_indextoname` when *`network`* is an `INTEGER` and return a `STRING`. When *`what`* is `NIL`, return an association list of pairs `(`*`index . name`*`)` using `if_nameindex`.

`(RAWSOCK:IFADDRS)`

Call `getifaddrs` and return a `LIST` of ifaddrs objects.

`(RAWSOCK:SOCKET-OPTION socket name &KEY :LEVEL)`
`(SETF (RAWSOCK:SOCKET-OPTION socket name &KEY :LEVEL) value)`

Call `getsockopt` and `setsockopt`, returns and sets individual (for specific option *`name`* and *`level`*) and multiple (when *`name`* is `NIL` and/or *`level`* is `:ALL`) options. (See also `SOCKET:SOCKET-OPTIONS`.)

`(RAWSOCK:CONVERT-ADDRESS family address)`

Convert between `STRING` and `INTEGER` IP *`address`* representations using

`inet_addr`	`inet_ntop`
`inet_ntoa`	`inet_pton`

`(RAWSOCK:MAKE-SOCKADDR family &OPTIONAL data)`

Create a sockaddr object. *`data`* should be a sequence of `(UNSIGNED-BYTE 8)` or an `INTEGER` (meaning `(MAKE-LIST `*`data`*` :initial-element 0)`). When omitted, the standard platform-specific size is used.

`(RAWSOCK:SOCKADDR-FAMILY address)`

Return the numeric *`family`* of the sockaddr object.

(RAWSOCK:SOCKADDR-DATA *address*)
: Return a fresh `VECTOR` displaced to the *`data`* field of the **C** struct sockaddr object.

Warning

Modifying this `VECTOR`'s content will modify the *`address`* argument data!

(RAWSOCK:OPEN-UNIX-SOCKET *pathname* &OPTIONAL (*type* :STREAM))
: Open a **UNIX** socket special file. Returns two values: *`socket`* and *`address`*.

(RAWSOCK:OPEN-UNIX-SOCKET-STREAM *pathname* &REST *options* &KEY (*type* :STREAM) &ALLOW-OTHER-KEYS)
: Open a **UNIX** socket special file. Returns two values: *`stream`* and *`address`*. *`type`* is passed to `RAWSOCK:OPEN-UNIX-SOCKET`, other *`options`* to `EXT:MAKE-STREAM` (but see Do not use `EXT:MAKE-STREAM`!).

(RAWSOCK:IPCSUM *buffer* &KEY start end) - **IP**
(RAWSOCK:ICMPCSUM *buffer* &KEY start end) - **ICMP**
(RAWSOCK:TCPCSUM *buffer* &KEY start end) - **TCP**
(RAWSOCK:UDPCSUM *buffer* &KEY start end) - **UDP**
: Compute the appropriate protocol checksum and record it in the appropriate location. *`buffer`* is assumed to be a suitable packet for the protocol, with the appropriate header etc. The typical packet you send is both **IP** and **TCP** and thus has two checksums, so you would want to call *two* functions.

(RAWSOCK:CONFIGDEV *socket name address* &KEY *promisc noarp*)
: Set some socket options and IP *`address`* with `ioctl`.

33.18. The FastCGI Interface

The FastCGI module speeds up **CLISP** CGI scripts launched by a Web server. Working with a FastCGI-enabled Web server such as Apache with mod_fastcgi, a **CLISP** program using the FastCGI protocol will run many times faster than a conventional CGI program. The performance improvements stem from the fact that the script's process remains running across **HTTP** requests, eliminating startup overhead and allowing for caching of data structures and other resources. This is the same approach used is in other languages (e.g., mod_perl for Perl).

When this module is present, `*FEATURES*` contains the symbol `:FASTCGI`.

33.18.1. Overview of FastCGI

Traditional CGI programs work by doing input/output with the Web server via the following channels:

1. Examining environment variables; e.g., `HTTP_USER_AGENT` is the variable set by the Web server to name the browser used
2. Reading from standard input. E.g., to get input data in a "method=POST" request
3. Writing an **HTTP** response document (usually "Content-type: text/html") to the standard output, for eventual transmission back to the browser client
4. Writing error messages to the standard error, usually captured by the Web server and logged in its log files.

FastCGI involves replacing calls the standard routines to do the above with calls in the **“FASTCGI”** package. These calls will then work exactly as before when the program is invoked as a CGI, but will also work when invoked by a FastCGI-enabled Web server.

FastCGI programs persist across **HTTP** requests, and thus incur startup overhead costs only once. For Lisp Web programs, this overhead can be substantial: code must be compiled and loaded, files and databases must be opened, etc. Further, because the program stays running from **HTTP** request to **HTTP** request, it can cache information in memory such as database connections or large in-memory data structures.

33.18.2. Functions in Package “FASTCGI”

Access to FastCGI is via these functions in package **“FASTCGI”**.

`(FASTCGI:IS-CGI)`

Returns `T` if the **CLISP** program has been launched as a traditional CGI rather than in FastCGI. In traditional CGI, program I/O is via operating system environment variables and standard file streams. Under FastCGI, I/O is done directly with the Web server via the FastCGI protocol.

`(FASTCGI:ACCEPT)` ***`cgi-forms`*** **`(FASTCGI:FINISH)`**

In FastCGI mode, the program loops, `ACCEPT`ing to begin the execution of an **HTTP** request, and `FINISH`ing to signal that the script is finished writing its response to the **HTTP** request. `ACCEPT` blocks until the next **HTTP** request comes in, returning `T` if there is a new request to handle, and `NIL` if no more **HTTP** requests will occur, usually because the Web server itself has terminated, in which case the FastCGI server loop should also exit.

A typical FastCGI top-level server loop looks like:

```
(do ()
    ((not (fastcgi:accept)))
  (run-my-script)
  (fastcgi:finish))
```

`(FASTCGI:GETENV` ***`varname`*** **`)`**

Use in place of `EXT:GETENV` to get the value of the environment variable named *`varname`*, which should be a string. Unlike `EXT:GETENV`, which accesses the actual host operating system environment, `FASTCGI:GETENV` obtains its environment via the Web server, over its FastCGI communications channel. For more information, see the FastCGI Web site. Returns `NIL` if *`varname`* is not defined in the operating system environment. See here for a list of useful variables. You must first have called `ACCEPT` and not yet have called `FINISH`.

`(FASTCGI:WRITE-STDOUT` ***`string`*** **`)`**

Use in place of standard Lisp calls which print to standard output (i.e., as part of the **HTTP** response). You must first have called `ACCEPT` and not yet have called `FINISH`.

`(FASTCGI:WRITE-STDERR` ***`string`*** **`)`**

Use in place of standard Lisp calls which print to standard error. Rather than being part of the **HTTP** response, data written to standard error are usually collected by the Web server in its error log. This is useful for diagnostic purposes.

`(FASTCGI:SLURP-STDIN)`

Reads in the entirety of standard input and returns it as a string. This is usually done for **HTTP** requests with `METHOD="post"`, when the data are passed to the CGI script via standard input rather than via the environment variable `QUERY_STRING`. There is no way to read standard input in pieces, which could be a problem, say, for **HTTP** uploads of very large files.

`(FASTCGI:OUT` *`tree`*`)`

Like `WRITE-STDOUT`, except that *`tree`* may be an arbitrarily nested list structure containing (at the leaves) numbers and strings. For example, `(FASTCGI:OUT '("foo" (" " 10 " " 20)))` will write the string `"foo 10 20"`. This function is useful when building strings in memory for display.

33.18.3. FastCGI Example

Below is a simple example CGI script using FastCGI.

```
#!/usr/local/bin/clisp -q -K full

(do ((count 1 (1+ count)))
    ((not (fastcgi:accept)) nil)
  (fastcgi:out "Content-type: text/plain" #\Newline #\New
  (fastcgi:out
   "I am running in mode: " (if (fastcgi:is-cgi) "CGI" "F
   "This is execution no.: " count #\Newline
   "The browser string is '" (fastcgi:getenv "HTTP_USER_A
  (fastcgi:finish))
```

33.18.4. Building and configuring the FastCGI Interface

It is necessary to download the FastCGI developers' kit, build it, and install it, before building **CLISP** with FastCGI support. You also need to upgrade your Web server to speak the FastCGI protocol. For Apache this means building in mod_fastcgi, either statically or dynamically, and then adding a line to your Apache config like:

```
Addhandler fastcgi-script .fcgi
```

After that, you can convert `foo.cgi` by linking it to a script names `foo.fcgi`. Since a FastCGI script is also a valid CGI script, it can be run unmodified in either mode.

33.19. GTK Interface

This is an **“FFI”**-based interface to GTK+ version 2.

The package **“GTK”** is case-sensitive.

When this module is present, `*FEATURES*` contains the symbol `:GTK`.

33.19.1. High-level functions

`(glade-load *filename*)`
: Load and connect the UI described in the Glade-generated file *`filename`*.

`(run-glade-file *filename* *name*)`
: Run the widget *`name`* described in the Glade-generated file *`filename`*.

`(gui *filename*)`
: Run the **CLISP** demo GUI described in the Glade-generated file *`filename`*, normally a variation of `modules/gtk2/ui.glade`.

Part IV. Internals of the CLISP Implementation

Table of Contents

Chapter 34. The source files of CLISP

Table of Contents

For files in **CLISP** binary distributions, see the section called “Files”.

34.1. File Types

`#P"*.d"`
The source files for unpreprocessed **C** code.

`#P".c"`
The **C** code after preprocessing; also the result of compiling some **“FFI”** forms (see `FFI:*OUTPUT-C-FUNCTIONS*`).

`#P".lisp"`
The source files for Lisp code.

`#P"*.fas"`
Compiled lisp code (platform-independent bytecodes).

`#P".lib"`
Lisp “header”, produced by `COMPILE-FILE` and used by `REQUIRE`

34.2. Source Pre-Processing

C sources are pre-processed with the following tools before being passed to the **C** compiler:

`utils/comment5.c`
Convert **/bin/sh**-style comments (lines starting with `"# "`) to **C**-style comments (`/**/`).

Warning

The use of **/bin/sh**-style comments is deprecated.

`utils/varbrace.d`
Add braces to **C** source code, so that variable declarations (introduced with the pseudo-keyword `var`) can be used within blocks, like in **C++** and **C99**.

`utils/ccpaux.c`

When **cpp** cannot handle indented directives, remove the indentation.

`utils/gctrigger.d`
: Add `GCTRIGGER` statements at the head of function bodies (for functions marked with the `maygc` pseudo-keyword).

`utils/deema.c`
: When **cpp** cannot handle empty macro arguments, insert `_EMA_` instead.

`utils/ccmp2c.c`
: For the **`clx/new-clx`** module only. Allows **cpp**-style preprocessing **before** **modprep** processing. Should be merged into **modprep** eventually.

`utils/modprep.lisp`
: For some modules only, see Section 32.2.7.1, "Modprep".

34.3. Files

34.3.1. Unpreprocessed C code

34.3.1.1. Includes

`src/lispbibl.d`
: main include file

`src/fsubr.d`
: list of all built-in special forms

`src/subr.d`
: list of all built-in functions

`src/pseudofun.d`
: list of all “pseudo functions”

`src/constpack.d`
: list of packages accessed by **C** code

`src/constsym.d`
: list of symbols accessed by **C** code

`src/constobj.d`
: list of miscellaneous objects accessed by **C** code

`src/unix.d`
: include file for the **UNIX** implementations

`src/win32.d`
: include file for the **Win32** based versions

`src/xthread.d`
: include file for thread support

`src/modules.h`
: list of foreign modules

34.3.1.2. Internal C Modules

`src/spvw.d`
: Memory management (garbage-collection), startup; some OS interface.

`src/avl.d`
: An implementation of AVL (Adelson-Velskii and Landis) trees.

`src/sort.d`
: A sorting routine.

`src/subrkw.d`
: The list of all built-in functions with keywords in lambda list.

`src/spvwtabf.d`
: The table of built-in special operators and functions.

`src/spvwtabs.d`
: The table of all `SYMBOL`s accessed by **C** code.

`src/spvwtabo.d`
: The table of miscellaneous objects accessed by **C** code.

`src/eval.d`
: Evaluator (form interpreter) and bytecode interpreter.

`bytecode.d`
: List of bytecodes.

`src/control.d`
: Special operator interpreter.

`src/pathname.d`
: Pathnames, file- and directory-related functions.

`src/stream.d`
: `STREAM`s of all kinds: `FILE-STREAM`s, terminal streams, `STRING-STREAM`s etc.

`src/socket.d`
: Opening sockets for TCP/IP and CLX.

`src/io.d`
: The lisp reader (parser) and printer (also pretty printer).

`src/array.d`
: Functions dealing with `ARRAY`s and `VECTOR`s.

`src/hashtabl.d`
: Functions dealing with `HASH-TABLE`s.

`src/list.d`
: Functions dealing with `LIST`s.

`src/package.d`
: Functions dealing with `PACKAGE`s.

`src/record.d`
: Functions dealing with records (structures, closures, etc.)

`src/sequence.d`
: The generic `SEQUENCE` functions.

`src/charstrg.d`

Functions dealing with CHARACTERs and STRINGs.

src/debug.d
Support for debugging and the read-eval-print loop (see Section 25.1, "Debugging Utilities [CLHS-25.1.2]").

src/error.d
ERROR handling and SIGNALing.

src/errunix.d
UNIX-specific error messages.

src/errwin32.d
Win32-specific error messages.

src/misc.d
Miscellaneous functions.

src/time.d
Timing functions.

src/predtype.d
Predicates, type tests.

src/symbol.d
Functions dealing with SYMBOLs.

src/unixaux.d
Auxiliary functions (**UNIX** version only).

src/win32aux.d
Auxiliary functions (**Win32** version only).

src/foreign.d
"FFI" support.

src/lisparit.d
Functions dealing with numbers (arithmetic), see Section 34.3.1.3, "Number system (arithmetic)".

src/noreadline.d
Dummy plug-in for the GNU readline library.

34.3.1.3. Number system (arithmetic)

src/lisparit.d
initialization, input/output of numbers, lisp functions

src/aridecl.d
declarations

`src/arilev0.d`
: arithmetic at the machine level

`src/arilev1.d`
: digit sequences

`src/arilev1c.d`
: operations on digit sequences, written in **C**

`src/arilev1i.d`
: operations on digit sequences, as inline functions

`src/arilev1e.d`
: operations on digit sequences, bindings to external routines

`src/intelem.d`
: `INTEGER`s: elementary operations

`src/intlog.d`
: `INTEGER`s: logical connectives

`src/intplus.d`
: `INTEGER`s: addition and subtraction

`src/intcomp.d`
: `INTEGER`s: comparison

`src/intbyte.d`
: `INTEGER`s: byte operations `LDB`, `DPB`

`src/intmal.d`
: `INTEGER`s: multiplication

`src/intdiv.d`
: `INTEGER`s: division

`src/intgcd.d`
: `INTEGER`s: `GCD` and `LCM`

`src/int2adic.d`
: `INTEGER`s: operations on 2-adic integers

`src/intsqrt.d`
: `INTEGER`s: square root, n-th root

`src/intprint.d`
: subroutines for `INTEGER` output

`src/intread.d`
: subroutines for `INTEGER` input

`src/rational.d`
: rational numbers (`RATIO`s)

`src/sfloat.d`
: elementary operations for `SHORT-FLOAT`s

`src/ffloat.d`
: elementary operations for `SINGLE-FLOAT`s

`src/dfloat.d`
: elementary operations for `DOUBLE-FLOAT`s

`src/lfloat.d`

elementary operations for LONG-FLOATs

src/flo_konv.d
: conversions between FLOATs

src/flo_rest.d
: general FLOAT operations

src/realelem.d
: elementary functions for REAL numbers

src/realrand.d
: random numbers

src/realtran.d
: transcendental functions for REAL numbers

src/compelem.d
: elementary functions for COMPLEX numbers

src/comptran.d
: transcendental functions for COMPLEX numbers

34.3.1.3.1. External routines for the arithmetic system, written in assembly language

src/ari68000.d
: written in 68000 assembler, MIT syntax

src/ari68020.d
: written in 68020 assembler, MIT syntax

src/arisparc.d
: written in SPARC assembler

src/arisparc64.d
: written in 64-bit SPARC assembler

src/ari80386.d
: written in i386/i486 assembler

src/arimips.d
: written in MIPS assembler

src/arimips64.d
: written in 64-bit MIPS assembler

src/arihppa.d
: written in HPPA-1.0 assembler

src/arivaxunix.d
: written in VAX assembler, Unix assembler syntax

src/ariarm.d
: written in ARM assembler

34.3.1.4. External routines for accessing the stack, written in assembly language

`src/sp68000.d`
: written in 68000 assembler, MIT syntax

`src/spsparc.d`
: written in SPARC assembler

`src/spsparc64.d`
: written in 64-bit SPARC assembler

`src/sp80386.d`
: written in i386/i486 assembler

`src/spmips.d`
: written in MIPS assembler

34.3.2. Other assembly language stuff

`src/asmi386.sh`
: converts i386 assembler from MIT syntax to a macro syntax

`src/asmi386.hh`
: expands i386 assembler in macro syntax to either MIT or Intel syntax

34.3.3. Lisp source files

`src/init.lisp`
: the first file to be loaded during bootstrapping, loads everything else

`src/defseq.lisp`
: defines the usual sequence types for the generic sequence functions

`src/backquote.lisp`
: implements the backquote read macro

`src/defmacro.lisp`
: implements `DEFMACRO`

`src/macros1.lisp`
: the most important macros

`src/macros2.lisp`
: some other macros

`src/defs1.lisp`
: miscellaneous definitions

`src/timezone.lisp`

site-dependent definition of time zone, except for **UNIX** and **Win32**.

`src/places.lisp`
: macros using places, definitions of most standard and extensiion places

`src/floatprint.lisp`
: defines `SYS::WRITE-FLOAT-DECIMAL` for printing floating point numbers in base 10

`src/type.lisp`
: functions working with type specifiers: `TYPEP`, `SUBTYPEP`

`src/defstruct.lisp`
: implements the macro `DEFSTRUCT`

`src/format.lisp`
: implements the function `FORMAT`

`src/room.lisp`
: implements the function `ROOM` (see also Section 25.2.7, “Function `ROOM`”)

`src/savemem.lisp`
: see Section 31.2, “Saving an Image”

`src/keyboard.lisp`
: implements the macro `EXT:WITH-KEYBOARD`

`src/runprog.lisp`
: implements the functions `EXT:RUN-PROGRAM`, `EXT:RUN-SHELL-COMMAND` etc.

`src/query.lisp`
: implements the functions `Y-OR-N-P` and `YES-OR-NO-P`

`src/reploop.lisp`
: support for debugging and the read-eval-print loop (see Section 25.1, “Debugging Utilities [CLHS-25.1.2]”)

`src/dribble.lisp`
: implements the functions `DRIBBLE` and `EXT:DRIBBLE-STREAM`

`src/complete.lisp`
: implements completion, see Section 21.2, “Terminal interaction”.

`src/describe.lisp`
: implements functions `DESCRIBE`, `APROPOS`, `APROPOS-LIST`

`src/trace.lisp`
: tracer

`src/macros3.lisp` (optional)
: the macros `EXT:LETF`, `EXT:LETF*` and `EXT:ETHE`

`src/config.lisp`
: (user written) site-dependent configuration, may be a link to one of the following:

src/cfgsunux.lisp
for **UNIX**, using SunOS

src/cfgunix.lisp
for any other **UNIX**

src/cfgwin32.lisp
for the **Win32**

See Section 31.12, “Customizing **CLISP** behavior”.

src/compiler.lisp
compiles Lisp code to bytecode

src/disassem.lisp
the function `DISASSEMBLE`

src/defs2.lisp
miscellaneous [ANSI CL standard] definitions

src/loop.lisp
implements the [ANSI CL standard]-compatible `LOOP` macro

src/clos.lisp
loads the various parts of the **CLOS**:

src/clos-package.lisp
declares the imports and exports of the **“CLOS”** package

src/clos-macros.lisp
defines some internal macros used by the **CLOS** implementation

src/clos-class0.lisp
defines the `class-version` structure

src/clos-metaobject1.lisp
defines the `CLOS:METAOBJECT` class

src/clos-slotdef1.lisp
defines the `CLOS:SLOT-DEFINITION` class and its subclasses

src/clos-slotdef2.lisp
defines `INITIALIZE-INSTANCE` methods for `CLOS:SLOT-DEFINITION` and its subclasses

src/clos-slotdef3.lisp
defines the generic functions that can be used on `CLOS:SLOT-DEFINITION` objects

src/clos-stablehash1.lisp
defines the `EXT:STANDARD-STABLEHASH` class

src/clos-stablehash2.lisp
defines `INITIALIZE-INSTANCE` methods for `EXT:STANDARD-STABLEHASH`

src/clos-specializer1.lisp
defines the `CLOS:SPECIALIZER` class and its subclasses

src/clos-specializer2.lisp
: defines `INITIALIZE-INSTANCE` methods for `CLOS:SPECIALIZER` and its subclasses

src/clos-specializer3.lisp
: defines the generic functions that can be used on `CLOS:SPECIALIZER` objects

src/clos-class1.lisp
: defines the `potential-class` class and its subclasses

src/clos-class2.lisp
: implements the mapping from class names to classes

src/clos-class3.lisp
: implements the `DEFCLASS` macro, class definition and class redefinition

src/clos-class4.lisp
: defines `INITIALIZE-INSTANCE` methods for `potential-class` and its subclasses

src/clos-class5.lisp
: implements the special logic of `MAKE-INSTANCE`, `INITIALIZE-INSTANCE` etc.

src/clos-class6.lisp
: defines the generic functions that can be used on `potential-class` objects

src/clos-method1.lisp
: defines the `METHOD` class and its subclasses

src/clos-method2.lisp
: implements the bulk of `DEFMETHOD`

src/clos-method3.lisp
: defines the generic functions that can be used on `METHOD` objects

src/clos-method4.lisp
: makes generic functions on `STANDARD-METHOD` objects extensible

src/clos-methcomb1.lisp
: defines the `METHOD-COMBINATION` class

src/clos-methcomb2.lisp
: implements method combination (part 2 of generic function dispatch and execution) and the `DEFINE-METHOD-COMBINATION` macro

src/clos-methcomb3.lisp
: defines `INITIALIZE-INSTANCE` methods for `METHOD-COMBINATION`

src/clos-methcomb4.lisp

makes generic functions on `METHOD-COMBINATION` objects extensible

`src/clos-genfun1.lisp`
: defines the `GENERIC-FUNCTION` class and its metaclass, superclass and subclasses

`src/clos-genfun2a.lisp`
: implements part 1 of generic function dispatch and execution

`src/clos-genfun2b.lisp`
: implements part 3 of generic function dispatch and execution

`src/clos-genfun3.lisp`
: implements creation of generic function objects, `DEFMETHOD`, `DEFGENERIC`

`src/clos-genfun4.lisp`
: defines `INITIALIZE-INSTANCE` methods for `GENERIC-FUNCTION` and its subclasses

`src/clos-genfun5.lisp`
: makes generic functions on `GENERIC-FUNCTION` objects extensible

`src/clos-slots1.lisp`
: implements low-level slot access, `WITH-SLOTS`, `WITH-ACCESSORS`

`src/clos-slots2.lisp`
: defines the generic functions that deal with slot access

`src/clos-dependent.lisp`
: implements notification from metaobjects to dependent objects

`src/clos-print.lisp`
: implements the function `PRINT-OBJECT`

`src/clos-custom.lisp`
: provides user customization of the **CLOS**

`src/condition.lisp`
: implements the **Common Lisp** Condition System (**CLCS**)

`src/gstream.lisp`
: generic stream default methods

`src/foreign1.lisp`
: **“FFI”** interface

`src/screen.lisp`
: the screen access package, see Section 32.1, “Random Screen Access”

`src/edit.lisp` **(optional)**
: the screen editor (`ED`), `EXT:UNCOMPILE`

`src/inspect.lisp`

implements `INSPECT` (tty and **HTTP** frontends)

`src/clhs.lisp`
implements `EXT:OPEN-HTTP`, `EXT:BROWSE-URL`

`src/exporting.lisp`
Macros that export their definienda, see Section 32.2.7.3, “Exporting”.

`src/threads.lisp`
MT interface

`src/spanish.lisp`
`src/german.lisp`
`src/french.lisp`
`src/russian.lisp`
`src/dutch.lisp`
i18n user messages

34.3.4. External Modules

`modules/`
individual external module sources

34.3.5. Documentation

`src/NEWS`
the list of the user-visible changes

`src/_README`
master for the distribution's README

`src/_README.en`
`src/_README.de`
`src/_README.es`
translations of `src/_README`

`doc/clisp.xml.in`
DocBook/XML sources for the **CLISP** manual page

***`build-dir`*`/clisp.1`**
the platform-specific **man** manual page, generated from `doc/clisp.xml.in` at *build* time

***`build-dir`*`/clisp.html`**
the platform-specific HTML manual page, generated from `doc/clisp.xml.in` at *build* time

`doc/impnotes.xml.in`

the master DocBook/XML file for these implementation notes; includes the following files

`doc/cl-ent.xml`
: **CLISP**-independent general **Common Lisp**-related entities

`doc/clhs-ent.xml`
: generated list of [Common Lisp HyperSpec] entities

`doc/impent.xml`
: **CLISP**-specific entities

`doc/unix-ent.xml`
: **UNIX**-related entities

`doc/mop-ent.xml`
: Meta-Object Protocol-related entities

`doc/impbody.xml`
: most of Part I, "Chapters or the Common Lisp HyperSpec"

`doc/impissue.xml`
: Chapter 28, *X3J13 Issue Index [CLHS-ic]*

`doc/gray.xml`
: Chapter 30, *Gray streams*

`doc/mop.xml`
: Chapter 29, *Meta-Object Protocol*

`doc/impext.xml`
: Chapter 31, *Platform Independent Extensions* and Chapter 32, *Platform Specific Extensions*

`doc/impbyte.xml`
: this Part IV, "Internals of the **CLISP** Implementation"

`doc/faq.xml`
: Appendix A, *Frequently Asked Questions (With Answers) about* ***CLISP***

`modules/**/*.xml`
: individual external module documentation

`doc/Symbol-Table.text`
: the mapping between lisp symbols and element IDs in these notes (see `DESCRIBE`).

`doc/impnotes.html`
: these HTML implementation notes, generated from `doc/impnotes.xml.in` at *release* time

34.3.6. Internationalization

`src/po/*.pot`

list of translatable messages ("portable object template")

`src/po/*.po`
: translated messages ("portable objects")

`src/po/*.gmo`
: translated messages ("GNU format message objects")

34.3.7. Automatic configuration on UNIX

`src/configure.in`
: lists features to be checked

`src/autoconf/autoconf.m4`
: **autoconf**'s driver macros. Part of GNU autoconf 2.57

`src/m4/`
: a repertoire of features. Use with GNU autoconf 2.57

`src/configure`
: configuration script, generated from `src/configure.in`

`src/intparam.c`
: figures out some machine parameters (word size, endianness etc.)

`src/floatparam.c`
: figures out some floating point arithmetics parameters (rounding, epsilons etc.)

`src/config.h.in`
: header file master, generated from `src/configure.in`. *`build-dir`*`/config.h` contains the values of the features discovered by `src/configure`.

`src/makemake.in`
: makefile construction script master

`src/_clisp.c`
: master for the distribution's driver program

`src/_distmakefile`
: master for the distribution's Makefile

Chapter 35. Overview of CLISP's Garbage Collection

Table of Contents

Abstract

These are internals, which are of interest only to the **CLISP** developers. If you are not subscribed to <`clisp-devel@lists.sourceforge.net`> (http://lists.sourceforge.net/lists/listinfo/clisp-devel), this chapter is probably not for you.

35.1. Introduction

Knowing that most `malloc` implementations are buggy and/or slow, and because **CLISP** needs to perform garbage-collection, **CLISP** has its own memory management subsystem in files `src/spvw*.d`, see Section 34.3.1.2, “Internal **C** Modules”.

35.2. Lisp objects in CLISP

Three kinds of storage are distinguished:

1. **CLISP** data (the “heap”), i.e. storage which contains **CLISP** objects and is managed by the garbage-collector.
2. **CLISP** stack (called `STACK`), contains **CLISP** objects visible to the garbage-collector
3. **C** data (including program text, data, `malloc`ed memory)

A **CLISP** object is one word, containing a tag (partial type information) and either immediate data or a pointer to storage. Pointers to **C** data have tag = `machine_type` = 0, pointers to **CLISP** stack have tag = `system_type`, most other pointers point to **CLISP** data.

Immediate objects

32-bit CPU

FIXNUM

SHORT-FLOAT

CHARACTER

64-bit CPU

In addition to the above,

SINGLE-FLOAT (with **TYPECODES**)

Let us turn to those **CLISP** objects that consume regular **CLISP** memory. Every **CLISP** object has a size which is determined when the object is allocated (using one of the `allocate_*()` routines). The size can be computed from the type tag and - if necessary - the length field of the object's header. The length field always contains the number of elements of the object. The number of bytes is given by the function `objsize()`.

CLISP objects which contain exactly 2 **CLISP** objects (i.e. CONSes, COMPLEX numbers, RATIOs) are stored in a separate area and occupy 2 words each. All other **CLISP** objects have “varying” length (more precisely, not a fixed length) and include a word for garbage-collection purposes at their beginning.

The garbage collector is invoked by `allocate_*()` calls according to certain heuristics. It marks all objects which are "live" (may be reached from the "roots"), compacts these objects and unmarks them. Non-live objects are lost; their storage is reclaimed.

2-pointer objects are compacted by a simple hole-filling algorithm: fill the left-most object into the right-most hole, and so on, until the objects are contiguous at the right and the hole is contiguous at the left.

Variable-length objects are compacted by sliding them down (their address decreases).

35.3. Object Pointer Representations

CLISP implements two ways of representing object pointers. (An object pointer, **C** type object, contains a pointer to the memory location of the object, or - for immediate object - all bits of the object itself.) Both of them have some things in common:

- There is a distinction between immediate objects (`CHARACTER`s, `FIXNUM`s, `SHORT-FLOAT`s, etc) and heap allocated objects.
- All object pointers are typed, i.e. contain a few bits of information about the type of the pointed-to object. At a minimum, these bits must allow to distinguish immediate and heap-allocated objects.
- Not all of the type information is contained in the object pointer. For example, **CLOS** objects can change their type when `CHANGE-CLASS` is called. To avoid scanning all the heap for references when this happens, the class information is stored in the heap allocated object, not in the object pointer.

The **`HEAPCODES`** object representation has a minimum of type bits in the object pointer, namely, 2 bits. They allow to distinguish immediate objects (which have some more type bits), `CONS`es (which have no type bits in the heap, since they occupy just two words in the heap, with no header), other heap objects (many, from `SIMPLE-VECTOR`s to `FFI:FOREIGN-POINTER`s), and Subrs. Most object types are distinguished by looking a the *`rectype`* field in the header of the heap object.

The **`TYPECODES`** object representation has about two dozen of types encoded in 6 or 7 bits in the object pointer. Typically these are the upper 8 bits of a word (on a 32-bit machine) or the upper 16 bits or 32 bits of a word (on a 64-bit machine). The particular values of the typecodes allow many common operations to be performed with a single bit test (e.g. `CONSP` and `MINUSP` for a `REAL` are bit tests) or range check. However, the *`rectype`* field still exists for many types, because there are many built-in types which do not need a particularly fast type test.

Which object representation is chosen is decided at build time depending on the available preprocessor definitions. You can define **`TYPECODES`** or **`HEAPCODES`** to force one or the other.

One might expect that **`TYPECODES`** is faster than **`HEAPCODES`** because it does not need to make as many memory accesses. This effect is, however, hardly measurable in practice (certainly not more than 5% faster). Apparently because, first, the situations where the type of an object is requested but then the object is not looked into are rare. It is much more common to look into an object, regardless of its type. Second, due to the existence of data caches in the CPU, accessing a heap location twice, once for the type test and then immediately afterwards for the data, is not significantly slower than just accessing the data.

TYPECODES is problematic on 32-bit machines, when you want to use more than 16 MB of memory, because the type bits (at bit 31..24) interfere with the bits of a heap address. For this reason, **HEAPCODES** is the default on 32-bit platforms.

HEAPCODES is problematic on platforms whose object alignment is less than 4. This affects only the mc680x0 CPU; however, here the alignment can usually be guaranteed through some **gcc** options.

35.4. Memory Models

There are 6 memory models. Which one is used, depends on the operating system and is determined at build time.

Memory Models

SPVW_MIXED_BLOCKS_OPPOSITE

The heap consists of one block of fixed length (allocated at startup). The variable-length objects are allocated from the left, the 2-pointer objects are allocated from the right. There is a hole between them. When the hole shrinks to 0, garbage-collect is invoked. garbage-collect slides the variable-length objects to the left and concentrates the 2-pointer objects at the right end of the block again. When no more room is available, some reserve area beyond the right end of the block is halved, and the 2-pointer objects are moved to the right accordingly.

Advantages and Disadvantages

(+) Simple management.

(+) No fragmentation at all.

(-) The total heap size is limited.

SPVW_MIXED_BLOCKS_OPPOSITE & TRIVIALMAP_MEMORY

The heap consists of two big blocks, one for variable-length objects and one for 2-pointer objects. The former one has a hole to the right and is extensible to the right, the latter one has a hole to the left and is extensible to the left. Similar to the previous model, except that the hole is unmapped.

Advantages and Disadvantages

(+) Total heap size grows depending on the application's needs.

(+) No fragmentation at all.

(*) Works only when SINGLEMAP_MEMORY is possible as well.

SPVW_MIXED_BLOCKS_STAGGERED & TRIVIALMAP_MEMORY

The heap consists of two big blocks, one for variable-length objects and one for 2-pointer objects. Both have a hole to the right, but are extensible to the right.

Advantages and Disadvantages

(+) Total heap size grows depending on the application's needs.

(+) No fragmentation at all.

(*) Works only when SINGLEMAP_MEMORY is possible as well.

SPVW_MIXED_PAGES

The heap consists of many small pages (usually around 8 KB). There are two kinds of pages: one for 2-pointer objects, one for variable-length objects. The set of all pages of a fixed kind is called a "Heap". Each page has its hole (free space) at its end. For every heap, the pages are kept sorted according to the size of their hole, using AVL trees. The garbage-collection is invoked when the used space has grown by 25% since the last GC; until that point new pages are allocated from the OS. The GC compacts the data in each page separately: data is moved to the left. Emptied pages are given back to the OS. If the holes then make up more than 25% of the occupied storage, a second GC turn moves objects across pages, from nearly empty ones to nearly full ones, with the aim to free as many pages as possible.

Advantages and Disadvantages

(-) Every allocation requires AVL tree operations, thus slower

(+) Total heap size grows depending on the application's needs.

(+) Works on operating systems which do not provide large contiguous areas.

SPVW_PURE_PAGES

Just like SPVW_MIXED_PAGES, except that every page contains data of only a single type tag, i.e. there is a Heap for every type tag.

Advantages and Disadvantages

(-) Every allocation requires AVL tree operations, thus slower

- **(+)** Total heap size grows depending on the application's needs.
- **(+)** Works on operating systems which do not provide large contiguous areas.
- **(-)** More fragmentation because objects of different type never fit into the same page.

SPVW_PURE_BLOCKS

There is a big block of storage for each type tag. Each of these blocks has its data to the left and the hole to the right, but these blocks are extensible to the right (because there is enough room between them). A garbage-collection is triggered when the allocation amount since the last GC reaches 50% of the amount of used space at the last GC, but at least 512 KB. The garbage-collection cleans up each block separately: data is moved left.

Advantages and Disadvantages

- **(+)** Total heap size grows depending on the application's needs.
- **(+)** No 16 MB total size limit.
- **(*)** Works only in combination with SINGLEMAP_MEMORY.

In page based memory models, an object larger than a page is the only object carried by its pages. There are no small objects in pages belonging to a big object.

The following combinations of memory model and `mmap` tricks are possible (the number indicates the order in which the respective models have been developed):

Table 35.1. Memory models with `TYPECODES`

	A	B	C	D	E
SPVW_MIXED_BLOCKS_OPPOSITE	1	10		2	9
SPVW_MIXED_BLOCKS_STAGGERED		7			8
SPVW_PURE_BLOCKS			5		6
SPVW_MIXED_PAGES	3				
SPVW_PURE_PAGES	4				

Table 35.2. Memory models with `HEAPCODES`

	A	B	E
SPVW_MIXED_BLOCKS_OPPOSITE	*	*	*
SPVW_MIXED_BLOCKS_STAGGERED		*	*
SPVW_MIXED_PAGES	*		

Legend to Table 35.1, “Memory models with TYPECODES” and Table 35.2, “Memory models with HEAPCODES”

A. no MAP_MEMORY
B. TRIVIALMAP_MEMORY
C. SINGLEMAP_MEMORY
D. MULTIMAP_MEMORY
E. GENERATIONAL_GC

35.5. The burden of garbage-collection upon the rest of CLISP

35.5.1. Lisp object invalidation

Every subroutine marked with “can trigger GC” or `maygc` may invoke garbage-collection. garbage-collector moves all the **CLISP** non-immediate objects and updates the pointers. But the garbage-collector looks only at the `STACK` and not in the **C** variables. (Anything else would not be portable.) Therefore at every "unsafe" point, i.e. every call to such a subroutine, all the **C** variables of type object **MUST BE ASSUMED TO BECOME GARBAGE**. (Except for objects that are known to be unmovable, e.g. immediate objects or Subrs.) Pointers inside **CLISP** data (e.g. to the characters of a `STRING` or to the elements of a `SIMPLE-VECTOR`) become **INVALID** as well.

35.5.2. Managing Lisp objects in C

The workaround is usually to allocate all the needed **CLISP** data first and do the rest of the computation with **C** variables, without calling unsafe routines, and without worrying about garbage-collection.

Alternatively, you can save a lisp object on the `STACK` using macros `pushSTACK()` and `popSTACK()`.

Warning

One should not mix these macros in one statement because **C** may execute different parts of the statement out of order. E.g.,

```
pushSTACK(listof(4));
```

is illegal.

35.5.3. Run-time GC-safety checks

Run-time GC-safety checking is available when you build **CLISP** with a C++ compiler, e.g.:

```
$ CC=g++ ./configure --with-debug build-g-gxx
```

When built like this, **CLISP** will `abort` when you reference GC-unsafe data after an allocation (which could have triggered a garbage-collection), and **gdb** will pinpoint the trouble spot.

Specifically, when **CLISP** is configured as above, there is a global integer variable `alloccount` and the object structure contains an integer *`allocstamp`* slot. If these two integers are not the same, the object is invalid. By playing with **gdb**, you should be able to figure out the precise spot where an allocation increments `alloccount` **after** the object has been retrieved from a GC-visible location.

35.5.4. Memory protection

Generational garbage-collector uses memory protection, so when passing pointers into the lisp heap to C functions, you may encounter errors (`errno=EFAULT`) unless you call `handle_fault_range (protection,region_start,region_end)` on the appropriate memory region. See files

`src/unixaux.d`
`src/win32aux.d`
`modules/syscalls/calls.c`
`modules/rawsock/rawsock.c`

for examples.

35.6. Foreign Pointers

Pointers to C functions and to `malloc`ed data can be hidden in **CLISP** objects of type machine_type; garbage-collect will not modify its value. But one should not dare to assume that a C stack pointer or the address of a C function in a shared library satisfies the same requirements.

If another pointer is to be viewed as a **CLISP** object, it is best to box it, e.g. in a `SIMPLE-BIT-VECTOR` or in an Fpointer (using `allocate_fpointer()`.)

Chapter 36. Extending CLISP

Table of Contents

Common Lisp is a *programmable* programming language.

--John Foderaro

CLISP can be easily extended the same way any other **Common Lisp** implementation can: create a lisp file with your variables, functions, macros, etc.; (optionally) compile it with `COMPILE-FILE`; `LOAD` it into a running **CLISP**, and save the memory image.

This method does not work when you need to use some functionality not available in **CLISP**, e.g., you want to call a **C** function. You are urged to use External Modules instead of adding built-in functions.

Note

CLISP comes with an **“FFI”** which allows you to access **C** libraries in an easy way (including creating `FFI:FOREIGN-FUNCTION`s dynamically).

36.1. Adding a built-in function

In the rare cases when you really need to modify **CLISP** internals and add a truly built-in function, you should read the **CLISP** sources for inspiration and enlightenment, choose a file where your brand-new built-in function should go to, and then ...

- add the `LISPFUN` form and the implementation there;
- add the `LISPFUN` header to file `subr.d`;
- declare the function name in file `constsym.d` in the appropriate package (probably **“EXT”**, if there is no specific package);
- if your function accepts keyword arguments, then an appropriate pair of forms must be added to `subrkw.d` and you must make sure that the keyword symbols are declared in `constsym.d`;
- export your function name from the appropriate package in file `init.lisp`;
- when you are done, you should run **make check-sources** in your build directory: this will check that the definitions (source files) and the declarations (`subr.d`, `subrkw.d` and `fsubr.d`) are in sync.

Warning

Be very careful with the *GC-unsafe* functions! Always remember about GC-safety!

These instructions are intentionally terse - you are encouraged to use modules and/or **"FFI"** instead of adding built-ins directly.

36.2. Adding a built-in variable

If you must be able to access the Lisp variable in the **C** code, follow these steps:

- declare the variable name in `constsym.d` in the appropriate package (probably **"CUSTOM"**, if there is no specific package);
- add a `define_variable()` call in function `init_symbol_values()` in file `spvw.d`;
- export your variable name from the appropriate package in file `init.lisp`;

36.3. Recompilation

Any change that forces **make** to remake `lisp.run`, will force recompilation of all `#P".lisp"` files and re-dumping of `lispinit.mem`, which may be time-consuming. This is not always necessary, depending on what kind of change you introduced.

On the other hand, if you change any of the following files:

`constobj.d`
`constsym.d`
`fsubr.d`
`subr.d`
`subrkw.d`

your `lispinit.mem` will *have* to be re-dumped.

Warning

If you change the signature of any system function mentioned in the `FUNTAB` arrays in file `eval.d`, all the `#P".fas"` files will become obsolete and will need to be recompiled. You will need to add a note to that effect to the `src/NEWS` file and augment the object `version` in file `constsym.d`. Please try to avoid this as much as possible.

Chapter 37. The CLISP bytecode specification

Table of Contents

37.1. Introduction

The **CLISP** compiler compiles **Common Lisp** programs into instruction codes for a virtual processor. This bytecode is optimized for saving space in the most common cases of **Common Lisp** programs. The main advantages/drawbacks of this approach, compared to native code compilation, are:

- Bytecode compiled programs are a lot smaller than when compiled to native code. This results in better use of CPU caches, and in less virtual memory paging. Users perceive this as good responsiveness.
- Maximum execution speed (throughput in tight loops) is limited.
- Since no bytecode instructions are provided for "unsafe" operations (like unchecked array accesses, or "fast" `CAR`/`CDR`), programs run with all safety checks enabled even when compiled.
- Execution speed of a program can easily be understood by looking at the output of the `DISASSEMBLE` function. A rule of thumb is that every elementary instruction costs 1 time unit, whereas a function call costs 3 to 4 time units.
- Needing to do no type inference, the compiler is pretty straightforward and fast. As a consequence, the definition of **CLOS** generic functions, which needs to compile small pieces of generated code, is not perceived to be slow.
- The compiler is independent from the hardware CPU. Different back-ends, one for each hardware CPU, are not needed. As a consequence, the compiler is fairly small (and would have been easily maintainable if it were written in a less kludgey way...), and it is impossible for the compiler writer to introduce CPU dependent bugs.

37.2. The virtual machine

The bytecode can be thought of as being interpreted by a virtual processor. The engine which interprets the bytecode (the "implementation of the virtual machine”) is actually a C function, but it could as well be a just-in-time compiler which translates a function's bytecode into hardware CPU instructions the first time said function is called.

The virtual machine is a stack machine with two stacks:

STACK
: a stack for **CLISP** objects and frames (“Lisp stack”).

SP
: a stack for other data and pointers (“Program stack”).

This two-stack architecture permits to save an unlimited number of **CLISP** objects on the `STACK` (needed for handling of **Common Lisp** multiple values), without consing[3]. Also, in a world with a compacting no-ambiguous-roots garbage collector, `STACK` must only hold **CLISP** objects, and `SP` can hold all the other data belonging to a frame, which would not fit into `STACK` without tagging/untagging overhead.

The scope of `STACK` and `SP` is only valid for a given function invocation. Whereas the amount of `STACK` space needed for executing a function (excluding other function calls) is unlimited, the amount of `SP` space needed is known a priori, at compile time. When a function is called, no relation is specified between the caller's `STACK` and the callee's `STACK`, and between the caller's `SP` and the callee's `SP`. The bytecode is designed so that outgoing arguments on the caller's `STACK` can be shared by the caller's incoming arguments area (on the callee's `STACK`), but a virtual machine implementation may also copy outgoing arguments to incoming arguments instead of sharing them.

The virtual machine has a special data structure, `values`, containing the “top of stack”, specially adapted to **Common Lisp** multiple values:

`mv_count`
: an unsigned integer.

`value1`
: the primary value, a **CLISP** object. If `mv_count` = 0, this is `NIL`.

`mv_space`
: all values except the first one, an array of **CLISP** objects.

The contents of `values` is short-lived. It does not survive a function call, not even a garbage-collection.

The interpretation of some bytecode instructions depends on a constant, `jmpbufsize`. This is a CPU-dependent number, the value of `SYSTEM::*JMPBUF-SIZE*`. In **C**, it is defined as `ceiling(sizeof (jmp_buf),sizeof(void*))`.

37.3. The structure of compiled functions

A compiled function consists of two objects: The function itself, containing the references to all **CLISP** objects needed for the bytecode, and a byte vector containing only immediate data, including the bytecode proper.

Typically, the byte vector is about twice as large as the function vector. The separation thus helps the garbage collector (since the byte vector does not need to be scanned for pointers).

A function looks like this (cf. the **C** type Cclosure):

name
: This is the name of the function, normally a symbol or a list of the form `(SETF` *`symbol`*`)`. It is used for printing the function and for error messages. This field is immutable.

`codevec`
: This is the byte-code vector, a `(VECTOR (UNSIGNED-BYTE 8))`. This field is immutable.

`consts`[]
: The remaining fields in the function object are references to other **CLISP** objects. These references are immutable, which is why they are called “constants”. (The referenced **CLISP** objects can be mutable objects, such as `CONS`es or `VECTOR`s, however.)

The Exception to the Immutability Rule

When a generic function's dispatch code is installed, the `codevec` and `consts` fields are destructively modified.

Some of the `consts` can play special roles. A function looks like this, in more detail:

name
: see name.

`codevec`
: see `codevec`.

`venv-const`*
: At most one object, representing the closed-up variables, representing the variables of the lexical environment in which this function was defined. It is a `SIMPLE-VECTOR`, which looks like this: `#(next value₁ ... valueₙ)` where *`value₁`*, ..., *`valueₙ`* are the values of the closed-up variables, and *`next`* is either `NIL` or a `SIMPLE-VECTOR` having the same structure.

`block-const`*
: Objects representing closed-up `BLOCK` tags, representing the `BLOCK` tags of the lexical environment in which this function was defined. Each is a `CONS` containing in the `CDR` part: either a frame pointer to the block frame, or `#<DISABLED>`. The `CAR` is the block's name, for error messages only.

`tagbody-const`*
: Objects representing closed-up `TAGBODY` tags, representing the `TAGBODY` tags of the lexical environment in which this function was defined. Each is a `CONS` containing in the `CDR` part: either a frame pointer to the `TAGBODY` frame, or `#<DISABLED>` if the `TAGBODY` has already been left. The `CAR` is a `SIMPLE-VECTOR` containing the names of the `TAGBODY` tags, for error messages only.

`keyword-const`*
: If the function was defined with a lambda list containing `&KEY`, here come the symbols ("keywords"), in their correct order. They are used by the interpreter during function call.

`other-const`*
: Other objects needed by the function's bytecode.

If `venv-const`, `block-const`, `tagbody-const` are all absent, the function is called *autonomous*. This is the case if the function does not refer to lexical variables, blocks or tags defined in compile code outside of the function. In particular, it is the case if the function is defined in a null lexical environment.

If some `venv-const`, `block-const`, or `tagbody-const` are present, the function (a “closure”) is created at runtime. The compiler only generates a prototype, containing `NIL` values instead of each `venv-const`, `block-`

const, tagbody-const. At runtime, a function is created by copying this prototype and replacing the NIL values by the definitive ones.

The list (keyword-const* other-const*) normally does not contain duplicates, because the compiler removes duplicates when possible. (Duplicates can occur nevertheless, through the use of LOAD-TIME-VALUE.)

The codevec looks like this (cf. the C type Codevec):

spdepth_1 (2 bytes)
: The 1st part of the maximal SP depth.

spdepth_jmpbufsize (2 bytes)
: The jmpbufsize part of the maximal SP depth. The maximal SP depth (precomputed by the compiler) is given by `spdepth_1 + spdepth_jmpbufsize * jmpbufsize`.

numreq (2 bytes)
: Number of required parameters.

numopt (2 bytes)
: Number of optional parameters.

flags (1 byte)

: **bit 0**
  : set if the function has the &REST parameter

  bit 7
  : set if the function has &KEY parameters

  bit 6
  : set if the function has &ALLOW-OTHER-KEYS

  bit 4
  : set if the function is a generic function

  bit 3
  : set if the function is a generic function and its effective method shall be returned (instead of being executed)

signature (1 byte)
: An abbreviation code depending on numreq, numopt, flags. It is used for speeding up the function call.

numkey (2 bytes, only if the function has &KEY)
: The number of &KEY parameters.

keyconsts (2 bytes, only if the function has &KEY)

The offset of the `keyword-const` in the function.

`byte*` (any number of bytes)
: The bytecode instructions.

37.4. The general structure of the instructions

All instructions consist of one byte, denoting the opcode, and some number of operands.

The conversion from a byte (in the range 0..255) to the opcode is performed by lookup in the table contained in the file `bytecode.d`.

There are the following types of operands, denoted by different letters:

`k`*, *`n`*, *`m`*, *`l`
: A (nonnegative) numeric operand. The next byte is read. If its bit 7 is zero, then the bits 6..0 give the value (7 bits). If its bit 7 is one, then the bits 6..0 and the subsequent byte together form the value (15 bits).

`b`
: A (nonnegative) 1-byte operand. The next byte is read and is the value.

`label`
: A label operand. A signed numeric operand is read: The next byte is read. If its bit 7 is zero, then the bits 6..0 give the value (7 bits, sign-extended). If its bit 7 is one, then the bits 6..0 and the subsequent byte together form the value (15 bits, sign-extended). If the latter 15-bit result is zero, then four more bytes are read and put together (32 bits, sign-extended). Finally, the bytecode pointer for the target is computed as the current bytecode pointer (pointing after the operand just read), plus the signed numeric operand.

37.5. The instruction set

37.5.1. Instructions for constants

mnemonic	description	semantics
`(NIL)`	Load `NIL` into values.	`value1` := `NIL`, `mv_count` := 1
`(PUSH-NIL n)`	Push *n* `NIL`s into the `STACK`.	*n* times do: *--`STACK` := `NIL`, values undefined
`(T)`	Load `T` into values.	`value1` := `T`, `mv_count` := 1
`(CONST n)`	Load the function's *n*th constant into values.	`value1` := `consts`[*n*], `mv_count` := 1

37.5.2. Instructions for lexical variables

mnemonic	description	semantics
`(LOAD n)`	Load a directly accessible local variable into values.	`value1` := *(`STACK`+*n*), `mv_count` := 1
`(LOADI k1 k2 n)`	Load an indirectly accessible local variable into values.	k := k_1 + `jmpbufsize` * k_2, `value1` := *(*(`SP`+*k*)+ *n*), `mv_count` := 1

mnemonic	description	semantics
(LOADC *n* *m*)	Load a closed-up variable, defined in the same function and directly accessible, into values.	value1 := SVREF(*(STACK+*n*),1+*m*), mv_count := 1
(LOADV *k* *m*)	Load a closed-up variable, defined in an outer function, into values.	*v* := venv-const, *m* times do: *v* := SVREF(*v*,0), value1 := SVREF(*v*,*m*), mv_count := 1
(LOADIC k_1 k_2 *n* *m*)	Load a closed-up variable, defined in the same function and indirectly accessible, into values.	$k := k_1$ + jmpbufsize * k_2, value1 := SVREF(*(*(SP+*k*)+*n*),1+*m*), mv_count := 1
(STORE *n*)	Store values into a directly accessible local variable.	*(STACK+*n*) := value1, mv_count := 1
(STOREI k_1 k_2 *n*)	Store values into an indirectly accessible local variable.	$k := k_1$ + jmpbufsize * k_2, *(*(SP+*k*)+ *n*) := value1, mv_count := 1
(STOREC *n* *m*)	Store values into a closed-up variable, defined in the same function and directly accessible.	SVREF(*(STACK+*n*),1+m) := value1, mv_count := 1
(STOREV *k* *m*)	Store values into a closed-up variable, defined in an outer function.	*v* := venv-const, *m* times do: *v* := SVREF(*v*,0), SVREF(*v*,*m*) := value1, mv_count := 1
(STOREIC k_1 k_2 *n* *m*)	Store values into a closed-up variable, defined in the same function and indirectly accessible.	$k := k_1$ + jmpbufsize * k_2, SVREF(*(*(SP+*k*)+*n*),1+*m*) := value1, mv_count := 1

37.5.3. Instructions for dynamic variables

mnemonic	description	semantics
(GETVALUE *n*)	Load a symbol's value into values.	value1 := symbol-value(consts[*n*]), mv_count := 1
(SETVALUE *n*)	Store values into a symbol's value.	symbol-value(consts[*n*]) := value1, mv_count := 1
(BIND *n*)	Bind a symbol dynamically.	Bind the value of the symbol consts[*n*] to value1, implicitly STACK -= 3, values undefined
(UNBIND1)	Dissolve one binding frame.	Unbind the binding frame STACK is pointing to, implicitly STACK += 3
(UNBIND *n*)	Dissolve *n* binding frames.	*n* times do: Unbind the binding frame STACK is pointing to, thereby incrementing STACK Thus, STACK += 1+2**n*
(PROGV)	Bind a set of symbols dynamically to a set of values.	*symbols* := *STACK++, *--SP := STACK, build a single binding frame binding the symbols in *symbols* to the values in value1, values undefined

37.5.4. Instructions for stack operations

mnemonic	description	semantics
(PUSH)	Push one object onto the STACK.	*--STACK := value1, values undefined
(POP)	Pop one object from the STACK, into values.	value1 := *STACK++, mv_count := 1
(SKIP *n*)	Restore a previous STACK pointer. Remove *n* objects from the STACK.	STACK := STACK + *n*
(SKIPI k_1 k_2 *n*)	Restore a previous STACK pointer. Remove an unknown number of objects from the STACK.	k := k_1 + jmpbufsize * k_2, STACK := *(SP+k), SP := SP+k+1, STACK := STACK + *n*

mnemonic	description	semantics
(SKIPSP k_1 k_2)	Restore a previous SP pointer.	k := k_1 + jmpbufsize * k_2, SP := SP+k

37.5.5. Instructions for control flow, jumps

mnemonic	description	semantics
(SKIP&RET n)	Clean up the STACK, and return from the function.	STACK := STACK+n, return from the function, returning values.
(SKIP&RETGF n)	Clean up the STACK, and return from the generic function.	If bit 3 is set in the function's flags, then STACK := STACK+n, mv_count := 1, and return from the function. Otherwise: if the current function has no &REST argument, then STACK := STACK+n-numreq, apply value1 to the numreq arguments still on the STACK, and return from the function. Else STACK := STACK+n-numreq-1, apply value1 to the numreq arguments and the &REST argument, all still on the STACK, and return from the function.
(JMP *label*)	Jump to *label*.	PC := *label*.
(JMPIF *label*)	Jump to *label*, if value1 is true.	If value1 is not NIL, PC := *label*.
(JMPIFNOT *label*)	Jump to *label*, if value1 is false.	If value1 is NIL, PC := *label*.
(JMPIF1 *label*)	Jump to *label* and forget secondary	If value1 is not NIL, mv_count := 1, PC := *label*.

mnemonic	description	semantics
	values, if `value1` is true.	
`(JMPIFNOT1 label)`	Jump to *`label`* and forget secondary values, if `value1` is false.	If `value1` is `NIL`, `mv_count` := 1, PC := *`label`*.
`(JMPIFATOM label)`	Jump to *`label`*, if `value1` is not a cons.	If `value1` is not a cons, PC := *`label`*. values undefined
`(JMPIFCONSP label)`	Jump to *`label`*, if `value1` is a cons.	If `value1` is a cons, PC := *`label`*. values undefined
`(JMPIFEQ label)`	Jump to *`label`*, if `value1` is `EQ` to the top-of-stack.	If eq(`value1`,*`STACK`++), PC := *`label`*. values undefined
`(JMPIFNOTEQ label)`	Jump to *`label`*, if `value1` is not `EQ` to the top-of-stack.	If not eq(`value1`,*`STACK`++), PC := *`label`*. values undefined
`(JMPIFEQTO n label)`	Jump to *`label`*, if the top-of-stack is `EQ` to a constant.	If eq(*`STACK`++,`consts`[*n*]), PC := *`label`*. values undefined
`(JMPIFNOTEQTO n label)`	Jump to *`label`*, if the top-of-stack is	If not eq(*`STACK`++,`consts`[*n*]), PC := *`label`*. values undefined

mnemonic	description	semantics
	not EQ to a constant.	
(JMPHASH *n* *label*)	Table-driven jump, depending on value1.	Lookup value1 in the hash table consts[*n*]. (The hash table's test is either EQ or EQL.) If found, the hash table value is a signed FIXNUM, jump to it: PC := PC + value. Else jump to *label*. values undefined
(JMPHASHV *n* *label*)	Table-driven jump, depending on value1, inside a generic function.	Lookup value1 in the hash table SVREF(consts[0],*n*). (The hash table's test is either EQ or EQL.) If found, the hash table value is a signed FIXNUM, jump to it: PC := PC + value. Else jump to *label*. values undefined
(JSR *label*)	Subroutine call.	*--STACK := function. Then start interpreting the bytecode at *label*, with values undefined. When a (RET) is encountered, program execution is resumed at the instruction after (JSR *label*).
(JMPTAIL *m* *n* *label*)	Tail subroutine call.	*n* >= *m*. The STACK frame of size *n* is reduced to size *m*: {*(STACK+*n*-*m*), ..., *(STACK+*n*-1)} := {*STACK, ..., *(STACK+*m*-1)}. STACK += n-m. *--STACK := function. Then jump to *label*, with values undefined.

37.5.6. Instructions for lexical environment, creation of closures

mnemonic	description	semantics
(VENV)	Load the venv-const into values.	value1 := consts[0], mv_count := 1.

mnemonic	description	semantics
(MAKE-VECTOR1&PUSH *n*)	Create a SIMPLE-VECTOR used for closed-up variables.	*v* := new SIMPLE-VECTOR of size *n*+1. SVREF(*v*,0) := value1. *--STACK := *v*. values undefined
(COPY-CLOSURE *m* *n*)	Create a closure by copying the prototype and filling in the lexical environment.	*f* := copy-function(consts[*m*]). For *i*=0,..,*n*-1: *f*_consts[i] := *(STACK+*n*-1-*i*). STACK += *n*. value1 := *f*, mv_count := 1

37.5.7. Instructions for function calls

mnemonic	description	semantics
(CALL *k* *n*)	Calls a constant function with *k* arguments.	The function consts[*n*] is called with the arguments *(STACK+*k*-1), ..., *(STACK+0). STACK += *k*. The returned values go into values.
(CALL0 *n*)	Calls a constant function with 0 arguments.	The function consts[*n*] is called with 0 arguments. The returned values go into values.
(CALL1 *n*)	Calls a constant function with 1 argument.	The function consts[*n*] is called with one argument *STACK. STACK += 1. The returned values go into values.
(CALL2 *n*)	Calls a constant function with 2 arguments.	The function consts[*n*] is called with two arguments *(STACK+1) and *(STACK+0). STACK += 2. The returned values go into values.
(CALLS1 *b*)	Calls a system function with no &REST.	Calls the system function FUNTAB[*b*]. The right number of arguments is already on the STACK (including #<UNBOUND>s in place of absent &OPTIONAL or &KEY parameters). The arguments are removed from the STACK. The returned values go into values.

mnemonic	description	semantics
(CALLS2 *b*)	Calls a system function with no &REST.	Calls the system function FUNTAB [256+*b*]. The right number of arguments is already on the STACK (including #<UNBOUND>s in place of absent &OPTIONAL or &KEY parameters). The arguments are removed from the STACK. The returned values go into values.
(CALLSR *m* *b*)	Calls a system function with &REST.	Calls the system function FUNTABR [*b*]. The minimum number of arguments is already on the STACK, and *m* additional arguments as well. The arguments are removed from the STACK. The returned values go into values.
(CALLC)	Calls a computed compiled function with no &KEY.	Calls the compiled function value1. The right number of arguments is already on the STACK (including #<UNBOUND>s in place of absent &OPTIONAL parameters). The arguments are removed from the STACK. The returned values go into values.
(CALLCKEY)	Calls a computed compiled function with &KEY.	Calls the compiled function value1. The right number of arguments is already on the STACK (including #<UNBOUND>s in place of absent &OPTIONAL or &KEY parameters). The arguments are removed from the STACK. The returned values go into values.
(FUNCALL *n*)	Calls a computed function.	Calls the function *(STACK+*n*) with the arguments *(STACK+*n*-1), ..., *(STACK+0). STACK += *n*+1. The returned values go into values.
(APPLY *n*)	Calls a computed function with an	Calls the function *(STACK+*n*) with the arguments *(STACK+*n*-1), ..., *

mnemonic	description	semantics
	unknown number of arguments.	(STACK+0) and a list of additional arguments value1. STACK += *n*+1. The returned values go into values.

37.5.8. Instructions for optional and keyword parameters

mnemonic	description	semantics
(PUSH-UNBOUND *n*)	Push *n* #<UNBOUND>s into the STACK.	*n* times do: *--STACK := #<UNBOUND>. values undefined
(UNLIST *n* *m*)	Destructure a proper LIST.	0 ≤ *m* ≤ *n*. *n* times do: *--STACK := CAR(value1), value1 := CDR (value1). During the last *m* iterations, the list value1 may already have reached its end; in this case, *--STACK := #<UNBOUND>. At the end, value1 must be NIL. values undefined
(UNLIST* *n* *m*)	Destructure a proper or dotted LIST.	0 ≤ *m* ≤ *n*, *n* > 0. *n* times do: *--STACK := CAR(value1), value1 := CDR(value1). During the last *m* iterations, the list value1 may already have reached its end; in this case, *--STACK := #<UNBOUND>. At the end, after *n* CDRs, *--STACK := value1. values undefined
(JMPIFBOUNDP *n* *label*)	Jump to *label*, if a local variable is not unbound.	If *(STACK+*n*) is not #<UNBOUND>, value1 := *(STACK+*n*), mv_count := 1, PC := *label*. Else: values undefined.
(BOUNDP *n*)	Load T or NIL into values, depending	If *(STACK+*n*) is not #<UNBOUND>, value1 := T, mv_count := 1. Else: value1 := NIL, mv_count := 1.

mnemonic	description	semantics
	on whether a local variable is bound.	
(UNBOUND->NIL *n*)	If a local variable is unbound, assign a default value NIL to it.	If *(STACK+*n*) is #<UNBOUND>, *(STACK+*n*) := NIL.

37.5.9. Instructions for multiple values

mnemonic	description	semantics
(VALUES0)	Load no values into values.	value1 := NIL, mv_count := 0
(VALUES1)	Forget secondary values.	mv_count := 1
(STACK-TO-MV *n*)	Pop the first *n* objects from STACK into values.	Load values(*(STACK+*n*-1),...,*(STACK+0)) into values. STACK += *n*.
(MV-TO-STACK)	Save values on STACK.	Push the mv_count values onto the STACK (in order: value1 comes first). STACK -= mv_count. values undefined
(NV-TO-STACK *n*)	Save *n* values on STACK.	Push the first *n* values onto the STACK (in order: value1 comes first). STACK -= *n*. values undefined
(MV-TO-LIST)	Convert multiple values into a list.	value1 := list of values, mv_count := 1
(LIST-TO-MV)	Convert a LIST into multiple values.	Call the function VALUES-LIST with value1 as argument. The returned values go into values.
(MVCALLP)	Start a MULTIPLE-VALUE-CALL invocation.	*--SP := STACK. *--STACK := value1.
(MVCALL)	Finish a MULTIPLE-	newSTACK := *SP++. Call the function *(newSTACK-1), passing it *

mnemonic	description	semantics
	VALUE-CALL invocation.	(newSTACK-2), ..., *(STACK+0) as arguments. STACK := newSTACK. The returned values go into values.

37.5.10. Instructions for BLOCK and RETURN-FROM

mnemonic	description	semantics
(BLOCK-OPEN *n* *label*)	Create a BLOCK frame.	Create a BLOCK frame, STACK -= 3, SP -= 2+jmpbufsize. The topmost (third) object in the block frame is CONS (consts[*n*],frame-pointer) (its *block-cons*). Upon a RETURN-FROM to this frame, execution will continue at *label*. values undefined.
(BLOCK-CLOSE)	Dissolve a BLOCK frame.	Dissolve the BLOCK frame at STACK, STACK += 3, SP += 2+jmpbufsize. Mark the *block-cons* as invalid.
(RETURN-FROM *n*)	Leave a BLOCK whose *block-cons* is given.	*block-cons* := consts[*n*]. If CDR (*block-cons*) = #<DISABLED>, an ERROR is SIGNALed. Else CDR(*block-cons*) is a frame-pointer. Unwind the stack up to this frame, pass it values.
(RETURN-FROM-I k_1 k_2 *n*)	Leave a BLOCK whose *block-cons* is indirectly accessible.	k := k_1 + jmpbufsize * k_2, *block-cons* := *(*(SP+k)+n). If CDR(*block-cons*) = #<DISABLED>, an ERROR is SIGNALed. Else CDR(*block-cons*) is a frame-pointer. Unwind the stack up to this frame, pass it values.

37.5.11. Instructions for TAGBODY and GO

mnemonic	description	semantics
(TAGBODY-OPEN *m* *label$_1$* ... *label$_n$*)	Create a TAGBODY frame.	Fetch consts[*m*], this is a SIMPLE-VECTOR with *n* elements, then decode *n* label operands. Create a TAGBODY frame, STACK -= 3+*n*, SP -= 1+jmpbufsize. The third object in the TAGBODY frame is CONS(consts[*m*],frame-pointer) (the *tagbody-cons*) Upon a GO to tag *label* of this frame, execution will continue at *label$_l$*. values undefined
(TAGBODY-CLOSE-NIL)	Dissolve a TAGBODY frame, and load NIL into values.	Dissolve the TAGBODY frame at STACK, STACK += 3+*m*, SP += 1+jmpbufsize. Mark the *tagbody-cons* as invalid. value1 := NIL, mv_count := 1.
(TAGBODY-CLOSE)	Dissolve a TAGBODY frame.	Dissolve the TAGBODY frame at STACK, STACK += 3+*m*, SP += 1+jmpbufsize. Mark the *tagbody-cons* as invalid.
(GO *n* *label*)	Jump into a TAGBODY whose *tagbody-cons* is given.	*tagbody-cons* := consts[*n*]. If CDR(*tagbody-cons*) = #<DISABLED>, an ERROR is SIGNALed. Else CDR(*tagbody-cons*) is a frame-pointer. Unwind the stack up to this frame, pass it the number *label*.
(GO-I *k$_1$* *k$_2$* *n* *label*)	Jump into a TAGBODY whose *tagbody-cons* is indirectly accessible.	*k* := *k$_1$* + jmpbufsize * *k$_2$*, *tagbody-cons* := *(*(SP+*k*)+*n*). If CDR(*tagbody-cons*) = #<DISABLED>, an ERROR is SIGNALed. Else CDR(*tagbody-cons*) is a frame-pointer. Unwind the stack up to this frame, pass it the number *label*.

37.5.12. Instructions for CATCH and THROW

mnemonic	description	semantics
(CATCH-OPEN *label*)	Create a CATCH frame.	Create a CATCH frame, with value1 as tag. STACK -= 3, SP -= 2+jmpbufsize. Upon a THROW to this tag execution continues at *label*.
(CATCH-CLOSE)	Dissolve a CATCH frame.	Dissolve the CATCH frame at STACK. STACK += 3, SP += 2+jmpbufsize.
(THROW)	Non-local exit to a CATCH frame.	*tag* := *STACK++. Search the innermost CATCH frame with tag *tag* on the STACK, unwind the stack up to it, pass it values.

37.5.13. Instructions for UNWIND-PROTECT

mnemonic	description	semantics
(UNWIND-PROTECT-OPEN *label*)	Create an UNWIND-PROTECT frame.	Create an UNWIND-PROTECT frame. STACK -= 2, SP -= 2+jmpbufsize. When the stack will be unwound by a non-local exit, values will be saved on STACK, and execution will be transferred to *label*.
(UNWIND-PROTECT-NORMAL-EXIT)	Dissolve an UNWIND-PROTECT frame, and start the cleanup code.	Dissolve the UNWIND-PROTECT frame at STACK. STACK += 2, SP += 2+jmpbufsize. *--SP := 0, *--SP := 0, *--SP := STACK. Save the values on the STACK, STACK -= mv_count.
(UNWIND-PROTECT-CLOSE)	Terminate the cleanup code.	newSTACK := *SP++. Load values(*(newSTACK-1), ..., *(STACK+0)) into values. STACK := newSTACK. SPword1 := *SP++, SPword2 := *SP++. Continue depending on SPword1 and SPword2. If both are 0, simply continue execution. If SPword2 is 0 but SPword1 is nonzero, interpret it as a label and jump to it.

mnemonic	description	semantics
(UNWIND-PROTECT-CLEANUP)	Dissolve an UNWIND-PROTECT frame, and execute the cleanup code like a subroutine call.	Dissolve the UNWIND-PROTECT frame at STACK, get *label* out of the frame. STACK += 2, SP += 2+jmpbufsize. *--SP := 0, *--SP := PC, *--SP := STACK. Save the values on the STACK, STACK -= mv_count. PC := *label*.

37.5.14. Instructions for HANDLER-BIND

mnemonic	description	semantics
(HANDLER-OPEN *n*)	Create a handler frame.	Create a handler frame, using consts[*n*] which contains the CONDITION types, the corresponding labels and the current SP depth (= function entry SP - current SP).
(HANDLER-BEGIN&PUSH)	Start a handler.	Restore the same SP state as after the HANDLER-OPEN. value1 := the CONDITION that was passed to the handler, mv_count := 1. *--STACK := value1.

37.5.15. Instructions for some inlined functions

mnemonic	description	semantics
(NOT)	Inlined call to NOT.	value1 := not(value1), mv_count := 1.
(EQ)	Inlined call to EQ.	value1 := eq(*STACK++,value1), mv_count := 1.
(CAR)	Inlined call to CAR.	value1 := CAR(value1), mv_count := 1.
(CDR)	Inlined call to CDR.	value1 := CDR(value1), mv_count := 1.

mnemonic	description	semantics
(CONS)	Inlined call to CONS.	value1 := cons(*STACK++,value1), mv_count := 1.
(SYMBOL-FUNCTION)	Inlined call to SYMBOL-FUNCTION.	value1 := SYMBOL-FUNCTION(value1), mv_count := 1.
(SVREF)	Inlined call to SVREF.	value1 := SVREF(*STACK++,value1), mv_count := 1.
(SVSET)	Inlined call to (SETF SVREF).	*arg1* := *(STACK+1), *arg2* := *(STACK+0), STACK += 2. SVREF (*arg2*,value1) := *arg1*. value1 := *arg1*, mv_count := 1.
(LIST *n*)	Inlined call to LIST.	value1 := LIST(*(STACK+*n*-1),...,*(STACK+0)), mv_count := 1, STACK += *n*.
(LIST* *n*)	Inlined call to LIST*.	value1 := LIST*(*(STACK+*n*-1),..., *(STACK+0),value1), mv_count := 1, STACK += *n*.

37.5.16. Combined instructions

The most frequent short sequences of instructions have an equivalent combined instruction. They are only present for space and speed optimization. The only exception is FUNCALL&SKIP&RETGF, which is needed for generic functions.

mnemonic	equivalent
(NIL&PUSH)	(NIL) (PUSH)
(T&PUSH)	(T) (PUSH)
(CONST&PUSH *n*)	(CONST *n*) (PUSH)
(LOAD&PUSH *n*)	(LOAD *n*) (PUSH)
(LOADI&PUSH k_1 k_2 *n*)	(LOADI k_1 k_2 *n*) (PUSH)
(LOADC&PUSH *n* *m*)	(LOADC *n* *m*) (PUSH)
(LOADV&PUSH *k* *m*)	(LOADV *k* *m*) (PUSH)
(POP&STORE *n*)	(POP) (STORE *n*)

mnemonic	equivalent
(GETVALUE&PUSH *n*)	(GETVALUE *n*) (PUSH)
(JSR&PUSH *label*)	(JSR *label*) (PUSH)
(COPY-CLOSURE&PUSH *m n*)	(COPY-CLOSURE *m n*) (PUSH)
(CALL&PUSH *k n*)	(CALL *k n*) (PUSH)
(CALL1&PUSH *n*)	(CALL1 *n*) (PUSH)
(CALL2&PUSH *n*)	(CALL2 *n*) (PUSH)
(CALLS1&PUSH *b*)	(CALLS1 *b*) (PUSH)
(CALLS2&PUSH *b*)	(CALLS2 *b*) (PUSH)
(CALLSR&PUSH *m n*)	(CALLSR *m n*) (PUSH)
(CALLC&PUSH)	(CALLC) (PUSH)
(CALLCKEY&PUSH)	(CALLCKEY) (PUSH)
(FUNCALL&PUSH *n*)	(FUNCALL *n*) (PUSH)
(APPLY&PUSH *n*)	(APPLY *n*) (PUSH)
(CAR&PUSH)	(CAR) (PUSH)
(CDR&PUSH)	(CDR) (PUSH)
(CONS&PUSH)	(CONS) (PUSH)
(LIST&PUSH *n*)	(LIST *n*) (PUSH)
(LIST*&PUSH *n*)	(LIST* *n*) (PUSH)
(NIL&STORE *n*)	(NIL) (STORE *n*)
(T&STORE *n*)	(T) (STORE *n*)
(LOAD&STOREC *k n m*)	(LOAD *k*) (STOREC *n m*)
(CALLS1&STORE *b k*)	(CALLS1 *b*) (STORE *k*)
(CALLS2&STORE *b k*)	(CALLS2 *b*) (STORE *k*)
(CALLSR&STORE *m n k*)	(CALLSR *m n*) (STORE *k*)
(LOAD&CDR&STORE *n*)	(LOAD *n*) (CDR) (STORE *n*)
(LOAD&CONS&STORE *n*)	(LOAD *n*+1) (CONS) (STORE *n*)
(LOAD&INC&STORE *n*)	(LOAD *n*) (CALL1 #'1+) (STORE *n*)
(LOAD&DEC&STORE *n*)	(LOAD *n*) (CALL1 #'1-) (STORE *n*)
(LOAD&CAR&STORE *m n*)	(LOAD *m*) (CAR) (STORE *n*)

mnemonic	equivalent
(CALL1&JMPIF *n* *label*)	(CALL1 *n*) (JMPIF *label*)
(CALL1&JMPIFNOT *n* *label*)	(CALL1 *n*) (JMPIFNOT *label*)
(CALL2&JMPIF *n* *label*)	(CALL2 *n*) (JMPIF *label*)
(CALL2&JMPIFNOT *n* *label*)	(CALL2 *n*) (JMPIFNOT *label*)
(CALLS1&JMPIF *b* *label*)	(CALLS1 *b*) (JMPIF *label*)
(CALLS1&JMPIFNOT *b* *label*)	(CALLS1 *b*) (JMPIFNOT *label*)
(CALLS2&JMPIF *b* *label*)	(CALLS2 *b*) (JMPIF *label*)
(CALLS2&JMPIFNOT *b* *label*)	(CALLS2 *b*) (JMPIFNOT *label*)
(CALLSR&JMPIF *m* *n* *label*)	(CALLSR *m* *n*) (JMPIF *label*)
(CALLSR&JMPIFNOT *m* *n* *label*)	(CALLSR *m* *n*) (JMPIFNOT *label*)
(LOAD&JMPIF *n* *label*)	(LOAD *n*) (JMPIF *label*)
(LOAD&JMPIFNOT *n* *label*)	(LOAD *n*) (JMPIFNOT *label*)
(LOAD&CAR&PUSH *n*)	(LOAD *n*) (CAR) (PUSH)
(LOAD&CDR&PUSH *n*)	(LOAD *n*) (CDR) (PUSH)
(LOAD&INC&PUSH *n*)	(LOAD *n*) (CALL1 #'1+) (PUSH)
(LOAD&DEC&PUSH *n*)	(LOAD *n*) (CALL1 #'1-) (PUSH)
(CONST&SYMBOL-FUNCTION *n*)	(CONST *n*) (SYMBOL-FUNCTION)
(CONST&SYMBOL-FUNCTION&PUSH *n*)	(CONST *n*) (SYMBOL-FUNCTION) (PUSH)
(CONST&SYMBOL-FUNCTION&STORE *n* *k*)	(CONST *n*) (SYMBOL-FUNCTION) (STORE *k*)
(APPLY&SKIP&RET *n* *k*)	(APPLY *n*) (SKIP&RET *k*)
(FUNCALL&SKIP&RETGF *n* *k*)	(FUNCALL *n*) (SKIP&RETGF *k*)

37.5.17. Shortcut instructions

There are special one-byte instructions (without explicit operands) for the following frequent instructions:

mnemonic	operand range
(LOAD *n*)	$0 \leq n < 15$
(LOAD&PUSH *n*)	$0 \leq n < 25$
(CONST *n*)	$0 \leq n < 21$
(CONST&PUSH *n*)	$0 \leq n < 30$
(STORE *n*)	$0 \leq n < 8$

37.6. Bytecode Design

This section offers some insight into bytecode design in the form of questions and answers.

37.6.1. When to add a new bytecode?

Question:

Does it make sense to define a new bytecode instruction for RESTART-CASE? Why? Why not?

Answer: Is it speed critical?

RESTART-CASE is a glorified LET binding for SYSTEM::*ACTIVE-RESTARTS* and could well profit from a separate bytecode: it would make it non-consing[3]. (Remember that RESTARTs have dynamic extent and therefore do not really need to be heap allocated.)

The reason HANDLER-BIND has its own bytecodes and RESTART-CASE does not is that HANDLER-BIND can occur in inner computation loops, whereas RESTART-CASE occurs only as part of user-interface programming and therefore not in inner loops where its consing could hurt much.

37.6.2. Why JMPTAIL?

Question:

Consider this function and its disassembly:

```
 (defun foo (x y) (if (or (= x 0) (= y 0)) (+ x y) (foo y
 (DISASSEMBLE 'foo)
8      (LOAD&PUSH 1)
9      (LOAD&DEC&PUSH 3)
11     (JMPTAIL 2 5 L0)
```

Why are the arguments pushed onto the `STACK`, just to be popped off of it during the `JMPTAIL`? Why not a sequence of `LOAD`, `STORE` and `SKIP` instructions followed by a `JMP`?

Answer: This is a shortcut for the most common use

Using `JMPTAIL` requires 3 instructions, `JMP` requires more. When `JMPTAIL` needs to be called, we usually have some stuff close to the top of the `STACK` which will become the new arguments, and some junk between these new arguments and the closure object. `JMPTAIL` removes the junk. `JMPTAIL` is a convenient shortcut which shortens the bytecode - because typically one would really have to clean-up the `STACK` by hand or make the calculations in `src/compiler.lisp` more complicated.

Part V. Appendices

Table of Contents

Appendix A. Frequently Asked Questions (With Answers) about CLISP

Abstract

This is a list of frequently asked questions about **CLISP** on the CLISP mailing lists and the USENET newsgroup comp.lang.lisp. *All* the legitimate technical question are addressed in the **CLISP** documentation (**CLISP** impnotes, clisp(1)), and for such questions this list provides a link into the docs. The frequently asked political questions are answered here in *full* detail (meaning that no further explanations of the issues could be provided).

Please submit more questions (and answers!) to <`clisp-list@lists.sourceforge.net`> (http://lists.sourceforge.net/lists/listinfo/clisp-list).

FAQ

A.1. Meta Information

A.1.1. Miscellaneous

A.1.1.1. What is “FAQ fine”?

We assess a nominal fine of $10 for asking a question that is document. We further assess a fine of $1 for asking a questio **CLISP** manual. The fines are payable to the person who ansv these amounts to be exorbitant, please feel free to ignore this

This should **not** discourage you from asking questions, but ra

A.1.1.2. The official **CLISP** documentation sucks - is anything better

As with all generic complaints, the answer to this one is PTC

Additionally, the nightly builds of the **CLISP** implementatioı
head are available at http://clisp.podval.org/impnotes/. It cont
features and the general improvements in the documentation.

A.1.1.3. License - why GNU GPL?

Because **CLISP** uses GNU readline.

Note that this does not necessarily prevent you from distribut
CLISP. See *Note* in `COPYRIGHT`.

A.1.1.4. What about [ANSI CL standard] compliance?

CLISP *purports to conform* to the [ANSI CL standard] speci

`$ clisp -ansi`

from the [ANSI CL standard] standard are bugs and are not (

On the other hand, some decisions made by the ANSI X3J13
technical point of view as were most of them, and some of th
after the alternative behavior has already been implemented i
pains to modify **CLISP** to unconditionally comply with the [
cases except for a handful of situations where they believed tl
which cases the committee behavior is *still* optionally availab

CLISP does not start in the ansi mode by default for historic
Dumping an image or passing a command line argument are

A.1.1.5. How do I ask for help?

Politely - please refer to `Netiquette`.

If you have a question about **CLISP**, you have the following
audience size):

USENET group comp.lang.lisp
This is the right place to ask all general Lisp questions, s
string?"

CLISP User Mailing List <clisp-list@lists.sourcefo
(http://lists.sourceforge.net/lists/listinfo/clisp-list)
AKA http://news.gmane.org/gmane.lisp.clisp.general
This is the right place to ask user-level CLISP-specific c
CLISP image?"

CLISP Developer Mailing List <clisp-devel@lists.sou
(http://lists.sourceforge.net/lists/listinfo/clisp-devel)
AKA http://news.gmane.org/gmane.lisp.clisp.devel
This is the right place to discuss CLISP internals, submi
subscribed to post. If you read this list on Gmane and do
can subscribe to it using the aforementioned web interfac

Individual CLISP developers
This is *never* the right thing to do, unless you want to *hir*
(commercial support, custom enhancements etc). This is
developers are very busy, they might get weeks to answe
be able to help you in the meantine; as well as for the be
mailing lists are publicly archived (you are encouraged t
not copy your messages to the individual developers.

A.1.1.6. Which mailing lists should I subscribe to?

Cross-posting in the CLISP mailing lists is very actively disc
you can subscribe to all mailing lists that are relevant to you

<clisp-announce@lists.sourceforge.net> (http://list
announce)
extremely low-level moderated list, you should definitel
interest in CLISP whatsoever

<clisp-list@lists.sourceforge.net> (http://lists.sou
subscribe to this list of you use CLISP and want to ask (

<clisp-devel@lists.sourceforge.net> (http://lists.so
subscribe to this list if you want to help with CLISP dev
a daily digest).

A.1.1.7. Why is my mail to a mailing list rejected?

CLISP mailing lists get a lot of spam, so the maintainers hav
get a note that “your message is held for moderator's approva
to /dev/null and try again, noting the following:

<clisp-announce@lists.sourceforge.net> (http://list announce)
: do not mail here without a prior discussion on <clisp-c (http://lists.sourceforge.net/lists/listinfo/clisp-devel)

<clisp-devel@lists.sourceforge.net> (http://lists.so
: subscriber-only, you *must* post from a **subscribed addre**

<clisp-list@lists.sourceforge.net> (http://lists.sou
: the only open list, so it is filtered especially aggressively

- no `MIME` mail (no HTML formatting, no attachment
- the list address must be in `CC` or `TO`, not `BCC`.
- do not mention “virgin”, “penis” or “viagra” in the s

If you do not like this policy, please volunteer to maintain the
through all the “held for moderator's approval” mail and appr
day.

A.1.1.8. How do I report bugs?

Patiently!

A.1.1.9. How do I help?

Please read Chapter 36, *Extending **CLISP*** and submit your p
entry (see other entries there for inspiration), to <clisp-dev
(http://lists.sourceforge.net/lists/listinfo/clisp-devel).

See `src/CodingStyle` for the style one should follow.

If your patch is more than just a few lines, it is *much* preferre
the web and send the link to the list.

The patch must be against the CVS head (reasonably recent).

A.1.1.10. How do I debug **CLISP**?

When debugging the core:

```
$ ./configure --with-debug --build build-g
$ cd build-g
$ gdb lisp.run              ;; or lisp.exe on wir
(gdb) boot
(gdb) run
```

When debugging module `foo`:

```
$ ./configure --with-debug --with-module=foo --
$ cd build-g
$ gdb full/lisp.run         ;; or lisp.exe on wir
(gdb) full
(gdb) run
```

When debugging a **base** module, use **base** instead of **full** and

A.1.2. Logo

A.1.2.1. Why is **CLISP** using menorah as the logo?

Whimsical If you must have some answer and you do not care whether it
that **Common Lisp** brings the *Light* to a programmer, and **Cl**

Accordingly, **CLISP** enables you to *see* the truth, thus you ca
if you are a *seasoned* expert, you might pronounce it as *sea-li*

Historical **CLISP** has been using the menorah for the logo since the pro
by Bruno Haible and Michael Stoll. This *probably* reflects the
people, Judaism or the State of Israel (neither of the two origi
ask the original authors for details yourself. Both of them are
prompt reply.

A.1.2.2. Shouldn't the logo be changed now due to the current politica

The **CLISP** developers, both the original creators and the cur
mainstream view that blames the Jews for everything from hi
Niño and Sun spots.

Moreover, today, when Jews are being pushed out of the Ame
with various obscene boycott and divestment campaigns, it is
against the resurgence of Nazism.

For more information, please see:

- Committee for Accuracy in Middle East Reporting in Ar
- Information Regarding Israel's Security
- Middle East Media Research Institute
- YES to peace, NO to terror
- More links

A.1.2.3. Aren't there other political issues of concern?

Yes, there are! For example, in 1989 the communist governm
murdered some 3000+ student human rights protesters at the
appear to have already forgotten this crime. A note to that eff

src/timezone.lisp until 2002, when it was decided that is
moved here.

We also oppose software patents and support other liberal (i.e

A.1.2.4. Aren't you afraid of losing some users who are offended by th

Do you have in mind people like this one? Good riddance!

A.1.2.5. Using software to promote a political agenda is unprofessiona

Expressing their opinion is a perfectly natural thing for the au
views or religious beliefs. The use of the menorah has its root
authors are proud to display it. If you are unlucky enough to l
opinion, due to the constraints of a government, society, relig
relationships", the Free World condoles with you. The author
constraints. If you are unhappy about their artistic preferences
are free to ignore them.

Many scientists have been doing art, politics and religion. Re
mathematics and Christianity. Albert Einstein helped the U.S
in the hands of the Nazis. Bram Moolenaar, the author of VIM
Uganda.

A.2. Running CLISP

A.2.8. How do I create a GUI for my CLISP program?

A.2.1. Where is `DEFUN`?

Pass `-M` to the runtime (`lisp.run` or `lisp.exe`). Use the dri invoking the runtime directly.

A.2.2. Where is the IDE?

Emacs-based.

- `inferior-lisp`
- SLIME
- ILISP
- The Common Lisp Cookbook

non-Emacs-based.

- VisualCLisp
- Jabberwocky
- GClisp
- Portable Hemlock

A.2.3. What are the command line arguments?

See clisp(1).

A.2.4. How do I get out of the debugger?

See Section 25.1, “Debugging Utilities [CLHS-25.1.2]”.

A.2.5. What **CLISP** extensions are available?

Distributed with CLISP

Quite a few modules are included with **CLISP**, pass `--w`
them and use the **full** linking set.

3rd party

See the *incomplete* list of "Common Lisp software runni

DIY

See Section 32.2, "External Modules" and Section 32.3,
information on how to interface with external **C** libraries

HTTP (very Frequently Asked!)

- **CLISP** comes with `src/inspect.lisp` which imp
- CLOCC/src/donc/ has an **HTTP** server
- CLOCC/src/cllib/ handles URLs
- mod_lisp hooks lisp into Apache
- Object-Oriented HTTP server for **CLISP**

Both AllegroServe and CL-HTTP require multithreading

A.2.6. Where is the init ("RC") file on my platform?

Read the file `<clisp.html#opt-norc>` in your build directo
version of the user manual clisp(1) for your platform).

A.2.7. Where are the modules with which I built **CLISP**?

In the **full** linking set. Run **CLISP** like this:

```
$ clisp -K full
```

If your **CLISP** was configured with option `--with-dynamic`
`(REQUIRE name)` where *name* is a `STRING`, e.g., "**`rawsock`**".

Making **base** the default linking set has some advantages:

Shared Library Hell
Avoid problems when a module requires a shared library
is present but with wrong version) on your system.

Smaller Images Are Faster
Adding things to the heap increases working set size cau
important for small to medium applications.

Uniform User Experience
Composition of the **full** linking set is up to the packager,

See <clisp-list@lists.sourceforge.net> (http://lists.s
more information (SFmail/200407212204.44395.bruno%

A.2.8. How do I create a GUI for my **CLISP** program?

Use module Section 33.19, “GTK Interface”: use Glade to cre

```
$ ./configure --with-module=gtk --build bui
$ ./build-gtk/clisp -K full -x '(gtk:run-gl
```

There are many other options, see "Common Lisp software ru

A.3. Application Delivery

A.3.1. How do I create an executable file with all my code in it?

Use `EXT:SAVEINITMEM`, see also Section 32.6, “Quickstartin

A.3.2. When I deliver my application with **CLISP** does it have to be

Not necessarily.

CLISP is Free Software, covered by the GNU GPL, with spe
applications that run in **CLISP**. The precise terms can be fou
source and binary distributions of **CLISP**. Here is an informa
practice. Please refer to the said `COPYRIGHT` file when in dou

In many cases, **CLISP** does not force an application to be co
encourage you to release your software under an open source
your users are numerous, in particular they are free to modify
needs/requirements change, and they are free to recompile th
machine or operating system.

CLISP *extensions*, i.e. programs which need to access non-p
packages **"SYSTEM"**, **"CLOS"**, **"FFI"**, etc), must be cover

Other programs running in **CLISP** have to or need not to be p
their distribution form:

- Programs distributed as Lisp source or `#P".fas"` files c
coming from **CLISP**.
- Programs distributed as **CLISP** memory images can be d
non-**CLISP** `#P".fas"` files which make up the memory
instructions) for rebuilding the memory image.
- If you need to distribute a modified **CLISP** executable (
modules written in **C**), you must distribute its full source
with this, you can instead put the additional modules into
which your Lisp program will communicate via `SOCKET`

A.4. Troubles

A.4.1. Where is the binary distribution for my platform?

The **CLISP** maintainers can only build **CLISP** binary distrib
CompileFarm platforms that are up an running at the time of
reasonably modern **C** compiler.

Note that **CLISP** is included in many software distributions,
CLISP's home page.

A.4.2. But a previous release *had* a binary distribution for my platfo

It was probably contributed by a user who did not (yet?) cont
release. You can find out who contributed a specific binary d
in the SourceForge Files section.

A.4.3. Why does not **CLISP** build on my platform?

Please see file `unix/PLATFORMS` in your source distribution
troublesome platforms as well as instructions on porting **CLI**

A.4.4. What do these messages mean: “`invalid byte #x94 in C`
“`character #\u00B3 cannot be represented in the`

This means that you are trying to read (“invalid byte”) or writ
non-**ASCII** character from (or to) a character stream which h
default is described in `-E`*`domain encoding`*.

This may also be caused by filesystem access. If your `CUSTOI`
incorrectly, many filesystem accesses (like `LOAD`, `DIRECTORY`
will traverse the directories mentioned in `CUSTOM:*LOAD-PA`
too. You will need to set `CUSTOM:*PATHNAME-ENCODING*` o
Using a “1:1” encoding, such as `ISO-8859-1`, should help yo

Note that this error may be signaled by the Print part of the re
you call. E.g., if file `"foo"` contains non-**ASCII** characters, y

```
(with-open-file (s "foo" :direction :input :ext
  (read-line s))
```

If instead you type

```
(with-open-file (s "foo" :direction :input :ext
  (setq l (read-line s))
  nil)
```

CLISP will just print `NIL` and signal the error when you type

A.4.5. What does this message mean: “`Display all 1259 possi`

CLISP uses GNU readline for command line editing and con
possibilities” message (and sometimes many screens of symb
an inappropriate place. You can turn this feature off if you ar
TABs in your code.

A.4.6. Why does not command line editing work?

See Section 21.2.1, “Command line editing with GNU readli

A.4.7. How do I avoid stack overflow?

CLISP has *two stacks*, the “program stack” and the “lisp stac

Avoiding stack overflow: Generic

- You will always get a stack overflow when you try to pri `*PRINT-CIRCLE*` is `NIL`. Just set `*PRINT-CIRCLE*` to `T`
- You will always get a stack overflow on infinite recursio
- Some simple functions (like Ackermann's) recurse more stack on relatively small inputs.
- Compiled code uses less stack (and memory) and is faste
- If you really do need more Lisp stack, you can increase i memory.
- If you get a segmentation fault after (or instead of) a “pro make sure that you had GNU libsigsegv installed when y

Avoiding stack overflow: Platform-specific

Platform Dependent: Win32 platform only.
modify `SYSTEM.INI` or change the `PIF` that you use to ir set program stack using **editbin** (`SFmail/sa0ptz4o4e3` `Gmane/general/5523`) (answered on <`clisp-list@l` (http://lists.sourceforge.net/lists/listinfo/clisp-list))

Platform Dependent: UNIX platform only.
Increase program stack with **ulimit -s** (or **limit stacksize**

A.4.8. Why does my program return different values on each invoca

The following code modifies itself:

```
(let ((var '(a b c)))
  (nconc var '(1 2 3)))
```

and will not work as one would naively expect. (on the first i
second invocation will produce a circular list, the third will h

Instead you must do

```
(let ((var (copy-list '(a b c))))
  (nconc var (copy-list '(1 2 3))))
```

`DISASSEMBLE` will show the constants in your compiled func
bytecode specification for the explanation of the `DISASSEMBI`

See Lisp Programming Style for more useful information.

A.4.9. Why is autoconf invoked during build?

When building from the CVS HEAD development sources, y
tries to regenerate some `configure` scripts for you. **This is n**
an officially released source distribution). Please just **touch** tl

```
$ touch src/configure
```

and re-run **make**.

You can also pass `--disable-maintainer-mode` to the top
default when you are **not** working from the CVS).

A.4.10. Why don't floating point arithmetics return what I want?

```
(- 1.1 0.9)
⇒ 0.20000005
```

This not a bug, at least not a bug in **CLISP**. You may argue t
make sure that you do know What Every Computer Scientist
Arithmetic.

See also <clisp-list@lists.sourceforge.net> (http://l list) (SFmail/17121.26476.75643.47774%40thalassa.i Gmane/general/9850).

PS. If you want *exact* calculations, use RATIONALs:

```
(- 11/10 9/10)
⇒ 1/5
```

A.4.11. Why does

```
$ clisp -x '(RANDOM 1s0)'
```

always print the same number?

Reproducibility is important. See Section 12.3.1, "Random N

A.4.12. Why is an extra line break inserted by the pretty printer?

See Variable CUSTOM:*PPRINT-FIRST-NEWLINE*.

A.4.13. How do I disable this annoying warning?

CLISP often issues WARNINGs when it encounters suspicious than to suppress them. To figure out where the warning is co

```
(SETQ *BREAK-ON-SIGNALS* 'WARNING)
```

and examine the stack (see Section 25.1, "Debugging Utilitie warning is coming from.

If everything else fails, read the manual.

Why does DEFVAR affect previously defined lexical closures?

A.4.14.

```
(defun adder (val) (lambda (x) (+ x val)))
⇒ ADDER
(setq add-10 (adder 10))
⇒ ADD-10
(funcall add-10 32)
⇒ 42 ; as expected
(defvar val 12)
⇒ VAL
(funcall add-10 0)
⇒ 12 ; why not 10?!
```

Explanation The above code does not conform to [ANSI CL standard], the results. See Section 3.2.3, "Semantic Constraints [CLHS-3.2.

Remedy Always follow the naming convention for global special varia DEFPARAMETER (e.g., *FOO*) and DEFCONSTANT (e.g., +BAR+

More
Gmane/general/11945
Gmane/general/11949

A.4.15. Why is the function FOO broken?!

When confronted with unexpected behavior, try looking in th

E.g., **CLISP** DIRECTORY is different from the **CMU CL** impl results you want, you should search the **CLISP** implementati

Alternatively, since the implementation notes are organized in DIRECTORY belongs to the Chapter 20 in [ANSI CL standard] *[CLHS-20]* in impnotes and look for "DIRECTORY" there.

Appendix B. GNU Free Documentation License

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of

Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this

License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (C) YEAR YOUR NAME.
>
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

GNU General Public License

Version 2, June 1991

Version 2, June 1991

Table of Contents

C.1. Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software - to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps:

1. copyright the software, and
2. offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

C.2. TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

C.2.1. Section 0

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a “work based on the Program ” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification ”.) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

C.2.2. Section 1

You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

C.2.3. Section 2

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License.

Exception:

If the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

C.2.4. Section 3

You may copy and distribute the Program (or a work based on it, under Section 2 in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-

readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

C.2.5. Section 4

You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

C.2.6. Section 5

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

C.2.7. Section 6

Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

C.2.8. Section 7

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

C.2.9. Section 8

If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

C.2.10. Section 9

The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

C.2.11. Section 10

If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

C.2.12. NO WARRANTY Section 11

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

C.2.13. Section 12

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE

PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

C.3. How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'. This is free software, and you are welcome to redistribute
it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

Index

Symbols

D

E

F

G

L

M

N

References

Books

[CLtL1] Guy L. Steele, Jr.. *Common Lisp: the Language (1st Edition)*. 1984. 465 pages. ISBN 0-932376-41-X. Digital Press.

[CLtL2] Guy L. Steele, Jr.. *Common Lisp: the Language (2nd Edition)*. 1990. 1032 pages. ISBN 1-555-58041-6. Digital Press.

[AMOP] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. 1991. 335 pages. ISBN 0-262-61074-4. MIT Press.

ANSI standard documents

[ANSI CL] ANSI CL standard1994. *ANSI Common Lisp standard X3.226 -1994 - Information Technology - Programming Language - Common Lisp*.

[CLHS] Common Lisp HyperSpec*Common Lisp HyperSpec*.

These notes document **CLISP** version 2.43 Last modified: 2007-11-18