

ECL User's Guide

Giuseppe Attardi
Juan Jose Garcia Ripoll (revised version)
Daniel Kochmański (revised revised version)

Copyright © 1990, Giuseppe Attardi
Copyright © 2000, Juan Jose Garcia Ripoll
Copyright © 2015, Daniel Kochmański

Preface

Embeddable Common Lisp is an implementation of Common-Lisp originally designed for being *embeddable* into C based applications. This document describes the Embeddable Common Lisp implementation and how it differs from [ANSI, see [Bibliography], page 193] and [Steele:84, see [Bibliography], page 193]. Chapter 4 [Developer’s guide], page 149, and Chapter 1 [User’s guide], page 9, for the details about the implementation and how to interface with other languages.

Table of Contents

Introduction	3
About this book	3
User's guide	3
Developer's guide	3
Standards	3
Extensions	3
What is ECL	3
History	4
Credits	6
Copyrights	7
Copyright of ECL	7
Copyright of this manual	8
 1 User's guide	 9
1.1 Building ECL	9
1.1.1 Autoconf based configuration	9
1.1.2 Platform specific instructions	10
1.1.2.1 MSVC based configuration	10
1.1.2.2 Android	10
1.2 Entering and leaving Embeddable Common Lisp	11
1.3 The break loop	13
1.4 Embedding ECL	13
1.4.1 Embedding Reference	13
1.4.1.1 Starting and Stopping	13
1.4.1.2 Catching Errors and Managing Interrupts	16
 2 Standards	 21
2.1 Overview	21
2.1.1 Reading this manual	21
2.1.2 C Reference	21
2.2 Evaluation and compilation	23
2.2.1 Compiler declaration <code>optimize</code>	23
2.2.2 <code>declaim</code> and <code>proclaim</code>	24
2.2.3 C Reference	25
2.2.3.1 ANSI Dictionary	25
2.3 Types and classes	25
2.3.1 C Reference	25
2.3.1.1 ANSI Dictionary	25
2.4 Data and control flow	26
2.4.1 Shadowed bindings	26
2.4.2 Minimal compilation	26
2.4.3 Function types	27

2.4.4	C Calling conventions.....	27
2.4.5	C Reference	28
2.4.5.1	ANSI Dictionary.....	32
2.5	Objects	33
2.5.1	C Reference	33
2.5.1.1	ANSI Dictionary.....	33
2.6	Structures.....	34
2.6.1	Redefining a defstruct structure.....	34
2.6.2	C Reference	34
2.6.2.1	ANSI Dictionary.....	34
2.7	Conditions	34
2.7.1	C Reference	34
2.7.1.1	ANSI dictionary	36
2.8	Symbols.....	36
2.8.1	C Reference	36
2.8.1.1	ANSI Dictionary.....	37
2.9	Packages	38
2.9.1	C Reference	38
2.9.1.1	ANSI Dictionary.....	38
2.10	Numbers	39
2.10.1	Numeric types.....	39
2.10.2	Floating point exceptions.....	40
2.10.3	Random-States	40
2.10.4	Infinity and Not a Number	41
2.10.5	Dictionary.....	41
2.10.6	C Reference	41
2.10.6.1	Number C types	42
2.10.6.2	Number constructors	43
2.10.6.3	Number accessors.....	44
2.10.6.4	Number coercion	44
2.10.6.5	ANSI dictionary	45
2.11	Characters	47
2.11.1	Unicode vs. POSIX locale	47
2.11.1.1	Character types	48
2.11.1.2	Character names.....	48
2.11.2	#\Newline characters.....	48
2.11.3	C Reference	49
2.11.3.1	C types.....	49
2.11.3.2	Constructors.....	49
2.11.3.3	Predicates	49
2.11.3.4	Character case.....	50
2.11.3.5	ANSI Dictionary.....	50
2.12	Conses	51
2.12.1	C Reference	51
2.12.1.1	Accessors.....	51
2.12.1.2	ANSI Dictionary.....	51
2.13	Arrays	54
2.13.1	Array limits	54

2.13.2	Specializations.....	54
2.13.3	C Reference	55
2.13.3.1	Types and constants.....	55
2.13.3.2	ecl_aet_to_symbol, ecl_symbol_to_aet.....	56
2.13.3.3	Constructors.....	56
2.13.3.4	Accessors.....	57
2.13.3.5	Array properties.....	58
2.13.3.6	ANSI Dictionary.....	58
2.14	Strings.....	60
2.14.1	String types & Unicode.....	60
2.14.2	C reference.....	60
2.14.2.1	Base string constructors	60
2.14.2.2	String accessors.....	61
2.14.2.3	ANSI dictionary.....	61
2.15	Sequences.....	62
2.15.1	C Reference	62
2.15.1.1	ANSI dictionary.....	63
2.16	Hash tables	64
2.16.1	Extensions	64
2.16.1.1	Weakness in hash tables	64
2.16.1.2	Thread-safe hash tables.....	65
2.16.1.3	Hash tables serialization.....	65
2.16.1.4	Custom equivalence predicate.....	65
2.16.1.5	Example.....	65
2.16.2	C Reference	66
2.16.2.1	ANSI dictionary.....	66
2.17	Filenames.....	66
2.17.1	Syntax.....	66
2.17.2	Wild pathnames and matching.....	68
2.17.3	C Reference	68
2.17.3.1	ANSI dictionary.....	68
2.18	Files.....	69
2.18.1	Dictionary.....	69
2.18.2	C Reference	70
2.18.2.1	ANSI Dictionary.....	70
2.19	Streams.....	70
2.19.1	ANSI Streams.....	70
2.19.1.1	Supported types	70
2.19.1.2	Element types.....	70
2.19.1.3	External formats	70
2.19.2	Dictionary.....	73
2.19.2.1	Sequence Streams.....	73
2.19.2.2	File Stream Extensions	73
2.19.2.3	External Format Extensions	74
2.19.3	C Reference	75
2.19.3.1	ANSI dictionary.....	75
2.20	Printer.....	77
2.20.1	C Reference	78

2.20.1.1	ANSI Dictionary	78
2.21	Reader	79
2.21.1	<code>*read-supress*</code>	79
2.21.2	C Reference	79
2.21.2.1	ANSI Dictionary	79
2.22	System construction	80
2.22.1	C Reference	80
2.22.1.1	ANSI Dictionary	80
2.23	Environment	80
2.23.1	Dictionary	81
2.23.2	C Reference	83
2.23.2.1	ANSI Dictionary	83
3	Extensions	85
3.1	System building	85
3.1.1	Compiling with ECL	85
3.1.1.1	Portable FASL	86
3.1.1.2	Native FASL	87
3.1.1.3	Object file	88
3.1.1.4	Static library	88
3.1.1.5	Shared library	89
3.1.1.6	Executable	89
3.1.1.7	Summary	90
3.1.2	Compiling with ASDF	90
3.1.2.1	Example code to build	90
3.1.2.2	Build it as an single executable	90
3.1.2.3	Build it as shared library and use in C	91
3.1.2.4	Build it as static library and use in C	92
3.1.3	C compiler configuration	93
3.1.3.1	Compiler flags	93
3.1.3.2	Compiler & Linker programs	93
3.2	Operating System Interface	94
3.2.1	Command line arguments	94
3.2.2	External processes	96
3.2.3	FIFO files (named pipes)	98
3.2.4	Operating System Interface Reference	98
3.3	Foreign Function Interface	98
3.3.1	What is a FFI?	98
3.3.2	Two kinds of FFI	99
3.3.3	Foreign objects	100
3.3.4	Higher level interfaces	101
3.3.5	SFFI Reference	103
3.3.6	DFFI Reference	107
3.3.7	UFFI Reference	107
3.3.7.1	Primitive Types	107
3.3.7.2	Aggregate Types	109
3.3.7.3	Foreign Objects	112
3.3.7.4	Foreign Strings	116

3.3.7.5	Functions and Libraries	120
3.4	Native threads	122
3.4.1	Tasks, threads or processes	122
3.4.2	Processes (native threads)	122
3.4.3	Processes dictionary	122
3.4.4	Locks (mutexes)	126
3.4.5	Locks dictionary	126
3.4.6	Readers-writer locks	127
3.4.7	Read-Write locks dictionary	128
3.4.8	Condition variables	128
3.4.9	Condition variables dictionary	129
3.4.10	Semaphores	129
3.4.11	Semaphores dictionary	129
3.4.12	Atomic operations	130
3.4.13	Atomic operations dictionary	130
3.5	Signals and Interrupts	133
3.5.1	Problems associated to signals	134
3.5.2	Kinds of signals	134
3.5.2.1	Synchronous signals	134
3.5.2.2	Asynchronous signals	135
3.5.3	Signals and interrupts in ECL	135
3.5.3.1	Handling of asynchronous signals	135
3.5.3.2	Handling of synchronous signals	136
3.5.4	Considerations when embedding ECL	137
3.5.5	Signals Reference	137
3.6	Memory Management	138
3.6.1	Introduction	138
3.6.2	Boehm-Weiser garbage collector	138
3.6.3	Memory limits	139
3.6.4	Memory conditions	140
3.6.5	Finalization	140
3.6.6	Memory Management Reference	141
3.7	Meta-Object Protocol (MOP)	143
3.7.1	Introduction	143
3.8	Gray Streams	143
3.9	Tree walker	143
3.10	Local package nicknames	143
3.10.1	Overview	143
3.10.2	Package local nicknames dictionary	144
3.11	Package locks	145
3.11.1	Package Locking Overview	145
3.11.2	Operations Violating Package Locks	145
3.11.3	Package Lock Dictionary	146
3.12	CDR Extensions	147

4	Developer's guide	149
4.1	Sources structure	149
4.1.1	src/c	149
4.2	Contributing	152
4.3	Defun preprocessor	153
4.4	Manipulating Lisp objects	154
4.4.1	Objects representation	154
4.4.2	Constructing objects	157
4.5	Environment implementation	165
4.6	The interpreter	165
4.6.1	ECL stacks	165
4.6.2	Procedure Call Conventions	165
4.6.3	The lexical environment	167
4.6.4	The interpreter stack	167
4.7	The compiler	169
4.7.1	The compiler translates to C	169
4.7.2	The compiler mimics human C programmer	169
4.7.3	Implementation of Compiled Closures	171
4.7.4	Use of Declarations to Improve Efficiency	172
4.7.5	Inspecting generated C code	173
4.8	Porting ECL	174
4.9	Removed features	174
Indexes		177
	Concept index	177
	Configure option index	178
	Feature index	179
	Example index	179
	Function index	180
	Variable index	184
	Type index	185
	Common Lisp symbols	185
	C/C++ index	188
Bibliography		193

Introduction

About this book

This manual is part of the ECL software system. It documents deviations of ECL from various standards ([ANSI, see [Bibliography], page 193], [AMOP, see [Bibliography], page 193],...), extensions, daily working process (compiling files, loading sources, creating programs, etc) and the internals of this implementation.

It is not intended as a source to learn Common Lisp. There are other tutorials and textbooks available in the Net which serve this purpose. The homepage of the Common-Lisp.net (<https://common-lisp.net>) contains a good list of links of such teaching and learning material.

This book is structure into four parts:

User's guide

We begin with [Chapter 1 [User's guide], page 9] which provides introductory material showing the user how to build and use ECL and some of its unique features. This part assumes some basic Common Lisp knowledge and is suggested as an entry point for a new users who want to start using Embeddable Common Lisp.

Developer's guide

[Chapter 4 [Developer's guide], page 149] documents Embeddable Common Lisp implementation details. This part is not meant for normal users but rather for the ECL developers and other people who want to contribute to Embeddable Common Lisp. This section is prone to change due to the dynamic nature of a software. Covered topics include source code structure, contributing guide, internal implementation details and many other topics relevant to the development process.

Standards

[Chapter 2 [Standards], page 21] documents all parts of the standard which are left as implementation specific or to which ECL doesn't adhere. For instance, precision of floating point numbers, available character sets, actual input/output protocols, etc.

Section covers also *C Reference* as a description of ANSI Common-Lisp from the C/C++ programmer perspective and *ANSI Dictionary* for Common-Lisp constructs available from C/C++.

Extensions

[Chapter 3 [Extensions], page 85] introduces all features which are specific to ECL and which lay outside the standard. This includes configuring, building and installing ECL multiprocessing capabilities, graphics libraries, interfacing with the operating system, etc.

What is ECL

Common-Lisp is a general purpose programming language. It lays its roots in the LISP programming language [LISP1.5, see [Bibliography], page 193] developed by John McCarthy in

the 80s. Common-Lisp as we know it ANSI Common-Lisp is the result of an standardization process aimed at unifying the multiple lisp dialects that were born from that language.

Embeddable Common Lisp is an implementation of the Common-Lisp language. As such it derives from the implementation of the same name developed by Giuseppe Attardi, which itself was built using code from the Kyoto Common-Lisp [Yasa:85, see [Bibliography], page 193]. [History], page 4, for the history of the code you are about to use.

Embeddable Common Lisp (ECL for short) uses standard C calling conventions for Lisp compiled functions, which allows C programs to easily call Lisp functions and vice versa. No foreign function interface is required: data can be exchanged between C and Lisp with no need for conversion.

ECL is based on a Common Runtime Support (CRS) which provides basic facilities for memory management, dynamic loading and dumping of binary images, support for multiple threads of execution. The CRS is built into a library that can be linked with the code of the application. ECL is modular: main modules are the program development tools (top level, debugger, trace, stepper), the compiler, and CLOS. A native implementation of CLOS is available in ECL. A runtime version of ECL can be built with just the modules which are required by the application.

The ECL compiler compiles from Lisp to C, and then invokes the C compiler to produce binaries. Additionally portable bytecode compiler is provided for machines which doesn't have C compiler. While former releases of ECL adhere to the the reference of the language given in Common-Lisp: *The Language*2 [Steele90, see [Bibliography], page 193], the ECL is now compliant with X3J13 ANSI Common Lisp [ANSI, see [Bibliography], page 193].

History

The ECL project is an implementation of the Common Lisp language inherits from many other previous projects, as shown in Figure 1. The oldest ancestor is the Kyoto Common Lisp, an implementation developed at the the Research Institute for Mathematical Sciences, Kyoto University [Yasa:85, see [Bibliography], page 193]. This implementation was developed partially in C and partially in Common Lisp itself and featured a lisp to C translator.

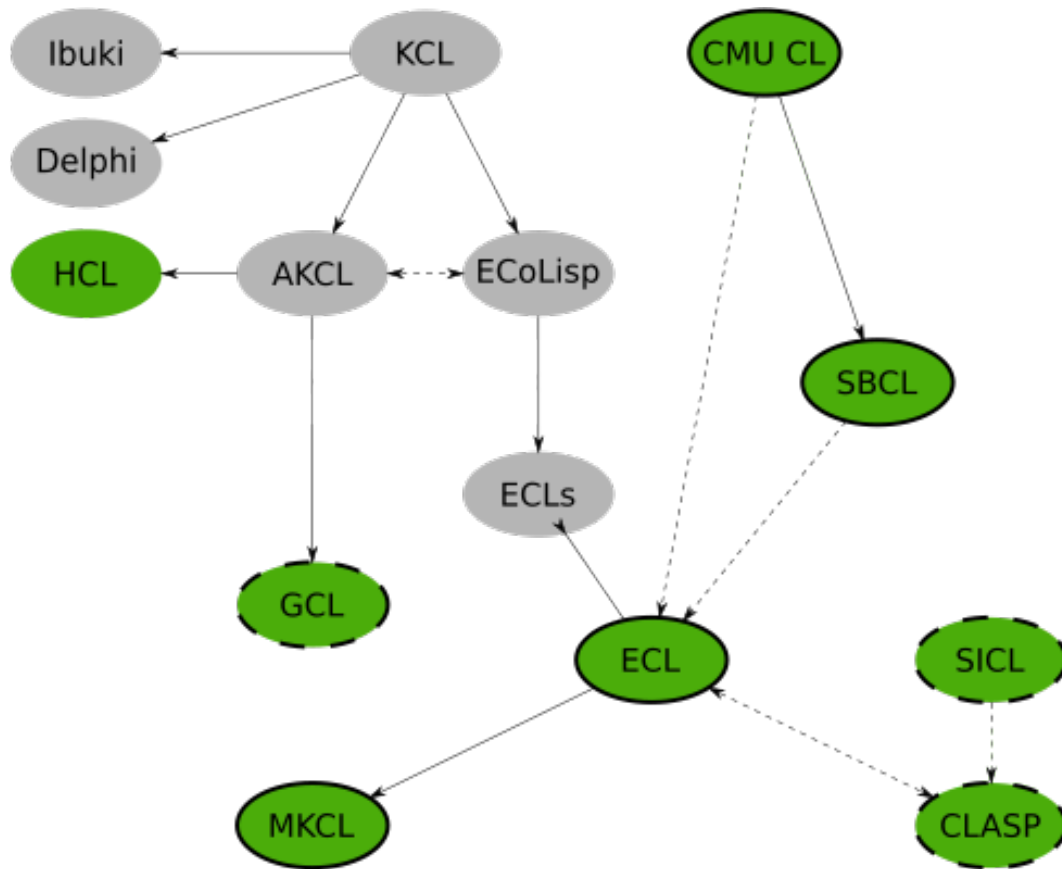


Figure 1: ECL's family tree

The KCL implementation remained a proprietary project for some time. During this time, William F. Schelter improved KCL in several areas and developed Austin Kyoto Common-Lisp (AKCL). However, those changes had to be distributed as patches over the proprietary KCL implementation and it was not until much later that both KCL and AKCL became freely available and gave rise to the GNU Common Lisp project, GCL.

Around the 90's, Giuseppe Attardi worked on the KCL and AKCL code basis to produce an implementation of Common Lisp that could be embedded in other C programs [Attardi:95, see [Bibliography], page 193]. The result was an implementation sometimes known as ECL and sometimes as ECoLisp, which achieved rather good compliance to the informal specification of the language in CLTL2 [Steele:90, see [Bibliography], page 193], and which run on a rather big number of platforms.

The ECL project stagnated a little bit in the coming years. In particular, certain dependencies such as object binary formats, word sizes and some C quirks made it difficult to port it to new platforms. Furthermore, ECL was not compliant with the ANSI specification, a goal that other Common Lisps were struggling to achieve.

This is where the ECLS or ECL-Spain project began. Juanjo García-Ripoll took the ECoLisp sources and worked on them, with some immediate goals in mind: increase portability,

make the code 64-bit clean, make it able to build itself from scratch, without other implementation of Common Lisp and restore the ability to link ECL with other C programs.

Those goals were rather quickly achieved. ECL became ported to a number of platforms and with the years also compatibility with the ANSI specification became a more important goal. At some point the fork ECLS, with agreement of Prof. Attardi, took over the original ECL implementation and it became what it is nowadays, a community project.

In 2013 once again project got unmaintained. In 2015 Daniel Kochmański took the position of a maintainer with consent of Juanjo García-Ripoll.

The ECL project owes a lot to different people who have contributed in many different aspects, from pointing out bugs and incompatibilities of ECL with other programs and specifications, to actually solving these bugs and porting ECL to new platforms.

Current development of ECL is still driven by Daniel Kochmański with main focus on improving ANSI compliance and compatibility with the Common Lisp libraries ecosystem, fixing bugs, improving speed and the portability. The project homepage is located at <https://common-lisp.net/project/ecl/>.

Credits

The Embeddable Common Lisp project is an implementation of the Common-Lisp language that aims to comply with the ANSI Common-Lisp standard. The first ECL implementations were developed by Giuseppe Attardi's who produced an interpreter and compiler fully conformant with the Common-Lisp as reported in *Steele:84*. ECL derives itself mostly from Kyoto Common-Lisp, an implementation developed at the Research Institute for Mathematical Sciences (RIMS), Kyoto University, with the cooperation of Nippon Data General Corporation. The main developers of Kyoto Common-Lisp were Taiichi Yuasa and Masami Hagiya, of the Research Institute for Mathematical Sciences, at Kyoto University.

We must thank Giuseppe Attardi, Yuasa and Hagiya and Juan Jose Garcia Ripoll for their wonderful work with preceding implementations and for putting them in the Public Domain under the GNU General Public License as published by the Free Software Foundation. Without them this product would have never been possible.

This document is an update of the original ECL documentation, which was based in part on the material in [Yuasa:85, see [Bibliography], page 193]

The following people or organizations must be credited for support in the development of Kyoto Common-Lisp: Prof. Reiji Nakajima at RIMS, Kyoto University; Nippon Data General Corporation; Teruo Yabe; Toshiyasu Harada; Takashi Suzuki; Kibo Kurokawa; Data General Corporation; Richard Gabriel; Daniel Weinreb; Skef Wholey; Carl Hoffman; Naruhiko Kawamura; Takashi Sakuragawa; Akinori Yonezawa; Etsuya Shibayama; Hagiwara Laboratory; Shuji Doshita; Takashi Hattori.

William F. Schelter improved KCL in several areas and developed Austin Kyoto Common-Lisp (AKCL). Many ideas and code from AKCL have been incorporated in Embeddable Common Lisp.

The following is the partial list of contributors to ECL: Taiichi Yuasa and Masami Hagiya (KCL), William F. Schelter (Dynamic loader, conservative GC), Giuseppe Attardi (Top-level, trace, stepper, compiler, CLOS, multithread), Marcus Daniels (Linux port) Cornelis

van der Laan (FreeBSD port) David Rudloff (NeXT port) Dan Stanger, Don Cohen, and Brian Spilbury.

We have to thank for the following pieces of software that have helped in the development of Embeddable Common Lisp

BRUNO HAIBLE

For the Cltl2-compliance test

PETER VAN EYNDE

For the ANSI-compliance test

SYMBOLIC'S INC.

For the ANSI-compliant LOOP macro.

The Embeddable Common Lisp project also owes a lot to the people who have tested this program and contributed with suggestions, error messages and documentation: Eric Marsden, Hannu Koivisto, Jeff Bowden and Yuto Hayamizu, Bo Yao and others whose name we may have omitted.

Copyrights

Copyright of ECL

ECL is distributed under the GNU LGPL, which allows for commercial uses of the software. A more precise description is given in the Copyright notice which is shipped with ECL.

----- BEGINNING OF COPYRIGHT FOR THE ECL CORE ENVIRONMENT -----

Copyright (C) 2019, Daniel Kochmanski and Marius Gerbershagen

Copyright (c) 2018, Daniel Kochmański

Copyright (c) 2013, Juan Jose Garcia Ripoll

Copyright (c) 1990, 1991, 1993 Giuseppe Attardi

Copyright (c) 1984 Taiichi Yuasa and Masami Hagiya

All Rights Reserved

ECL is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version; see file 'Copying'.

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 Library General Public License for more details.

You should have received a copy of the GNU Library General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

PLEASE NOTE THAT:

This license covers all of the ECL program except for the files
 src/lsp/loop2.lsp ; Symbolic's LOOP macro
 src/lsp/pprint.lsp ; CMUCL's pretty printer
 src/lsp/format.lsp ; CMUCL's format
 and the directories
 contrib/ ; User contributed extensions
 examples/ ; Examples for the ECL usage
 Look the precise copyright of these extensions in the corresponding
 files.

Examples are licensed under: (SPDX-License-Identifier) BSD-2-Clause

Report bugs, comments, suggestions to the ecl mailing list:
 ecl-devel@common-lisp.net.

---- END OF COPYRIGHT FOR THE ECL CORE ENVIRONMENT -----

Copyright of this manual

Copyright Daniel Kochmański and Marius Gerbershagen, 2020

Copyright Daniel Kochmański, 2016

Copyright Juan José García-Ripoll, 2006

Copyright Kevin M. Rosenberg, 2002-2003 (UFFI Reference)

Trademark AllegroCL is a registered trademark of Franz Inc.

Trademark Lispworks is a registered trademark of Xanalys Inc.

Trademark Microsoft Windows is a registered trademark of Microsoft Inc.

Trademark Other brand or product names are the registered trademarks or trademarks of their respective holders.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, with the no Front-Cover Texts, and with no Back-Cover Texts. Exact text of the license is available at <https://www.gnu.org/copyleft/fdl.html>.

1 User's guide

1.1 Building ECL

Due to its portable nature ECL works on every (at least) 32-bit architecture which provides a proper C99 compliant compiler.

Operating systems on which ECL is reported to work: Linux, Darwin (Mac OS X), Solaris, FreeBSD, NetBSD, OpenBSD, DragonFly BSD, Windows and Android. On each of them ECL supports native threads.

In the past Juanjo José García-Ripoll maintained a test farm which performed ECL tests for each release on number of platforms and architectures. Due to lack of the resources we can't afford such doing, however each release is tested by volunteers with an excellent package `cl-test-grid` (<https://common-lisp.net/project/cl-test-grid>) created and maintained by Anton Vodonosov.

1.1.1 Autoconf based configuration

ECL, like many other FOSS programs, can be built and installed with a GNU tool called Autoconf. This is a set of automatically generated scripts that detect the features of your machine, such as the compiler type, existing libraries, desired installation path, and configures ECL accordingly. The following procedure describes how to build ECL using this procedure and it applies to all platforms except for the Windows ports using Microsoft Visual Studio compilers (however you may build ECL with cygwin or mingw using the autoconf as described here).

To build Embeddable Common Lisp you need to

1. Extract the source code and enter it's directory

```
$ tar -xf ecl-xx.x.x.tgz
$ cd ecl-xx.x.x
```

2. Run the configuration file, build the program and install it

```
$ ./configure --prefix=/usr/local
$ make                      # -jX if you have X cores
$ make install
```

3. Make sure the program is installed and ready to run:

```
$ /usr/local/bin/ecl
```

```
ECL (Embeddable Common-Lisp) 16.0.0
Copyright (C) 1984 Taiichi Yuasa and Masami Hagiya
Copyright (C) 1993 Giuseppe Attardi
Copyright (C) 2000 Juan J. Garcia-Ripoll
Copyright (C) 2015 Daniel Kochmanski
ECL is free software, and you are welcome to redistribute it
under certain conditions; see file 'Copyright' for details.
Type :h for Help.
Top level in: #<process TOP-LEVEL>.
>
```

1.1.2 Platform specific instructions

1.1.2.1 MSVC based configuration

You need Microsoft Visual Studio 2015 or better to compile ECL, which flavor(Professional, Community, etc) does not matter.

You also need yasm (<http://yasm.tortall.net>) optionally to build gmp, fetch yasm-1.3.0-win64.exe and yasm-1.3.0-win32.exe, and put them in your system PATH directory.

In the Visual Studio's startup menu, click Developer Command Prompt for Visual Studio (<https://docs.microsoft.com/en-us/dotnet/framework/tools/developer-command-prompt-for-vs>) to open the console window. Alternatively, open the developer console from the start menu through "Visual Studio 20xx" -> "Visual Studio Tools" -> "VC" and select "x64 Native Tools Command Prompt for VS 20xx" or "x86 Native Tools Command Prompt for VS 20xx", depending on whether you want to build 32 or 64bit versions of ECL.

1. Change to the msvc directory.
2. Run nmake to build ECL.
3. Run nmake install prefix=d:\Software\ECL where the prefix is the directory where you want to install ECL.
4. If you want to build debug version, add ECL_DEBUG=1 to nmake command line.
5. If you want to build 64bit version, add ECL_WIN64=1 to nmake command line, you can also set GMP_TYPE=AMD64 to use specific assembly codes.
6. Optionally, if you want to build a self-installing executable, you can install NSIS and run nmake windows-nsi.

1.1.2.2 Android

Cross compiling ECL for Android requires first building the host ECL program. At present this host ECL needs to have the same word size and same optional capabilities (e.g. threads, C99 complex floats) as the target system. Therefore, to build the host ECL for a 32 bit ARM system, use the following commands:

```
# C99 complex numbers are not fully supported on Android
./configure ABI=32 CFLAGS="-m32 -g -O2" LDFLAGS="-m32 -g -O2" \
    --prefix='pwd'/ecl-android-host \
    --disable-c99complex
make -j9
make install
rm -r build
export ECL_TO_RUN='pwd'/ecl-android-host/bin/ecl
```

The next step is to configure the cross compilation toolchain. This requires the Android NDK version 15 or higher.

```
export NDK_PATH=/opt/android-ndk
export ANDROID_API=23
export TOOLCHAIN_PATH='pwd'/android-toolchain
${NDK_PATH}/build/tools/make_standalone_toolchain.py --arch arm --install-dir ${TOOLCH
export SYSROOT=${TOOLCHAIN_PATH}/sysroot
```

```
export PATH=${TOOLCHAIN_PATH}/bin:$PATH
```

Here, `ANDROID_API` is the minimum Android API version ECL will run on. Finally, we can build and install the target ECL:

```
# Boehm GC is not compatible with ld.gold linker, force use of ld.bfd
export LDFLAGS="--sysroot=${SYSROOT} -D__ANDROID_API__=${ANDROID_API} -fuse-ld=bfd"
export CPPFLAGS="--sysroot=${SYSROOT} -D__ANDROID_API__=${ANDROID_API} -isystem ${SYSROOT}/usr/include"
export CC=arm-linux-androideabi-clang
./configure --host=arm-linux-androideabi \
            --prefix='pwd'/ecl-android \
            --disable-c99complex \
            --with-cross-config='pwd'/src/util/android-arm.cross_config
make -j9
make install
```

Library and assets are installed in the "ecl-android" directory and are ready to run on the Android system.

1.2 Entering and leaving Embeddable Common Lisp

Embeddable Common Lisp is invoked by the command `ecl`.

```
% ecl
ECL (Embeddable Common-Lisp) 0.0e
Copyright (C) 1984 Taiichi Yuasa and Masami Hagiya
Copyright (C) 1993 Giuseppe Attardi
Copyright (C) 2000 Juan J. Garcia-Ripoll
Copyright (C) 2015 Daniel Kochmanski
ECL is free software, and you are welcome to redistribute it
under certain conditions; see file 'Copyright' for details.
Type :h for Help.  Top level.
Top level in: #<process TOP-LEVEL>.
>
```

When invoked, Embeddable Common Lisp will print the banner and initialize the system. The number in the Embeddable Common Lisp banner identifies the revision of Embeddable Common Lisp. 0.0e is the value of the function `lisp-implementation-version`.

Unless user specifies `--norc` flag when invoking the Embeddable Common Lisp, it will look for the initialization files `~/.ecl` and `~/.eclrc`. If he wants to load his own file from the current directory, then he should pass the file path to the `--load` parameter:

```
% ecl --norc --load init.lisp
```

After the initialization, Embeddable Common Lisp enters the *top-level loop* and prints the prompt `'>'`.

```
Type :h for Help.  Top level.
>
```

The prompt indicates that Embeddable Common Lisp is now ready to receive a form from the terminal and to evaluate it.

Usually, the current package (i.e., the value of `*package*`) is the user package, and the prompt appears as above. If, however, the current package is other than the user package, then the prompt will be prefixed with the package name.

```
> (in-package "CL")
#<"COMMON-LISP" package>
COMMON-LISP> (in-package "SYSTEM")
#<"SI" package>
SI>
```

To exit from Embeddable Common Lisp, call the function `ext:quit`.

```
> (quit)
%
```

Alternatively, you may type `^D`, i.e. press the key D while pressing down the control key (Ctrl).

```
> ^D

%
```

The top-level loop of Embeddable Common Lisp is almost the same as that defined in Section 20.2 of [Steele:84, see [Bibliography], page 193]. Since the input from the terminal is in line mode, each top-level form should be followed by a newline. If more than one value is returned by the evaluation of the top-level form, the values will be printed successively. If no value is returned, then nothing will be printed.

```
> (values 1 2)
1
2
> (values)

>
```

When an error is signaled, control will enter the break loop.

```
> (defun foo (x) (bar x))
foo

> (defun bar (y) (bee y y))

bar
> (foo 'lish)
Condition of type: UNDEFINED-FUNCTION
The function BEE is undefined.
```

Available restarts:

1. (RESTART-TOPLEVEL) Go back to Top-Level REPL.

```
Broken at FOO. In: #<process TOP-LEVEL>.
>>
```

'>>' in the last line is the prompt of the break loop. Like in the top-level loop, the prompt will be prefixed by the current package name, if the current package is other than the `cl-user` package.

To go back to the top-level loop, type `:q`

```
>>:q
```

```
Top level in: #<process TOP-LEVEL>.  
>
```

If more restarts are present, user may invoke them with by typing `:rN`, where `N` is the restart number. For instance to pick the restart number two, type `:r2`.

See [Section 1.3 [The break loop], page 13] for the details of the break loop.

The terminal interrupt (usually caused by typing `^C` (Control-C)) is a kind of error. It breaks the running program and calls the break level loop.

Example:

```
> (defun foo () (do () (nil)))  
foo
```

```
> (foo)  
^C
```

```
Condition of type: INTERACTIVE-INTERRUPT  
Console interrupt.
```

```
Available restarts:
```

1. (CONTINUE) CONTINUE
2. (RESTART-TOPLEVEL) Go back to Top-Level REPL.

```
Broken at FOO. In: #<process TOP-LEVEL>.  
>>
```

1.3 The break loop

1.4 Embedding ECL

1.4.1 Embedding Reference

1.4.1.1 Starting and Stopping

`int cl_boot (int argc, char **argv);` [Function]
Setup the lisp environment.

`argc` An integer with the number of arguments to this program.

`argv` A vector of strings with the arguments to this program.

Description This function must be called before any other function from the ECL library, including the creation of any lisp object or evaluating any lisp code. The only exception are `ecl_set_option` and `ecl_get_option`.

```
int cl_shutdown (void);
```

[Function]

Close the lisp environment.

Description This function must be called before exiting a program that uses the ECL environment. It performs some cleaning, including the execution of any finalizers, unloading shared libraries and deleting temporary files that were created by the compiler.

```
void ecl_set_option (int option, cl_fixnum value);
```

[Function]

Set a boot option.

option An integer from Table 1.1.

value A `cl_index` value for this option

Description This functions sets the value of different options that have to be customized *before* ECL boots. The table of options and default values [Table 1.1] shows that some of them are boolean, and some of them are unsigned integers.

We distinguish three sets of values. The first set determines whether ECL handles certain exceptions, such as access to forbidden regions of memory, interrupts via , floating point exceptions, etc.

The second set is related to the sizes of different stacks. Currently ECL uses four stacks: a bind stack for keeping assignments to special variables; a frame stack for implementing blocks, tagbodies and catch points; an interpreter stack for evaluating bytecodes, and finally the machine or C stack, of the computer we run in. We can set the expected size of these stacks, together with the size of a safety area which, if penetrated, will lead to the generation of a correctable error.

Name (ECL_OPT_*)	Type	Default	Description
INCREMENTAL_GC	boolean	TRUE	Activate generational garbage collector.
TRAP_SIGSEGV	boolean	TRUE	Capture SIGSEGV signals.
TRAP_SIGFPE	boolean	TRUE	Capture floating point exceptions.
TRAP_SIGINT	boolean	TRUE	Capture user interrupts.
TRAP_SIGILL	boolean	TRUE	Capture SIGILL exception.
TRAP_INTERRUPT_SIGNAL	boolean	TRUE	Capture the signal that implements <code>mp:interrupt-process</code> .
SIGNAL_HANDLING_THREAD	boolean	TRUE	Create a signal to capture and process asynchronous threads (See Section 3.5.2.2 [Signals and Interrupts - Asynchronous signals], page 135).
BOOTED	boolean	TRUE/FALSE	Has ECL booted (read only).
BIND_STACK_SIZE	cl_index	8192	Size of stack for binding special variables.
BIND_STACK_SAFETY_AREA	cl_index	128	Size of stack for nonlocal jumps.
FRAME_STACK_SIZE	cl_index	2048	
FRAME_STACK_SAFETY_AREA	cl_index	128	
LISP_STACK_SIZE	cl_index	32768	Size of interpreter stack.
LISP_STACK_SAFETY_AREA	cl_index	128	Size of C stack in bytes. The effect and default value of this option depends on the operating system. On Unix, the default is 0 which means that ECL will use the stack size provided by the OS. If set to a non-default value, ECL will set the stack size to the given value unless the stack size provided by the OS is already large enough. On Windows, the stack size is set at build time and cannot be changed at run-time. Here, we use a default of 1 MiB. For other operating systems, it is up to the user to set this value to the available stack size so that ECL can reliably detect stack overflows.
C_STACK_SIZE	cl_index	0 or 1048576	
C_STACK_SAFETY_AREA	cl_index	4192	If nonzero, specify the unix signal which is used to communicate between different Lisp threads.
THREAD_INTERRUPT_	unsigned	0	
SIGNAL	int		

Table 1.1: Boot options for embedded ECL

`cl_fixnum ecl_get_option (int option);` [Function]

Read the value of a boot option.

option An integer from Table 1.1.

Description This functions reads the value of different options that have to be customized *before* ECL boots. The table of options and default values is Table 1.1.

`bool ecl_import_current_thread (cl_object name, cl_object bindings);` [Function]

Import an external thread in the Lisp environment.

name Thread name.

bindings Unused (specifying initial bindings for external threads is not supported currently)

returns True if the thread was successfully imported, false otherwise.

Description External threads, i.e. threads which are not created in the Lisp world using the routines described in Section 3.4.2 [Processes (native threads)], page 122, need to be imported with `ecl_import_current_thread` before Lisp code can be executed.

See also `ecl_release_current_thread`

`void ecl_release_current_thread (void);` [Function]

Release an external thread imported with `ecl_import_current_thread`. Must be called before thread exit to prevent memory leaks.

`ECLDIR` [Environment variable]

Specify a non-standard installation directory.

Description ECL includes various files for external modules (e.g. asdf, sockets), character encodings or documentation strings. The installation directory for these files is chosen during build time by the configure script. If the directory is moved to a different place, the `ECLDIR` environment variable should be updated accordingly. Note that the contents of the variable are parsed as a Common Lisp pathname, thus it must end with a slash.

1.4.1.2 Catching Errors and Managing Interrupts

`ECL_CATCH_ALL` [Macro]

Create a protected region.

C Macro

```
cl_env_ptr env = ecl_process_env();
ECL_CATCH_ALL_BEGIN(env) {
    /*
     * Code that is protected. Uncaught lisp conditions, THROW,
     * signals such as SIGSEGV and SIGBUS may cause jump to
     * this region.
     */
} ECL_CATCH_ALL_IF_CAUGHT {
```



```

    /*
     * If the exception, lisp condition or other control transfer
     * is caught, this code is executed.
     */
} ECL_CATCH_ALL_END
/*
 * In all cases we exit here.
 */

```

Description This is a set of three macros that create an `unwind-protect` region that prevents any nonlocal transfer of control to outer loops. In the Lisp speak, the previous code is equivalent to

```

(block nil
  (unwind-protect
    (progn
      ;; Code that is protected
    )
    (return nil)))

```

As explained in `ECL_UNWIND_PROTECT`, it is normally advisable to set up an `unwind-protect` frame to avoid the embedded lisp code to perform arbitrary transfers of control.

See also `ECL_UNWIND_PROTECT`

`ECL_UNWIND_PROTECT` [Macro]
Create a protected region.

C Macro

```

cl_env_ptr env = ecl_process_env();
ECL_UNWIND_PROTECT_BEGIN(env) {
  /*
   * Code that is protected. Uncaught lisp conditions, THROW,
   * signals such as SIGSEGV and SIGBUS may cause jump to
   * this region.
   */
} ECL_UNWIND_PROTECT_EXIT {
  /*
   * If the exception, lisp condition or other control transfer
   * is caught, this code is executed. After this code, the
   * process will jump to the original destination of the
   * THROW, GOTO or other control statement that was interrupted.
   */
} ECL_UNWIND_PROTECT_END
/*
 * We only exit here if NO nonlocal jump was interrupted.
 */

```

Description When embedding ECL it is normally advisable to set up an `unwind-protect` frame to avoid the embedded lisp code to perform arbitrary

transfers of control. Furthermore, the unwind protect form will be used in at least in the following occasions:

- In a normal program exit, caused by `ext:quit`, ECL unwinds up to the outermost frame, which may be an `ECL_CATCH_ALL` or `ECL_UNWIND_PROTECT` macro.

Besides this, normal mechanisms for exit, such as `ext:quit`, and uncaught exceptions, such as serious signals (See Section 3.5.2.1 [Signals and Interrupts - Synchronous signals], page 134), are best handled using `unwind-protect` blocks.

See also `ECL_CATCH_ALL`

`ecl_clear_interrupts ()` [Macro]
Clear all pending signals and exceptions.

Description This macro clears all pending interrupts.

See also `ecl_disable_interrupts` and `ecl_enable_interrupts`.

`ecl_disable_interrupts ()` [Macro]
Postpone handling of signals and exceptions.

Description This macro sets a thread-local flag indicating that all received signals should be queued for later processing. Note that it is not possible to execute lisp code while interrupts are disabled in this way. For this purpose, use the `mp:without-interrupts` macro. Every call to `ecl_disable_interrupts` must be followed by a corresponding call to `ecl_enable_interrupts`, otherwise race conditions will appear.

See also `ecl_enable_interrupts` and `ecl_clear_interrupts`.

`ecl_enable_interrupts ();` [Macro]
Activate handling of signals and exceptions.

Description This macro sets a thread-local flag indicating that all received signals can be handled. If there are any pending signals, they will be immediately processed.

See also `ecl_disable_interrupts` and `ecl_clear_interrupts`.

`ECL_WITH_LISP_FPE` [Macro]
Execute Lisp code with correct floating point environment

Description Unless floating point exceptions are disabled (via the `--without-fpe` configure option or `ECL_OPT_TRAP_SIGFPE` runtime option), ECL will change the floating point environment when booting. This macro allows for execution of Lisp code while saving and later restoring the floating point environment of surrounding C code so that changes in the floating point environment don't leak outside.

`ECL_WITH_LISP_FPE` can be also used before ECL has booted or before an external thread has been imported.

Example

```
#include <ecl/ecl.h>
#include <stdio.h>

int main(int argc, char **argv) {
    ECL_WITH_LISP_FPE_BEGIN {
```

```

    cl_boot(argc, argv);
} ECL_WITH_LISP_FPE_END;

double a = 1.0 / 0.0;
double b;

ECL_WITH_LISP_FPE_BEGIN {
    cl_object form = ecl_read_from_cstring("(handler-case"
                                           "(/ 1d0 0d0)"
                                           "(division-by-zero () 0d0))");
    b = ecl_to_double(si_safe_eval(3, form, ECL_NIL, ECL_NIL));
} ECL_WITH_LISP_FPE_END;

printf("%g %g\n", a, b);

cl_shutdown();
return 0;
}

```

will output

inf 0

See also `ext:trap-fpe`

2 Standards

2.1 Overview

2.1.1 Reading this manual

Common Lisp users

Embeddable Common Lisp supports all Common-Lisp data types exactly as defined in the [ANSI, see [Bibliography], page 193]. All functions and macros are expected to behave as described in that document and in the HyperSpec [HyperSpec, see [Bibliography], page 193] which is the online version of [ANSI, see [Bibliography], page 193]. In other words, the Standard is the basic reference for Common Lisp and also for Embeddable Common Lisp, and this part of the manual just complements it, describing implementation-specific features such as:

- Platform dependent limits.
- Behavior which is marked as *implementation specific* in the standard.
- Some corner cases which are not described in [ANSI, see [Bibliography], page 193].
- The philosophy behind certain implementation choices, etc.

In order to aid in locating these differences, this first part of the manual copies the structure of the ANSI Common-Lisp standard, having the same number of chapters, each one with a set of sections documenting the implementation-specific details.

C/C++ programmers

The second goal of this document is to provide a reference for C programmers that want to create, manipulate and operate with Common Lisp programs at a lower level, or simply embedding Embeddable Common Lisp as a library.

The C/C++ reference evolves in parallel with the Common Lisp one, in the form of one section with the name "C Reference" for each chapter of the ANSI Common-Lisp standard. Much of what is presented in those sections is redundant with the Common Lisp specification. In particular, there is a one-to-one mapping between types and functions which should be obvious given the rules explained in the next section *C Reference*.

We must remark that the reference in this part of the manual is not enough to know how to embed Embeddable Common Lisp in a program. In practice the user or developer will also have to learn how to build programs (Section 3.1 [System building], page 85), interface with foreign libraries (Section 3.3 [Foreign Function Interface], page 98), manage memory (Section 3.6 [Memory Management], page 138), etc. These concepts are explained in a different (Section 1.4 [Embedding ECL], page 13) part of the book.

2.1.2 C Reference

One type for everything: `cl_object`

ECL is designed around the basic principle that Common Lisp already provides everything that a programmer could need, orienting itself around the creation and manipulation of

Common Lisp objects: conses, arrays, strings, characters, ... When embedding ECL there should be no need to use other C/C++ types, except when interfacing data to and from those other languages.

All Common Lisp objects are represented internally through the same C type, `cl_object`, which is either a pointer to a union type or an integer, depending on the situation. While the inner guts of this type are exposed through various headers, the user should never rely on these details but rather use the macros and functions that are listed in this manual.

There are two types of Common Lisp objects: immediate and memory allocated ones. Immediate types fit in the bits of the `cl_object` word, and do not require the garbage collector to be created. The list of such types may depend on the platform, but it includes at least the `fixnum` and `character` types.

Memory allocated types on the other hand require the use of the garbage collector to be created. ECL abstracts this from the user providing enough constructors, either in the form of Common Lisp functions (`cl_make_array`, `cl_complex`,...), or in the form of C/C++ constructors (`ecl_make_symbol`, etc).

Memory allocated types must always be kept alive so that the garbage collector does not reclaim them. This involves referencing the object from one of the places that the collector scans:

- The fields of an object (array, structure, etc) which is itself alive.
- A special variable or a constant.
- The C stack (i.e. automatic variables in a function).
- Global variables or pointers that have been registered with the garbage collector.

For memory allocation details See Section 3.6 [Memory Management], page 138. For object implementation details See Section 4.4 [Manipulating Lisp objects], page 154.

Naming conventions

As explained in the introduction, each of the chapters in the Common Lisp standard can also be implemented using C functions and types. The mapping between both languages is done using a small set of rules described below.

- Functions in the Common Lisp (`cl`) package are prefixed with the characters `cl_`, functions in the System (`si`) and Extensions (`ext`) package are prefix with `si_`, etc, etc.
- If a function takes only a fixed number of arguments, it is mapped to a C function with also a fixed number of arguments. For instance, `cos` maps to `cl_object cl_cos(cl_object)`, which takes a single Lisp object and returns a Lisp object of type `float`.
- If the function takes a variable number of arguments, its signature consists on an integer with the number of arguments and zero or more of required arguments and then a C `vararg`. This is the case of `cl_object cl_list(cl_narg nargs, ...)`, which can be invoked without arguments, as in `cl_list(0)`, with one, `cl_list(1, a)`, etc.
- Functions return at least one value, which is either the first value output by the function, or `nil`. The extra values may be retrieved immediately after the function call using the function `ecl_nth_value`.

In addition to the Common Lisp core functions (`cl_*`), there exist functions which are devoted only to C/C++ programming, with tasks such as coercion of objects to and from C types, optimized functions, inlined macroexpansions, etc. These functions and macros typically carry the prefix `ecl_` or `ECL_` and only return one value, if any.

Most (if not all) Common Lisp functions and constructs available from C/C++ are available in “ANSI Dictionary” sections which are part of the [Chapter 2 [Standards], page 21] entries.

Only in Common Lisp

Some parts of the language are not available as C functions, even though they can be used in Common Lisp programs. These parts are either marked in the “ANSI Dictionary” sections using the tag *Only in Common Lisp*, or they are simply not mentioned (macros and special constructs). This typically happens with non-translatable constructs such as

- Common Lisp macros such as `with-open-files`
- Common Lisp special forms, such as `cond`
- Common Lisp generic functions, which cannot be written in C because of their dynamical dispatch and automatic redefinition properties.

In most of those cases there exist straightforward alternatives using the constructs and functions in ECL. For example, `unwind-protect` can be implemented using a C macro which is provided by ECL

```
cl_env_ptr env = ecl_process_env();
ECL_UNWIND_PROTECT_BEGIN(env) {
    /* protected code goes here */
} ECL_UNWIND_PROTECT_EXIT {
    /* exit code goes here */
} ECL_UNWIND_PROTECT_END;
```

Common Lisp generic functions can be directly accessed using `funcall` or `apply` and the function name, as shown in the code below

```
cl_object name = ecl_make_symbol("MY-GENERIC-FUNCTION", "CL-USER");
cl_object output = cl_funcall(2, name, argument);
```

Identifying these alternatives requires some knowledge of Common Lisp, which is why it is recommended to approach the embeddable components in ECL only when there is some familiarity with the language.

2.2 Evaluation and compilation

2.2.1 Compiler declaration optimize

The `optimize` declaration includes three concepts: `debug`, `speed`, `safety` and `space`. Each of these declarations can take one of the integer values 0, 1, 2 and 3. According to these values, the implementation may decide how to compile or interpret a given lisp form.

ECL currently does not use all these declarations, but some of them definitely affect the speed and behavior of compiled functions. For instance, the `debug` declaration, as shown in Table 2.1, the value of debugging is zero, the function will not appear in the debugger and, if redefined, some functions might not see the redefinition.

Behavior	0	1	2	3
Compiled functions in the same source file are called directly	Y	Y	N	N
Compiled function appears in debugger backtrace	N	N	Y	Y
All functions get a global entry (SI:C-LOCAL is ignored)	N	N	Y	Y

Table 2.1: Behavior for different levels of `debug`

A bit more critical is the value of `safety` because as shown in Table 2.2, it may affect the safety checks generated by the compiler. In particular, in some circumstances the compiler may assume that the arguments to a function are properly typed. For instance, if you compile with a low value of `safety`, and invoke `rplaca` with an object which is not a list, the consequences are unspecified.

Behavior	0	1	2	3
The compiler generates type checks for the arguments of a lambda form, thus enforcing any type declaration written by the user.	N	Y	Y	Y
The value of an expression or a variable declared by the user is assumed to be right.	Y	Y	N	N
We believe type declarations and type inference and, if the type of a form is inferred to be right for a function, slot accessor, etc, this may be inlined. Affects functions like <code>car</code> , <code>cdr</code> , etc	Y	Y	N	N
We believe types defined before compiling a file do not change before the compiled code is loaded.	Y	Y	N	N
Arguments in a lisp form are assumed to have the appropriate types so that the form will not fail.	Y	N	N	N
The slots or fields in a lisp object are accessed directly without type checks even if the type of the object could not be inferred (see line above). Affects functions like <code>pathname-type</code> , <code>car</code> , <code>rest</code> , etc.	Y	N	N	N

Table 2.2: Behavior for different levels of `safety`

2.2.2 `declaim` and `proclaim`

Declarations established with `proclaim` stay in force indefinitely. Declarations established with `declaim` in a file do not persist after the file has been compiled. However, they are established with `proclaim` at load time when the compiled file is loaded. This means that when compiling two files, `declaim` declarations in the first file will not be in force when compiling the second file unless the first file was loaded before the second one was compiled.

2.2.3 C Reference

`cl_env_ptr ecl_process_env ()` [C/C++ identifier]
 ECL stores information about each thread on a dedicated structure, which is the process environment. A pointer to this structure can be retrieved using the function or macro above. This pointer can be used for a variety of tasks, such as defining special variable bindings, controlling interrupts, retrieving function output values, etc.

2.2.3.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function
compile	[Only in Common Lisp]
eval	cl_object cl_eval (cl_object form) - DEPRECATED , see [si_safe_eval], page 164
macroexpand	cl_object cl_macroexpand(cl_narg nargs, cl_object form, ...)
macroexpand-1	cl_object cl_macroexpand_1(cl_narg nargs, cl_object form, ...)
proclaim	[Only in Common Lisp]
special-operator-p	cl_object cl_special_operator_p(cl_object form)
constantp	cl_object cl_constantp (cl_narg nargs, cl_object arg, ...)

2.3 Types and classes

ECL defines the following additional built-in classes in the `cl` package:

- `single-float`
- `double-float`
- `long-float`

2.3.1 C Reference

2.3.1.1 ANSI Dictionary

Common Lisp and C equivalence

Synopsis

Lisp symbol	C function
coerce	cl_object cl_coerce(cl_object object, cl_object result_type)
subtypep	cl_object cl_subtypep(cl_narg nargs, cl_object type1, cl_object type2, ...)
type-of	cl_object cl_type_of(cl_object object)
typep	cl_object cl_typep(cl_narg nargs, cl_object object, cl_object type_specifier, ...)
type-error-datum	[Only in Common Lisp]
type-error-expected-type	[Only in Common Lisp]

2.4 Data and control flow

2.4.1 Shadowed bindings

ANSI doesn't specify what should happen if any of the `let`, `flet` and `labels` special operators contain many bindings sharing the same name. Because the behavior varies between the implementations and the programmer can't rely on the spec ECL signals an error if such situation occur.

Moreover, while ANSI defines lambda list parameters in the terms of `let*`, when used in function context programmer can't provide an initialization forms for required parameters. If required parameters share the same name an error is signaled.

Described behavior is present in ECL since version 16.0.0. Previously the `let` operator were using first binding. Both `flet` and `labels` were signaling an error if C compiler was used and used the last binding as a visible one when the byte compiler was used.

2.4.2 Minimal compilation

Former versions of ECL, as well as many other lisps, used linked lists to represent code. Executing code thus meant traversing these lists and performing code transformations, such as macro expansion, every time that a statement was to be executed. The result was a slow and memory hungry interpreter.

Beginning with version 0.3, ECL was shipped with a bytecodes compiler and interpreter which circumvent the limitations of linked lists. When you enter code at the lisp prompt, or when you load a source file, ECL begins a process known as minimal compilation. Barely this process consists on parsing each form, macroexpanding it and translating it into an intermediate language made of bytecodes.

The bytecodes compiler is implemented in `src/c/compiler.d`. The main entry point is the lisp function `si::make-lambda`, which takes a name for the function and the body of the lambda lists, and produces a lisp object that can be invoked. For instance,

```
> (defvar fun (si::make-lambda 'f '((x) (1+ x))))
*FUN*
> (funcall fun 2)
3
```

ECL can only execute bytecodes. When a list is passed to `eval` it must be first compiled to bytecodes and, if the process succeeds, the resulting bytecodes are passed to the interpreter. Similarly, every time a function object is created, such as in `defun` or `defmacro`, the compiler processes the lambda form to produce a suitable bytecodes object.

The fact that ECL performs this eager compilation means that changes on a macro are not immediately seen in code which was already compiled. This has subtle implications. Take the following code:

```
> (defmacro f (a b) '(+ ,a ,b))
F
> (defun g (x y) (f x y))
G
> (g 1 2)
3
> (defmacro f (a b) '(- ,a ,b))
```

```
F
> (g 1 2)
3
```

The last statement always outputs 3 while in former implementations based on simple list traversal it would produce -1.

2.4.3 Function types

Functions in ECL can be of two types: they are either compiled to bytecodes or they have been compiled to machine code using a lisp to C translator and a C compiler. To the first category belong function loaded from lisp source files or entered at the toplevel. To the second category belong all functions in the ECL core environment and functions in files processed by `compile` or `compile-file`.

The output of `(symbol-function fun)` is one of the following:

- a function object denoting the definition of the function `fun`,
- a list of the form `(macro . function-object)` when `fun` denotes a macro,
- or simply `'special`, when `fun` denotes a special form, such as `block`, `if`, etc.

ECL usually keeps the source code of a function unless the global variable `si:*keep-definitions*` was false when the function was translated into bytecodes. Therefore, if you don't need to use `compile` and `disassemble` on defined functions, you should issue `(setf si:*keep-definitions* nil)` at the beginning of your session.

si:*keep-definitions* [Variable]
If set to `t` ECL will preserve the compiled function source code for disassembly and recompilation.

In Table 2.3 we list all Common Lisp values related to the limits of functions.

call-arguments-limit	65536
lambda-parameters-limit	call-arguments-limit
multiple-values-limit	64
lambda-list-keywords	(&optional &rest &key &allow-other-keys &aux &whole &environment &body)

Table 2.3: Function related constants

2.4.4 C Calling conventions

ECL is implemented using either a C or a C++ compiler. This is not a limiting factor, but imposes some constraints on how these languages are used to implement functions, multiple values, closures, etc. In particular, while C functions can be called with a variable number of arguments, there is no facility to check how many values were actually passed. This forces us to have two types of functions in ECL

- Functions that take a fixed number of arguments have a simple C signature, with all arguments being properly declared, as in `cl_object cl_not(cl_object arg1)`.

- Functions with a variable number of arguments, such as those accepting `&optional`, `&rest` or `&key` arguments, must take as first argument the number of remaining ones, as in `cl_object cl_list(cl_narg nargs, ...)`. Here *narg* is the number of supplied arguments.

The previous conventions set some burden on the C programmer that calls ECL, for she must know the type of function that is being called and supply the right number of arguments. This burden disappears for Common Lisp programmers, though.

As an example let us assume that the user wants to invoke two functions which are part of the ANSI [ANSI, see [Bibliography], page 193] standard and thus are exported with a C name. The first example is `cl_cos`, which takes just one argument and has a signature `cl_object cl_cos(cl_object)`.

```
#include <math.h>
...
cl_object angle = ecl_make_double_float(M_PI);
cl_object c = cl_cos(angle);
printf("\nThe cosine of PI is %g\n", ecl_double_float(c));
```

The second example also involves some Mathematics, but now we are going to use the C function corresponding to `+`. As described in Section 2.10.6.5 [Numbers - ANSI dictionary], page 45, the C name for the plus operator is `cl_P` and has a signature `cl_object cl_P(cl_narg nargs, ...)`. Our example now reads as follows

```
cl_object one = ecl_make_fixnum(1);
cl_object two = cl_P(2, one, one);
cl_object three = cl_P(3, one, one, one);
printf("\n1 + 1 is %d\n", ecl_fixnum(two));
printf("\n1 + 1 + 1 is %d\n", ecl_fixnum(three));
```

Note that most Common Lisp functions will not have a C name. In this case one must use the symbol that names them to actually call the functions, using `cl_funcall` or `cl_apply`. The previous examples may thus be rewritten as follows

```
/* Symbol + in package CL */
cl_object plus = ecl_make_symbol("+", "CL");
cl_object one = ecl_make_fixnum(1);
cl_object two = cl_funcall(3, plus, one, one);
cl_object three = cl_funcall(4, plus, one, one, one);
printf("\n1 + 1 is %d\n", ecl_fixnum(two));
printf("\n1 + 1 + 1 is %d\n", ecl_fixnum(three));
```

Another restriction of C and C++ is that functions can only take a limited number of arguments. In order to cope with this problem, ECL uses an internal stack to pass any argument above a hardcoded limit, `ECL_C_CALL_ARGUMENTS_LIMIT`, which is as of this writing 63. The use of this stack is transparently handled by the Common Lisp functions, such as `apply`, `funcall` and their C equivalents, and also by a set of macros, `cl_va_arg`, which can be used for coding functions that take an arbitrary name of arguments.

2.4.5 C Reference

```
void ecl_bds_bind (cl_env_ptr cl_env, cl_object var, cl_object value); [Function]
```

`void ecl_bds_push (cl_env_ptr cl_env, cl_object var);` [Function]

Bind a special variable

Description Establishes a variable binding for the symbol *var* in the Common Lisp environment *env*, assigning it *value*.

This macro or function is the equivalent of `let*` and `let`.

`ecl_bds_push` does a similar thing, but reuses the old value of the same variable. It is thus the equivalent of `(let ((var var)) ...)`

Every variable binding must undone when no longer needed. It is best practice to match each call to `ecl_bds_bind` by another call to `ecl_bds_unwind1` in the same function.

`void ecl_bds_unwind1 (cl_env_ptr cl_env);` [Function]

`void ecl_bds_unwind_n (cl_env_ptr cl_env, int n);` [Function]

Undo one variable binding

Description `ecl_bds_unwind1` undoes the outermost variable binding, restoring the original value of the symbol in the process.

`ecl_bds_unwind_n` does the same, but for the *n* last variables.

Every variable binding must undone when no longer needed. It is best practice to match each call to `ecl_bds_bind` by another call to `ecl_bds_unwind1` in the same function.

`cl_object ecl_setq (cl_env_ptr cl_env, cl_object var, cl_object value);` [Function]

C equivalent of `setq`

Description Assigns *value* to the special variable denoted by the symbol *var*, in the Common Lisp environment *cl_env*.

This function implements a variable assignment, not a variable binding. It is thus the equivalent of `setq`.

`cl_object ecl_symbol_value (cl_env_ptr cl_env, cl_object var);` [Function]

Description Retrieves the value of the special variable or constant denoted by the symbol *var*, in the Common Lisp environment *cl_env*.

This function implements the equivalent of `symbol-value` and works both on special variables and constants.

If the symbol is not bound, an error is signaled.

`typedef struct { ... } ecl_va_list[1];` [Macro]

`ecl_va_start (ecl_va_list arglist, last_argument, nargs, n_ordinary);` [Macro]

`cl_object ecl_va_arg (ecl_va_list arglist);` [Macro]

`cl_object ecl_va_end (ecl_va_list arglist);` [Macro]

Accepting a variable number of arguments

Description The macros above are used to code a function that accepts an arbitrary number of arguments. We will describe them in a practical example

```
cl_object my_plus(cl_narg nargs, cl_object required1, ...)
```

```

{
    cl_env_ptr env = ecl_process_env();
    cl_object other_value;
    ecl_va_list varargs;
    ecl_va_start(varargs, required1, nargs, 1);
    while (nargs > 1) {
        cl_object other_value = ecl_va_arg(varargs);
        required1 = ecl_plus(required1, other_value);
    }
    ecl_va_end(varargs);
    ecl_return1(env, required1);
}

```

The first thing to do is to declare the variable that will hold the arguments. This is *varargs* in our example and it has the type `ecl_va_list`.

This arguments list is initialized with the `ecl_va_start` macro, based on the supplied number of arguments, *nargs*, the number of required arguments which are passed as ordinary C arguments (1 in this case), the last such ordinary arguments, *required*, and the buffer for the argument list, *varargs*.

Once *varargs* has been initialized, we can retrieve these values one by one using `ecl_va_arg`. Note that the returned value always has the type `cl_object`, for it is always a Common Lisp object.

The last statement before returning the output of the function is `ecl_va_end`. This macro performs any required cleanup and should never be omitted.

```

cl_object ecl_nvalues (cl_env_ptr env); [Function]
cl_object ecl_nth_value (cl_env_ptr env, int n); [Function]

```

Accessing output values

Description Common Lisp functions may return zero, one or more values. In ECL, the first two cases do not require any special manipulation, as the C function returns either `nil` or the first (zeroth) value directly. However, if one wishes to access additional values from a function, one needs to use these two macros or functions

- `ecl_nvalues(env)` returns the number of values that the function actually outputs. The single argument is the lisp environment. This value is larger or equal to 0 and smaller than `ECL_MULTIPLE_VALUES_LIMIT`.
- Once we know the number of return values, they can be directly accessed using the function `ecl_nth_value(env,n)`, where *n* is a number larger than or equal to 1, and smaller than `ECL_MULTIPLE_VALUES_LIMIT`, which must correspond to a valid output value. No checking is done.

Note that in both cases these macros and functions have to be used right after the Lisp function was called. This is so because other Lisp functions might destroy the content of the return stack.

Example A C/C++ excerpt:

```

cl_env_ptr env = ecl_process_env();
cl_object a = ecl_make_fixnum(13);
cl_object b = ecl_make_fixnum(6);

```

```
cl_object modulus = cl_floor(2, a, b);
cl_object remainder = ecl_nth_value(env, 1);
```

The somewhat equivalent Common Lisp code:

```
(multiple-value-bind (modulus equivalent)
  (floor 13 6))
```

```
ecl_return0 (cl_env_ptr cl_env); [Macro]
```

```
ecl_return1 (cl_env_ptr cl_env, cl_object value1); [Macro]
```

```
ecl_return2 (cl_env_ptr cl_env, cl_object value1, cl_object value2); [Macro]
```

```
ecl_return3 (cl_env_ptr cl_env, cl_object value1, cl_object value2,
  cl_object value3); [Macro]
```

Returning multiple values

Description Returns *N* values from a C/C++ function in a way that a Common Lisp function can recognize and use them. The 0-th value is returned directly, while values 1 to *N* are stored in the Common Lisp environment *cl_env*. This macro has to be used from a function which returns an object of type *cl_object*.

```
ECL_BLOCK_BEGIN [Macro]
```

```
ECL_BLOCK_BEGIN(env,code) {
```

```
  } ECL_BLOCK_END;
```

Description *ECL_BLOCK_BEGIN* establishes a block named *code* that becomes visible for the Common Lisp code. This block can be used then as a target for *cl_return*.

env must be the value of the current Common Lisp environment, obtained with *ecl_process_env*.

The C/C++ program has to ensure that the code in *ECL_BLOCK_END* gets executed, avoiding a direct exit of the block via *goto* or a C/C++ return.

```
ECL_CATCH_BEGIN [Macro]
```

```
ECL_CATCH_BEGIN(env,tag) {
```

```
  } ECL_CATCH_END;
```

Description *ECL_CATCH_BEGIN* establishes a destination for *throw* with the code given by *tag*.

env must be the value of the current Common Lisp environment, obtained with *ecl_process_env*.

The C/C++ program has to ensure that the code in *ECL_CATCH_END* gets executed, avoiding a direct exit of the catch block via *goto* or a C/C++ return.

```
ECL_UNWIND_PROTECT_BEGIN [Macro]
```

C macro for unwind-protect

Synopsis

```
ECL_UNWIND_PROTECT_BEGIN(env) {
```

```
  } ECL_UNWIND_PROTECT_EXIT {
```

```
} ECL_UNWIND_PROTECT_END;
```

Description ECL_UNWIND_PROTECT_BEGIN establishes two blocks of C code that work like the equivalent ones in Common Lisp: a protected block, contained between the "BEGIN" and the "EXIT" statement, and the exit block, appearing immediately afterwards. The form guarantees that the exit block is always executed, even if the protected block attempts to exit via some nonlocal jump construct (**throw**, **return**, etc).

env must be the value of the current Common Lisp environment, obtained with `ecl_process_env`.

The utility of this construct is limited, for it only protects against nonlocal exits caused by Common Lisp constructs: it does not interfere with C **goto**, **return** or with C++ exceptions.

2.4.5.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function or constant
apply	cl_object cl_apply(cl_narg nargs, cl_object function, ...)
call-arguments-limit	ECL_CALL_ARGUMENTS_LIMIT
compiled-function-p	cl_object cl_compiled_function_p(cl_object object)
complement	cl_object cl_complement(cl_object function)
constantly	cl_object cl_constantly(cl_object value)
every	cl_object cl_every(cl_narg nargs, cl_object predicate, ...)
eq	cl_object cl_eq(cl_object x, cl_object y)
eql	cl_object cl_eql(cl_object x, cl_object y)
equal	cl_object cl_equal(cl_object x, cl_object y)
equalp	cl_object cl_equalp(cl_object x, cl_object y)
fboundp	cl_object cl_fboundp(cl_object function_name)
fdefinition	cl_object cl_fdefinition(cl_object function_name)
(setf fdefinition)	cl_object si_fset(cl_narg nargs, cl_object function_name, cl_object definition, ...)
fmakunbound	cl_object cl_fmakunbound(cl_object function_name)
funcall	cl_object cl_funcall(cl_narg nargs, cl_object function, ...)
function-lambda-expression	cl_object cl_function_lambda_expression(cl_object function)
functionp	cl_object cl_functionp(cl_object object)
get-setf-expansion	cl_object cl_get_setf_expansion(cl_narg nargs, cl_object place, ...)
identity	cl_object cl_identity(cl_object x)
let, let*	cl_object ecl_bds.bind(cl_env_ptr env, cl_object symbol, cl_object value)
lambda-parameters-limit	ECL_LAMBDA_PARAMETERS_LIMIT
multiple-values-limit	ECL_MULTIPLE_VALUES_LIMIT
not	cl_object cl_not(cl_object object)
notevery	cl_object cl_notevery(cl_narg nargs, cl_object predicate, ...)
notany	cl_object cl_notany(cl_narg nargs, cl_object predicate, ...)
set	cl_object cl_set(cl_object symbol, cl_object value)

setq	cl-object ecl_setq(cl_env_ptr env, cl-object symbol, cl-object value)
symbol-value	cl-object ecl_symbol_value(cl_env_ptr env, cl-object symbol)
some	cl-object cl_some(cl_narg nargs, cl-object predicate, ...)
values-list	cl-object cl_values_list(cl-object list)

2.5 Objects

2.5.1 C Reference

2.5.1.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function
function-keywords	[Only in Common Lisp]
ensure-generic-function	cl-object cl_ensure_generic_function(cl_narg nargs, cl-object function_name, ...)
allocate-instance	[Only in Common Lisp]
reinitialize-instance	[Only in Common Lisp]
shared-initialize	[Only in Common Lisp]
update-instance-for-different-class	[Only in Common Lisp]
update-instance-for-redefined-class	[Only in Common Lisp]
change-class	[Only in Common Lisp]
slot-boundp	cl-object cl_slot_boundp(cl-object instance, cl-object slot_name)
slot-exists-p	cl-object cl_slot_exists_p(cl-object instance, cl-object slot_name)
slot-makunbound	cl-object cl_slot_makunbound(cl-object instance, cl-object slot_name)
slot-missing	[Only in Common Lisp]
slot-unbound	[Only in Common Lisp]
slot-value	cl-object cl_slot_value(cl-object instance, cl-object slot_name)
method-qualifiers	[Only in Common Lisp]
no-applicable-method	[Only in Common Lisp]
no-next-method	[Only in Common Lisp]
remove-method	[Only in Common Lisp]
make-instance	[Only in Common Lisp]
make-instances-obsolete	[Only in Common Lisp]
make-load-form	[Only in Common Lisp]
make-load-form-saving-slots	cl-object cl_make_load_form_saving_slots(cl_narg nargs, cl-object object, ...)
find-class	cl-object cl_find_class(cl_narg nargs, cl-object symbol, ...)
compute-applicable-methods	[Only in Common Lisp]
find-method	[Only in Common Lisp]
add-method	[Only in Common Lisp]

<code>initialize-instance</code>	[Only in Common Lisp]
<code>class-name</code>	[Only in Common Lisp]
<code>(setf class-name)</code>	[Only in Common Lisp]
<code>class-of</code>	<code>cl-object cl-class-of(cl-object object)</code>
<code>unbound-slot-instance</code>	[Only in Common Lisp]

2.6 Structures

2.6.1 Redefining a `defstruct` structure

ANSI Common-Lisp says that consequences of redefining a `defstruct` are undefined. ECL defines this behavior to signal an error if the new structure is not compatible. Structures are incompatible when:

They have a different number of slots

This is particularly important for other structures which could have included the current one and for already defined instances.

Slot name, type or offset is different

Binary compatibility between old and new instances.

2.6.2 C Reference

2.6.2.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function
<code>copy-structure</code>	<code>cl-object cl-copy_structure(cl-object structure)</code>

2.7 Conditions

2.7.1 C Reference

<code>ECL_HANDLER_CASE</code>	[Macro]
C macro for handler-case	

Synopsis

```
ECL_HANDLER_CASE_BEGIN(env,names) {

    } ECL_HANDLER_CASE(n,condition) { {

    } ECL_HANDLER_CASE_END;
```

Description `ECL_HANDLER_CASE_BEGIN` runs a block of C code with a set of error handlers bound to the names given by the list *names*. The subsequent `ECL_HANDLER_CASE` statements specify what to do when the *n*-th type of conditions is found, where *n* is an integer denoting the position of the name in the list *names*.

When a condition is signaled, ECL scans the list of signal handlers, looking for matches based on `typep`. If the match with the highest precedence belongs to the list *names*,

ECL will perform a non-local transfer of control to the appropriate `ECL_HANDLER_CASE`, passing it a *condition* object as unique argument.

The following example shows how to establish a handler for `error` conditions. Note how the first value to `ECL_HANDLER_CASE` matches the position of the condition name in the list:

```
cl_object error = ecl_make_symbol("ERROR","CL");
ECL_HANDLER_CASE_BEGIN(the_env, ecl_list1(error)) {
    /* This form is evaluated with bound handlers */
    output = cl_eval(1, form);
} ECL_HANDLER_CASE(1, condition) {
    /* This code is executed when an error happens */
    /* We just return the error that took place */
    output = condition;
} ECL_HANDLER_CASE_END;
```

ECL_RESTART_CASE

[Macro]

C macro for restart-case

Synopsis

```
ECL_RESTART_CASE_BEGIN(env,names) {

    } ECL_RESTART_CASE(n,args) { {

    } ECL_RESTART_CASE_END;
```

Description `ECL_RESTART_CASE_BEGIN` runs a block of C code with a set of restarts bound to the names given by the list *names*. The subsequent `ECL_RESTART_CASE` statements specify what to do when the *n*-th restart is invoked, where *n* is an integer denoting the position of the name in the list *names*.

When the restart is invoked, it can receive any number of arguments, which are grouped in a list and stored in a new variable created with the name *args*.

The following example shows how to establish an *abort* and a *use-value* restart. Note how the first value to `ECL_RESTART_CASE` matches the position of the restart name in the list:

```
cl_object abort = ecl_make_symbol("ABORT","CL");
cl_object use_value = ecl_make_symbol("USE-VALUE","CL");
ECL_RESTART_CASE_BEGIN(the_env, cl_list(2, abort, use_value)) {
    /* This form is evaluated with bound restarts */
    output = cl_eval(1, form);
} ECL_RESTART_CASE(1, args) {
    /* This code is executed when the 1st restart (ABORT) is invoked */
    output = ECL_NIL;
} ECL_RESTART_CASE(2, args) {
    /* This code is executed when the 2nd restart (USE-VALUE) is invoked */
    output = ECL_CAR(args);
} ECL_RESTART_CASE_END;
```

2.7.1.1 ANSI dictionary

Common Lisp and C equivalence

Lisp symbol	C function
abort	cl_object cl_abort(cl_narg nargs, ...)
break	[Only in Common Lisp]
cell-error-name	[Only in Common Lisp]
error	cl_object cl_error(cl_narg nargs, cl_object continue-format-control, cl_object datum, ...)
compute-restarts	cl_object cl_compute_restarts(cl_narg nargs, ...)
continue	cl_object cl_continue(cl_narg nargs, ...)
error	cl_object cl_error(cl_narg nargs, cl_object datum, ...)
find-restart	cl_object cl_find_restart(cl_narg nargs, cl_object identifier, ...)
handler-case	ECL_HANDLER_CASE macro
invalid-method-error	cl_object cl_invalid_method_error (cl_narg nargs, cl_object method, cl_object format, ...)
invoke-debugger	[Only in Common Lisp]
invoke-restart	cl_object cl_invoke_restart(cl_narg nargs, cl_object restart, ...)
invoke-restart-interactively	cl_object cl_invoke_restart_interactively(cl_object restart)
make-condition	cl_make_condition(cl_narg nargs, cl_object type, ...)
method-combination-error	cl_object cl_method_combination_error(cl_narg nargs, cl_object format, ...)
muffle-warning	cl_object cl_muffle_warning(cl_narg nargs, ...)
restart-name	[Only in Common Lisp]
restart-case	ECL_RESTART_CASE macro
signal	[Only in Common Lisp]
simple-condition-format-control	[Only in Common Lisp]
simple-condition-format-argument	[Only in Common Lisp]
store-value	cl_object cl_store_value(cl_narg nargs, ...)
use-value	cl_object cl_use_value(cl_narg nargs, ...)
warn	[Only in Common Lisp]

2.8 Symbols

There are no implementation-specific limits on the size or content of symbol names. It is however not allowed to write on the strings which have been passed to `#'make-symbol` or returned from `#'symbol-name`.

2.8.1 C Reference

`cl_object ecl_make_keyword (char *name);` [Function]
Find a lisp keyword

Description

Many Lisp functions take keyword arguments. When invoking a function with keyword arguments we need keywords, which are a kind of symbols that live in the `keyword` package. This function does the task of finding or creating those keywords from C strings.

- It is usually safe to store the resulting pointer, because keywords are always referenced by their package and will not be garbage collected (unless of course, you decide to delete it).
- Remember that the case of the string is significant. `ecl_make_keyword("T0")` with return `:T0`, while `ecl_make_keyword("to")` returns a completely different keyword, `:|to|`. In short, you usually want to use uppercase.

Example The following example converts a section of a string to uppercase characters:

```
cl_object start = ecl_make_keyword("START");
cl_object end = ecl_make_keyword("END");
...
sup = cl_string_upcase(4, s, start, ecl_make_fixnum(2),
                      end, ecl_make_fixnum(6));
```

```
cl_object ecl_make_symbol (const char *name, const char          [Function]
                          *package_name);
```

Find a lisp symbol

Description This function finds or create a symbol in the given package. First of all, it tries to find the package named by *package_name*. If it does not exist, an error is signaled. Then, a symbol with the supplied *name* is created and interned in the given package.

2.8.1.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol

boundp
copy-symbol
get

gensym
gentemp
keywordp
make-symbol
makunbound
remprop
set
symbolp
symbol-function
(setf symbol-function)

symbol-name
symbol-package
symbol-plist
(setf symbol-plist)

symbol-value

C function

cl_object cl_boundp(cl_object symbolp)
cl_object cl_copy_symbol(cl_narg nargs, cl_object symbol, ...)
cl_object cl_get(cl_narg nargs, cl_object sym, cl_object indicator, ...)

cl_object cl_gensym(cl_narg nargs, ...)
cl_object cl_gentemp(cl_narg nargs, ...)
cl_object cl_keywordp(cl_object object)
cl_object cl_make_symbol(cl_object name)
cl_object cl_makunbound(cl_object makunbound)
cl_object cl_remprop(cl_object symbol, cl_object indicator)
cl_object cl_set(cl_object symbol, cl_object value)
cl_object cl_symbolp(cl_object object)
cl_object cl_symbol_function(cl_object symbol)
cl_object si_fset(cl_narg nargs, cl_object function_name, cl_object definition, ...)

cl_object cl_symbol_name(cl_object symbol)
cl_object cl_symbol_package(cl_object symbol)
cl_object cl_symbol_plist(cl_object symbol)
cl_object si_set_symbol_plist(cl_object symbol, cl_object plist)

cl_object cl_symbol_value(cl_object symbol)

2.9 Packages

In Table 2.4 we list all packages available in ECL. The nicknames are aliases for a package. Thus, `system:symbol` may be written as `sys:symbol` or `si:symbol`. The module field explains which library provides what package. For instance, the ASDF package is obtained when loading the ASDF library with `(require 'asdf)`.

Name	Nickname	In module	Description
COMMON-LISP	CL	ECL core	Main Common Lisp package.
COMMON-LISP-USER	CL-USER	ECL core	User package.
CLOS	MOP	ECL core	Symbols from the AMOP.
EXT		ECL core	ECL extensions to the language & library.
SYSTEM	SI, SYS	ECL core	Functions and variables internal to the implementation. Never to be used.
FFI		ECL core	Foreign function interface
CMP	C	CMP	The compiler
SB-BSD-SOCKETS		SOCKETS	Sockets library compatible with SBCL's
SB-RT	RT, REGRESSION-TEST	RT	Test units (customized for ECL)
ASDF		ASDF	System definition file with ECL customizations.

Table 2.4: ECL packages

2.9.1 C Reference

2.9.1.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol

export
find-symbol
find-package
find-all-symbols
import
list-all-packages
rename-package

C function

cl-object cl-export(cl_narg nargs, cl-object symbols, ...)
cl-object cl_find_symbol(cl_narg nargs, cl-object string, ...)
cl-object cl_find_package(cl-object name)
cl-object cl_find_all_symbols(cl-object string)
cl-object cl_import(cl_narg nargs, cl-object symbols, ...)
cl-object cl_list_all_packages(void)
cl-object cl_rename_package(cl_narg nargs, cl-object package, cl-object new_name, ...)

shadow	cl-object cl_shadow(cl_narg nargs, cl-object symbols, ...)
shadowing-import	cl-object cl_shadowing_import(cl_narg nargs, cl-object symbols, ...)
delete-package	cl-object cl_delete_package(cl-object package)
make-package	cl-object cl_make_package(cl_narg nargs, cl-object package_name, ...)
unexport	cl-object cl_unexport(cl_narg nargs, cl-object symbols, ...)
unintern	cl-object cl_unintern(cl_narg nargs, cl-object symbol, ...)
unuse-package	cl-object cl_unuse_package(cl_narg nargs, cl-object package, ...)
use-package	cl-object cl_use_package(cl_narg nargs, cl-object package, ...)
intern	cl-object cl_intern(cl_narg nargs, cl-object string, ...)
package-name	cl-object cl_package_name(cl-object package)
package-nicknames	cl-object cl_package_nicknames(cl-object package)
package-shadowing-symbols	cl-object cl_package_shadowing_symbols(cl-object package)
package-use-list	cl-object cl_package_use_list(cl-object package)
package-used-by-list	cl-object cl_package_used_by_list(cl-object package)
packagep	cl-object cl_packagep(cl-object object)
package-error-package	[Only in Common Lisp]

2.10 Numbers

2.10.1 Numeric types

ECL supports all of the Common Lisp numeric tower, which is shown in Table 2.5. The details, however, depend both on the platform on which ECL runs and on the configuration which was chosen when building ECL.

Type	Description
fixnum	Signed integer with a number of bits given by ext:fixnum-bits, fit in a machine word.
bignum	Arbitrary size integers, only limited by amount of memory.
ratio	Arbitrary size rational number, made up of two integers.
short-float	Equivalent to single-float.
single-float	32-bits IEEE floating point number.
double-float	64-bits IEEE floating point number.
long-float	Either equivalent to double-float, or a 96/128 bits IEEE floating point number (long double in C/C++).
rational	An alias for (or integer ratio)
float	An alias for (or single-float double-float short-float long-float)
real	An alias for (or rational float)
complex	Complex number made of two real numbers of the above mentioned types or a <float> _Complex type in C99.

Table 2.5: Numeric types in ECL

In general, the size of a `fixnum` is determined by the word size of a machine, which ranges from 32 to 64 bits. Integers larger than this are implemented using the GNU Multiprecision library (<https://gmplib.org/>). Rationals are implemented using two integers, without caring whether they are `fixnum` or not. Floating point numbers include at least the two IEEE types of 32 and 64 bits respectively.

In machines where it is supported, it is possible to associate the lisp `long-float` with the machine type `long double` whose size ranges from 96 to 128 bits, and which are a bit slower.

In machines where a type `<float>_Complex` is supported numbers of type `(complex float)` are implemented with it, otherwise like ratios all complex numbers are pairs of numbers.

2.10.2 Floating point exceptions

ECL supports two ways of dealing with special floating point values, such as Not a Number (NaN), infinity or denormalized floats, which can occur in floating point computations. Either a condition is signaled or the value is silently used as it is. There are multiple options controlling which behaviour is selected: If ECL is built with the `--without-ieee-fp` configure option, then a condition is signaled for every infinity or NaN encountered. If not, floating point exceptions can be disabled at build time using the `--without-fpe` configure option. Otherwise, if both `--with-ieee-fp` and `--with-fpe` options are on, by default, a condition is signaled for invalid operation, division by zero and floating point overflows. This can be changed at runtime by using `ext:trap-fpe`. If the `ECL_OPT_TRAP_SIGFPE` boot option is false (see Table 1.1), no conditions are signaled by default (Note that in this case, if you enable trapping of floating point exceptions with `ext:trap-fpe`, then you have to install your own signal handler).

`ext:trap-fpe` *condition flag* [Function]

Control the signaling of the floating point exceptions

Synopsis

condition a symbol - one of `last`, `t`, `division-by-zero`, `floating-point-overflow`, `floating-point-underflow`, `floating-point-invalid-operation`, `floating-point-inexact` or an integer.

flag a generalized boolean

Description If *condition* is `last`, *flag* is ignored and the currently enabled floating point exceptions are returned in an implementation depended format (currently an integer). Otherwise, *flag* determines whether the current thread will signal a floating point exception for the conditions passed in *condition*. *condition* can be either a symbol denoting a single condition, `t` for all conditions that are enabled by default or a value obtained from an earlier call to `ext:trap-fpe` with `last`.

See also `ECL_WITH_LISP_FPE`

2.10.3 Random-States

ECL relies internally on a 32-bit Mersenne-Twister random number generator, using a relatively large buffer to precompute about 5000 pseudo-random bytes. This implies also that random states can be printed readably and also read, using the `#$` macro. There is

no provision to create random states from user arrays, though. Random state is printed unreadably by default.

The `#$` macro can be used to initialize the generator with a random seed (an integer), an array of random bytes (mainly used for reading back printed random-state) and another random-state (syntactic sugar for copying the random-state).

2.10.4 Infinity and Not a Number

The ANSI Common-Lisp standard does not specify the behaviour of numeric functions for infinite or not number valued floating point numbers. If ECL is configured to support these special values (see the `--with-ieee-fp` configure option) and floating point exceptions are disabled, numeric functions generally return the same value as the corresponding C function. This means, that the output will be a NaN for a NaN input, and the “mathematically correct” value (which may be NaN, e.g. for $\infty - \infty$) for an infinite real input. For complex floats, however, the return value of a numeric function called with a complex number for which the real or imaginary part is infinite, is undefined¹.

For other functions dealing with numbers, we adopt the following behaviour:

Comparison functions

All numeric comparisons with `=`, `<`, `<=`, `>`, `>=` involving NaN return false. Comparing two NaNs of the same type with `eq1` returns true.

min/max NaN values are ignored, i.e. the maximum/minimum is taken only over the number valued parameters.

Rounding functions

All rounding functions signal an `arithmetic-error` if any of the given parameters are not number valued or infinite.

2.10.5 Dictionary

`ext:{short,single,double,long}-float-{positive,negative}-infinity` [Constant]

Constant positive/negative infinity for the different floating point types.

`ext:nan` [Function]

Returns a double float NaN value. Coerce to other floating point types to get NaN values e.g. for single floats.

`ext:float-infinity-p x` [Function]

`ext:float-nan-p x` [Function]

Predicates to test if `x` is infinite or NaN.

2.10.6 C Reference

¹ The main reason for this is that some numeric functions for C complex numbers return mathematically incorrect values, for example `sinh(i*∞)` returns `i*NaN` instead of the mathematically correct `i*∞`. Keeping this consistent with our own implementation of complex arithmetic that is used when C complex numbers are not available would require too much work. Furthermore, complex arithmetic with infinities is unreliable anyway, since it quickly leads to NaN values (consider `i*∞ = (0+i*1)*(∞+i*0) = NaN+i*∞`; even this simple example is already mathematically incorrect).

2.10.6.1 Number C types

Numeric C types understood by ECL

Type names

<code>cl_fixnum</code>	<code>fixnum</code>		
<code>cl_index</code>	(integer 0 most-positive-fixnum)		
<code>float</code>	short-float, single-float		
<code>double</code>	double-float		
<code>long double (*)</code>	long-float	<code>ECL_LONG_FLOAT</code>	<code>:long-float</code>
<code>float _Complex (**)</code>	(complex single-float)	<code>ECL_COMPLEX_FLOAT</code>	<code>:complex-float</code>
<code>double _Complex (**)</code>	(complex double-float)	<code>ECL_COMPLEX_FLOAT</code>	<code>:complex-float</code>
<code>long-double _Complex (**)</code>	(complex long-float)	<code>ECL_COMPLEX_FLOAT</code>	<code>:complex-float</code>
<code>uint8_t</code>	(unsigned-byte 8)	<code>ecl_uint8_t</code>	
<code>int8_t</code>	(signed-byte 8)	<code>ecl_int8_t</code>	
<code>uint16_t</code>	(unsigned-byte 16)	<code>ecl_uint16_t</code>	<code>:uint16-t</code>
<code>int16_t</code>	(signed-byte 16)	<code>ecl_int16_t</code>	<code>:int16-t</code>
<code>uint32_t</code>	(unsigned-byte 32)	<code>ecl_uint32_t</code>	<code>:uint32-t</code>
<code>int32_t</code>	(signed-byte 32)	<code>ecl_int32_t</code>	<code>:int32-t</code>
<code>uint64_t</code>	(unsigned-byte 64)	<code>ecl_uint64_t</code>	<code>:uint64-t</code>
<code>int64_t</code>	(signed-byte 64)	<code>ecl_int64_t</code>	<code>:int64-t</code>
<code>short</code>	(integer ffi:c-short-min ffi:c-short-max)		
<code>unsigned short</code>	(integer 0 ffi:c-ushort-max)		
<code>int</code>	(integer ffi:c-int-min ffi:c-int-max)		
<code>unsigned int</code>	(integer 0 ffi:c-uint-max)		
<code>long</code>	(integer ffi:c-long-min ffi:c-long-max)		
<code>unsigned long</code>	(integer 0 ffi:c-long-max)		
<code>long long</code>	(integer ffi:c-long-long-min ffi:c-long-long-max)	<code>ecl_long_long_t</code>	<code>:long-long</code>
<code>unsigned long long</code>	(integer 0 ffi:c-ulong-long-max)	<code>ecl_ulong_long_t</code>	<code>:ulong-long</code>

Description The table above shows the relation between C types and the equivalent Common Lisp types. All types are standard C99 types, except for two. First, `cl_fixnum` is the smallest signed integer that can fit a fixnum. Second, `cl_index` is the smallest unsigned integer that fits a fixnum and is typically the unsigned counterpart of `cl_fixnum`.

(*) **DEPRECATED** Previous versions of ECL supported compilers that did not define the long double type. The `ECL_LONG_DOUBLE` macro and `long-double` features indicating whether support for long double was available are removed now.

(**) The `<float> _Complex` types do not exist on all platforms. When they exist the macro `ECL_COMPLEX_FLOAT` will be defined.

Many other types might also not exist on all platforms. This includes not only `long long` and `unsigned long long`, but also some of the C99 integer types. There are two ways to detect which integer types are available in your system:

- Check for the definition of C macros with a similar name, shown in the fifth column above.
- In Lisp code, check for the presence of the associated features, shown in the fourth column above.

2.10.6.2 Number constructors

Creating Lisp types from C numbers

Functions

<code>cl_object ecl_make_fixnum (cl_fixnum n)</code>	[Function]
<code>cl_object ecl_make_integer (cl_fixnum n)</code>	[Function]
<code>cl_object ecl_make_unsigned_integer (cl_index n)</code>	[Function]
<code>cl_object ecl_make_single_float (float n)</code>	[Function]
<code>cl_object ecl_make_double_float (double n)</code>	[Function]
<code>cl_object ecl_make_long_float (long double n)</code>	[Function]
<code>cl_object ecl_make_csfloor (float _Complex n)</code>	[Function]
<code>cl_object ecl_make_cdfloor (double _Complex n)</code>	[Function]
<code>cl_object ecl_make_clfloor (long double _Complex n)</code>	[Function]
<code>cl_object ecl_make_uint8_t (uint8_t n)</code>	[Function]
<code>cl_object ecl_make_int8_t (int8_t n)</code>	[Function]
<code>cl_object ecl_make_uint16_t (uint16_t n)</code>	[Function]
<code>cl_object ecl_make_int16_t (int16_t n)</code>	[Function]
<code>cl_object ecl_make_uint32_t (uint32_t n)</code>	[Function]
<code>cl_object ecl_make_int32_t (int32_t n)</code>	[Function]
<code>cl_object ecl_make_uint64_t (uint64_t n)</code>	[Function]
<code>cl_object ecl_make_int64_t (int64_t n)</code>	[Function]
<code>cl_object ecl_make_short_t (short n)</code>	[Function]
<code>cl_object ecl_make_ushort_t (unsigned short n)</code>	[Function]
<code>cl_object ecl_make_int (int n)</code>	[Function]
<code>cl_object ecl_make_uint (unsigned int n)</code>	[Function]
<code>cl_object ecl_make_long (long n)</code>	[Function]
<code>cl_object ecl_make_ulong (unsigned long n)</code>	[Function]
<code>cl_object ecl_make_long_long (long long n)</code>	[Function]
<code>cl_object ecl_make_ulong_long (unsigned long long n)</code>	[Function]
<code>cl_object ecl_make_ratio (cl_object numerator, cl_object denominator)</code>	[Function]
<code>cl_object ecl_make_complex (cl_object real, cl_object imag)</code>	[Function]

Description These functions create a Lisp object from the corresponding C number. If the number is an integer type, the result will always be an integer, which may be a bignum. If on the other hand the C number is a float, double or long double, the result will be a float.

There is some redundancy in the list of functions that convert from `cl_fixnum` and `cl_index` to lisp. On the one hand, `ecl_make_fixnum` always creates a

fixnum, dropping bits if necessary. On the other hand, `ecl_make_integer` and `ecl_make_unsigned_integer` faithfully convert to a Lisp integer, which may be a bignum.

Note also that some of the constructors do not use C numbers. This is the case of `ecl_make_ratio` and `ecl_make_complex`, because they are composite Lisp types. When c99 complex float support is built in `ecl_make_complex` will use C number for float types.

These functions or macros signal no errors.

2.10.6.3 Number accessors

Unchecked conversion from Lisp types to C numbers

Functions

<code>cl_fixnum ecl_fixnum (cl_object n)</code>	[Function]
<code>float ecl_single_float (cl_object n)</code>	[Function]
<code>double ecl_double_float (cl_object n)</code>	[Function]
<code>long double ecl_long_float (cl_object n)</code>	[Function]
<code>float _Complex ecl_csfloat (cl_object n)</code>	[Function]
<code>double _Complex ecl_cdfloat (cl_object n)</code>	[Function]
<code>long double _Complex ecl_clfloat (cl_object n)</code>	[Function]

Description These functions and macros extract a C number from a Lisp object. They do not check the type of the Lisp object as they typically just access directly the value from a C structure.

2.10.6.4 Number coercion

Checked conversion from Lisp types to C numbers

Functions

<code>cl_fixnum ecl_to_fixnum (cl_object n);</code>	[Function]
<code>cl_index ecl_to_unsigned_integer (cl_object n);</code>	[Function]
<code>float ecl_to_float (cl_object n);</code>	[Function]
<code>double ecl_to_double (cl_object n);</code>	[Function]
<code>long double ecl_to_long_double (cl_object n);</code>	[Function]
<code>float _Complex ecl_to_csfloat (cl_object n);</code>	[Function]
<code>double _Complex ecl_to_cdfloat (cl_object n);</code>	[Function]
<code>long double _Complex ecl_to_clfloat (cl_object n);</code>	[Function]
<code>uint8_t ecl_to_uint8_t (cl_object n);</code>	[Function]
<code>int8_t ecl_to_int8_t (cl_object n);</code>	[Function]
<code>uint16_t ecl_to_uint16_t (cl_object n);</code>	[Function]
<code>int16_t ecl_to_int16_t (cl_object n);</code>	[Function]
<code>uint32_t ecl_to_uint32_t (cl_object n);</code>	[Function]
<code>int32_t ecl_to_int32_t (cl_object n);</code>	[Function]
<code>uint64_t ecl_to_uint64_t (cl_object n);</code>	[Function]
<code>int64_t ecl_to_int64_t (cl_object n);</code>	[Function]
<code>short ecl_to_short (cl_object n);</code>	[Function]

<code>unsigned short ecl_to_ushort (cl_object n);</code>	[Function]
<code>int ecl_to_int (cl_object n);</code>	[Function]
<code>unsigned int ecl_to_uint (cl_object n);</code>	[Function]
<code>long ecl_to_long (cl_object n);</code>	[Function]
<code>unsigned long ecl_to_ulong (cl_object n);</code>	[Function]
<code>long long ecl_to_long_long (cl_object n);</code>	[Function]
<code>unsigned long long ecl_to_ulong_long (cl_object n);</code>	[Function]

Description These functions and macros convert a Lisp object to the corresponding C number type. The conversion is done through a coercion process which may signal an error if the argument does not fit the expected type.

2.10.6.5 ANSI dictionary

Common Lisp and C equivalence

Lisp symbol	C function
<code>=</code>	<code>cl_object cl_E(cl_narg nargs, ...)</code>
<code>/=</code>	<code>cl_object cl_NE(cl_narg nargs, ...)</code>
<code><</code>	<code>cl_object cl_L(cl_narg nargs, ...)</code>
<code>></code>	<code>cl_object cl_G(cl_narg nargs, ...)</code>
<code><=</code>	<code>cl_object cl_LE(cl_narg nargs, ...)</code>
<code>>=</code>	<code>cl_object cl_GE(cl_narg nargs, ...)</code>
<code>max</code>	<code>cl_object cl_max(cl_narg nargs, ...)</code>
<code>min</code>	<code>cl_object cl_min(cl_narg nargs, ...)</code>
<code>minusp</code>	<code>cl_object cl_minusp(cl_object real)</code>
<code>plusp</code>	<code>cl_object cl_plusp(cl_object real)</code>
<code>zerop</code>	<code>cl_object cl_zerop(cl_object number)</code>
<code>floor</code>	<code>cl_object cl_floor(cl_narg nargs, cl_object number, ...)</code>
<code>ffloor</code>	<code>cl_object cl_ffloor(cl_narg nargs, cl_object number, ...)</code>
<code>ceiling</code>	<code>cl_object cl_ceiling(cl_narg nargs, cl_object number, ...)</code>
<code>fceiling</code>	<code>cl_object cl_fceiling(cl_narg nargs, cl_object number, ...)</code>
<code>truncate</code>	<code>cl_object cl_truncate(cl_narg nargs, cl_object number, ...)</code>
<code>ftruncate</code>	<code>cl_object cl_ftruncate(cl_narg nargs, cl_object number, ...)</code>
<code>round</code>	<code>cl_object cl_round(cl_narg nargs, cl_object number, ...)</code>
<code>fround</code>	<code>cl_object cl_fround(cl_narg nargs, cl_object number, ...)</code>
<code>sin</code>	<code>cl_object cl_sin(cl_object radians)</code>
<code>cos</code>	<code>cl_object cl_cos(cl_object radians)</code>
<code>tan</code>	<code>cl_object cl_tan(cl_object radians)</code>
<code>asin</code>	<code>cl_object cl_asin(cl_object number)</code>
<code>acos</code>	<code>cl_object cl_acos(cl_object number)</code>
<code>atan</code>	<code>cl_object cl_atan(cl_narg nargs, cl_object number1, ...)</code>
<code>sinh</code>	<code>cl_object cl_sinh(cl_object number)</code>
<code>cosh</code>	<code>cl_object cl_cosh(cl_object number)</code>
<code>tanh</code>	<code>cl_object cl_tanh(cl_object number)</code>
<code>asinh</code>	<code>cl_object cl_asinh(cl_object number)</code>
<code>acosh</code>	<code>cl_object cl_acosh(cl_object number)</code>
<code>atanh</code>	<code>cl_object cl_atanh(cl_object number)</code>

<code>*</code>	<code>cl_object cl_X(cl_narg nargs, ...)</code>
<code>+</code>	<code>cl_object cl_P(cl_narg nargs, ...)</code>
<code>-</code>	<code>cl_object cl_M(cl_narg nargs, cl_object number, ...)</code>
<code>/</code>	<code>cl_object cl_N(cl_narg nargs, cl_object number, ...)</code>
<code>1+</code>	<code>cl_object cl_1P(cl_object number)</code>
<code>1-</code>	<code>cl_object cl_1M(cl_object number)</code>
<code>abs</code>	<code>cl_object cl_abs(cl_object number)</code>
<code>evenp</code>	<code>cl_object cl_evenp(cl_object integer)</code>
<code>oddp</code>	<code>cl_object cl_oddp(cl_object integer)</code>
<code>exp</code>	<code>cl_object cl_exp(cl_object number)</code>
<code>expt</code>	<code>cl_object cl_expt(cl_object base, cl_object power)</code>
<code>gcd</code>	<code>cl_object cl_gcd(cl_narg nargs, ...)</code>
<code>lcm</code>	<code>cl_object cl_lcm(cl_narg nargs, ...)</code>
<code>log</code>	<code>cl_object cl_log(cl_narg nargs, cl_object number, ...)</code>
<code>mod</code>	<code>cl_object cl_mod(cl_object number, cl_object divisor)</code>
<code>rem</code>	<code>cl_object cl_rem(cl_object number, cl_object divisor)</code>
<code>signum</code>	<code>cl_object cl_signum(cl_object number)</code>
<code>sqrt</code>	<code>cl_object cl_sqrt(cl_object number)</code>
<code>isqrt</code>	<code>cl_object cl_isqrt(cl_object natural)</code>
<code>make-random-state</code>	<code>cl_object cl_make_random_state(cl_narg nargs, ...)</code>
<code>random</code>	<code>cl_object cl_random(cl_narg nargs, cl_object limit, ...)</code>
<code>random-state-p</code>	<code>cl_object cl_random_state_p(cl_object object)</code>
<code>numberp</code>	<code>cl_object cl_numberp(cl_object object)</code>
<code>cis</code>	<code>cl_object cl_cis(cl_object radians)</code>
<code>complex</code>	<code>cl_object cl_complex(cl_narg nargs, cl_object realpart, ...)</code>
<code>complexp</code>	<code>cl_object cl_complexp(cl_object object)</code>
<code>conjugate</code>	<code>cl_object cl_conjugate(cl_object number)</code>
<code>phase</code>	<code>cl_object cl_phase(cl_object number)</code>
<code>realpart</code>	<code>cl_object cl_realpart(cl_object number)</code>
<code>imagpart</code>	<code>cl_object cl_imagpart(cl_object number)</code>
<code>upgraded-complex-part-type</code>	<code>cl_object cl_upgraded_complex_part_type(cl_narg nargs, cl_object typespec, ...)</code>
<code>realp</code>	<code>cl_object cl_realp(cl_object object)</code>
<code>numerator</code>	<code>cl_object cl_numerator(cl_object rational)</code>
<code>denominator</code>	<code>cl_object cl_denominator(cl_object rational)</code>
<code>rational</code>	<code>cl_object cl_rational(cl_object number)</code>
<code>rationalize</code>	<code>cl_object cl_rationalize(cl_object number)</code>
<code>rationalp</code>	<code>cl_object cl_rationalp(cl_object object)</code>
<code>ash</code>	<code>cl_object cl_ash(cl_object integer, cl_object count)</code>
<code>integer-length</code>	<code>cl_object cl_integer_length(cl_object integer)</code>
<code>integerp</code>	<code>cl_object cl_integerp(cl_object object)</code>
<code>parse-integer</code>	<code>cl_object cl_parse_integer(cl_narg nargs, cl_object string, ...)</code>
<code>boole</code>	<code>cl_object cl_boole(cl_object op, cl_object integer1, cl_object integer2)</code>
<code>logand</code>	<code>cl_object cl_logand(cl_narg nargs, ...)</code>
<code>logandc1</code>	<code>cl_object cl_logandc1(cl_object integer1, cl_object integer2)</code>

logandc2	cl-object cl_logandc2(cl-object integer1, cl-object integer2)
logeqv	cl-object cl_logeqv(cl_narg nargs, ...)
logior	cl-object cl_logior(cl_narg nargs, ...)
lognand	cl-object cl_lognand(cl-object integer1, cl-object integer2)
lognor	cl-object cl_lognor(cl-object integer1, cl-object integer2)
lognot	cl-object cl_lognot(cl-object integer)
logorc1	cl-object cl_logorc1(cl-object integer1, cl-object integer2)
logorc2	cl-object cl_logorc2(cl-object integer1, cl-object integer2)
logxor	cl-object cl_logxor(cl_narg nargs, ...)
logbitp	cl-object cl_logbitp(cl-object index, cl-object integer)
logcount	cl-object cl_logcount(cl-object integer)
logtest	cl-object cl_logtest(cl-object integer1, cl-object integer2)
byte	cl-object cl_byte(cl-object size, cl-object position)
bytes-size	cl-object cl_byte_size(cl-object bytespec)
byte-position	cl-object cl_byte_position(cl-object bytespec)
deposit-field	cl-object cl_deposit_field(cl-object newbyte, cl-object byte-spec, cl-object integer)
dpb	cl-object cl_dpb(cl-object newbyte, cl-object bytespec, cl-object integer)
ldb	cl-object cl_ldb(cl-object bytespec, cl-object integer)
ldb-test	cl-object cl_ldb_test(cl-object bytespec, cl-object integer)
mask-field	cl-object cl_mask_field(cl-object bytespec, cl-object integer)
decode-float	cl-object cl_decode_float(cl-object float)
scale-float	cl-object cl_scale_float(cl-object float, cl-object integer)
float-radix	cl-object cl_float_radix(cl-object float)
float-sign	cl-object cl_float_sign(cl_narg nargs, cl-object float1, ...)
float-digits	cl-object cl_float_digits(cl-object float)
float-precision	cl-object cl_float_precision(cl-object float)
integer-decode-float	cl-object cl_integer_decode_float(cl-object float)
float	cl-object cl_float(cl_narg nargs, cl-object number, ...)
floatp	cl-object cl_floatp(cl-object object)
arithmetic-error-operands	[Only in Common Lisp]
arithmetic-error-operation	[Only in Common Lisp]

2.11 Characters

ECL is fully ANSI Common-Lisp compliant in all aspects of the character data type, with the following peculiarities.

2.11.1 Unicode vs. POSIX locale

There are two ways of building ECL: with C or with Unicode character codes. These build modes are accessed using the `--disable-unicode` and `--enable-unicode` configuration options, the last one being the default.

When using C characters we are actually relying on the `char` type of the C language, using the C library functions for tasks such as character conversions, comparison, etc. In this case characters are typically 8 bit wide and the character order and collation are determined by the current POSIX or C locale. This is not very accurate, leaves out many languages and

character encodings but it is sufficient for small applications that do not need multilingual support.

When no option is specified ECL builds with support for a larger character set, the Unicode 6.0 standard. This uses 24 bit large character codes, also known as *codepoints*, with a large database of character properties which include their nature (alphanumeric, numeric, etc), their case, their collation properties, whether they are standalone or composing characters, etc.

2.11.1.1 Character types

If ECL is compiled without Unicode support, all characters are implemented using 8-bit codes and the type `extended-char` is empty. If compiled with Unicode support, characters are implemented using 24 bits and the `extended-char` type covers characters above code 255.

Type	With Unicode	Without Unicode
standard-char	<code>#\Newline</code> ,32-126	<code>#\Newline</code> ,32-126
base-char	0-255	0-255
extended-char	-	256-16777215

2.11.1.2 Character names

All characters have a name. For non-printing characters between 0 and 32, and for 127 we use the ordinary ASCII names. Characters above 127 are printed and read using hexadecimal Unicode notation, with a U followed by 24 bit hexadecimal number, as in `U0126`.

Character	Code
<code>#\Null</code>	0
<code>#\Ack</code>	1
<code>#\Bell</code>	7
<code>#\Backspace</code>	8
<code>#\Tab</code>	9
<code>#\Newline</code>	10
<code>#\Linefeed</code>	10
<code>#\Page</code>	12
<code>#\Esc</code>	27
<code>#\Escape</code>	27
<code>#\Space</code>	32
<code>#\Rubout</code>	127
<code>#\U0080</code>	128

Table 2.6

Note that `#\Linefeed` is synonymous with `#\Newline` and thus is a member of `standard-char`.

2.11.2 `#\Newline` characters

Internally, ECL represents the `#\Newline` character by a single code. However, when using external formats, ECL may parse character pairs as a single `#\Newline`, and vice versa, use

multiple characters to represent a single `#\Newline`, see Section 2.19.1.3 [Streams - External formats], page 70.

2.11.3 C Reference

2.11.3.1 C types

C character types

Type names

`ecl_character` `character`
`ecl_base_char` `base-char`

Description ECL defines two C types to hold its characters: `ecl_base_char` and `ecl_character`.

- When ECL is built without Unicode, they both coincide and typically match `unsigned char`, to cover the 256 codes that are needed.
- When ECL is built with Unicode, the two types are no longer equivalent, with `ecl_character` being larger.

For your code to be portable and future proof, use both types to really express what you intend to do.

2.11.3.2 Constructors

Creating and extracting characters from Lisp objects

Functions

`cl_object` `ECL_CODE_CHAR` (*ecl_character* *code*); [Macro]
`ecl_character` `ECL_CHAR_CODE` (*cl_object* *o*); [Macro]
`ecl_character` `ecl_char_code` (*cl_object* *o*); [Function]
`ecl_base_char` `ecl_base_char_code` (*cl_object* *o*); [Function]

Description These functions and macros convert back and forth from C character types to Lisp. The macros `ECL_CHAR_CODE` and `ECL_CODE_CHAR` perform this coercion without checking the arguments. The functions `ecl_char_code` and `ecl_base_char_code`, on the other hand, verify that the argument has the right type and signal an error otherwise.

2.11.3.3 Predicates

C predicates for Lisp characters

Functions

`bool` `ecl_base_char_p` (*ecl_character* *c*); [Function]
`bool` `ecl_alpha_char_p` (*ecl_character* *c*); [Function]
`bool` `ecl_alphanumericp` (*ecl_character* *c*); [Function]
`bool` `ecl_graphic_char_p` (*ecl_character* *c*); [Function]
`bool` `ecl_digitp` (*ecl_character* *c*); [Function]
`bool` `ecl_standard_char_p` (*ecl_character* *c*); [Function]

Description These functions are equivalent to their Lisp equivalents but return C booleans.

2.11.3.4 Character case

C functions related to the character case

Functions

```
bool ecl_upper_case_p (ecl_character c); [Function]
bool ecl_lower_case_p (ecl_character c); [Function]
bool ecl_both_case_p (ecl_character c); [Function]
ecl_character ecl_char_downcase (ecl_character c); [Function]
ecl_character ecl_char_upcase (ecl_character c); [Function]
```

Description These functions check or change the case of a character. Note that in a Unicode context, the output of these functions might not be accurate (for instance when the uppercase character has two or more codepoints).

2.11.3.5 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function
char=	cl_object cl_charE(cl_narg nargs, ...)
char/=	cl_object cl_charNE(cl_narg nargs, ...)
char<	cl_object cl_charL(cl_narg nargs, ...)
char>	cl_object cl_charG(cl_narg nargs, ...)
char<=	cl_object cl_charLE(cl_narg nargs, ...)
char>=	cl_object cl_charGE(cl_narg nargs, ...)
char-equal	cl_object cl_char_equal(cl_narg nargs, ...)
char-not-equal	cl_object cl_char_not_equal(cl_narg nargs, ...)
char-lessp	cl_object cl_char_lessp(cl_narg nargs, ...)
char-greaterp	cl_object cl_char_greaterp(cl_narg nargs, ...)
char-not-greaterp	cl_object cl_char_not_greaterp(cl_narg nargs, ...)
char-not-lessp	cl_object cl_char_not_lessp(cl_narg nargs, ...)
character	cl_object cl_character(cl_object char_designator)
characterp	cl_object cl_characterp(cl_object object)
alpha-char-p	cl_object cl_alpha_char_p(cl_object character)
alphanumericp	cl_object cl_alphanumericp(cl_object character)
digit-char	cl_object cl_digit_char(cl_narg nargs, cl_object character, ...)
digit-char-p	cl_object cl_digit_char_p(cl_narg nargs, cl_object character, ...)
graphic-char-p	cl_object cl_graphic_char_p(cl_object character)
standard-char-p	cl_object cl_standard_char_p(cl_object character)
char_upcase	cl_object cl_char_upcase(cl_object character)
char-downcase	cl_object cl_char_downcase(cl_object character)
upper-case-p	cl_object cl_upper_case_p(cl_object character)
lower-case-p	cl_object cl_lower_case_p(cl_object character)
both-case-p	cl_object cl_both_case_p(cl_object character)
char-code	cl_object cl_char_code(cl_object character)
char-int	cl_object cl_char_int(cl_object character)
code-char	cl_object cl_code_char(cl_object code)
char-name	cl_object cl_char_name(cl_object character)

name-char	cl-object cl_name_char(cl-object name)
char-code-limit	ECL_CHAR_CODE_LIMIT

2.12 Conses

2.12.1 C Reference

2.12.1.1 Accessors

Accessing the elements of conses

Functions

cl-object ECL_CONS_CAR (<i>cl-object o</i>)	[Function]
cl-object ECL_CONS_CDR (<i>cl-object o</i>)	[Function]
cl-object ECL_RPLACA (<i>cl-object o, cl-object v</i>)	[Function]
cl-object ECL_RPLACD (<i>cl-object o, cl-object v</i>)	[Function]
cl-object _ecl_car (<i>cl-object o</i>)	[Function]
cl-object _ecl_cdr (<i>cl-object o</i>)	[Function]
cl-object _ecl_caar (<i>cl-object o</i>)	[Function]
cl-object _ecl_cadr (<i>cl-object o</i>)	[Function]
...	

Description These functions access the elements of objects of type cons (ECL_CONS_CAR, ECL_CONS_CDR, ECL_RPLACA and ECL_RPLACD) or type list (_ecl_car, _ecl_cdr, _ecl_caar, ...). They don't check the type of their arguments.

2.12.1.2 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol

cons
consp
atom
rplaca
rplacd
car
cdr
caar
cdar
cadr
cddr
caaar
cdaar
cadar
cddar
caadr
cdadr

C function

cl-object cl_cons(cl-object car, cl-object cdr)
cl-object cl_consp(cl-object object)
cl-object cl_atom(cl-object object)
cl-object cl_rplaca(cl-object cons, cl-object car)
cl-object cl_rplacd(cl-object cons, cl-object cdr)
cl-object cl_car(cl-object cons)
cl-object cl_cdr(cl-object cons)
cl-object cl_caar(cl-object cons)
cl-object cl_cdar(cl-object cons)
cl-object cl_cadr(cl-object cons)
cl-object cl_cddr(cl-object cons)
cl-object cl_caaar(cl-object cons)
cl-object cl_cdaar(cl-object cons)
cl-object cl_cadar(cl-object cons)
cl-object cl_cddar(cl-object cons)
cl-object cl_caadr(cl-object cons)
cl-object cl_cdadr(cl-object cons)

caddr	cl_object cl_caddr(cl_object cons)
cdddr	cl_object cl_cdddr(cl_object cons)
caaaaar	cl_object cl_caaaaar(cl_object cons)
cdaaar	cl_object cl_cdaaar(cl_object cons)
cadaar	cl_object cl_cadaar(cl_object cons)
cddaar	cl_object cl_cddaar(cl_object cons)
caadar	cl_object cl_caadar(cl_object cons)
cdadar	cl_object cl_cdadar(cl_object cons)
caddar	cl_object cl_caddar(cl_object cons)
cdddar	cl_object cl_cdddar(cl_object cons)
caaaadr	cl_object cl_caaadr(cl_object cons)
cdaadr	cl_object cl_cdaadr(cl_object cons)
cadadr	cl_object cl_cadadr(cl_object cons)
cddadr	cl_object cl_cddadr(cl_object cons)
caaddr	cl_object cl_caaddr(cl_object cons)
cdaddr	cl_object cl_cdaddr(cl_object cons)
cadddr	cl_object cl_cadddr(cl_object cons)
cdddr	cl_object cl_cdddr(cl_object cons)
copy-tree	cl_object cl_copy_tree(cl_object tree)
sublis	cl_object cl_sublis(cl_narg nargs, cl_object alist, cl_object tree, ...)
nsublis	cl_object cl_sublis(cl_narg nargs, cl_object alist, cl_object tree, ...)
subst	cl_object cl_subst(cl_narg nargs, cl_object new, cl_object old, cl_object tree, ...)
subst-if	cl_object cl_subst_if(cl_narg nargs, cl_object new, cl_object predicate, cl_object tree, ...)
subst-if-not	cl_object cl_subst_if_not(cl_narg nargs, cl_object new, cl_object predicate, cl_object tree, ...)
nsubst	cl_object cl_nsubst(cl_narg nargs, cl_object new, cl_object old, cl_object tree, ...)
nsubst-if	cl_object cl_nsubst_if(cl_narg nargs, cl_object new, cl_object predicate, cl_object tree, ...)
nsubst-if-not	cl_object cl_nsubst_if_not(cl_narg nargs, cl_object new, cl_object predicate, cl_object tree, ...)
tree-equal	cl_object cl_tree_equal(cl_narg nargs, cl_object tree1, cl_object tree2, ...)
copy-list	cl_object cl_copy_list(cl_object list)
list	cl_object cl_list(cl_narg nargs, ...)
list*	cl_object cl_listX(cl_narg nargs, ...)
list-length	cl_object cl_list_length(cl_object list)
listp	cl_object cl_listp(cl_object object)
make-list	cl_object cl_make_list(cl_narg nargs, cl_object size, ...)
first	cl_object cl_first(cl_object list)
second	cl_object cl_second(cl_object list)
third	cl_object cl_third(cl_object list)

fourth	cl-object cl-fourth(cl-object list)
fifth	cl-object cl-fifth(cl-object list)
sixth	cl-object cl-sixth(cl-object list)
seventh	cl-object cl-seventh(cl-object list)
eighth	cl-object cl-eighth(cl-object list)
ninth	cl-object cl-ninth(cl-object list)
tenth	cl-object cl-tenth(cl-object list)
nth	cl-object cl_nth(cl-object n, cl-object list)
endp	cl-object cl-endp(cl-object list)
null	cl-object cl_null(cl-object object)
nconc	cl-object cl_nconc(cl_narg narg, ...)
append	cl-object cl_append(cl_narg narg, ...)
revappend	cl-object cl_revappend(cl-object list, cl-object tail)
nreconc	cl-object cl_nreconc(cl-object list, cl-object tail)
butlast	cl-object cl_butlast(cl_narg narg, cl-object list, ...)
nbutlast	cl-object cl_nbutlast(cl_narg narg, cl-object list, ...)
last	cl-object cl_last(cl_narg narg, cl-object list, ...)
ldiff	cl-object cl_ldiff(cl-object list, cl-object object)
tailp	cl-object cl_tailp(cl-object object, cl-object list)
nthcdr	cl-object cl_nthcdr(cl-object n, cl-object list)
rest	cl-object cl_rest(cl-object list)
member	cl-object cl_member(cl_narg narg, cl-object member, cl-object list,)
member-if	cl-object cl_member_if(cl_narg narg, cl-object predicate, cl-object list,)
member-if-not	cl-object cl_member_if_not(cl_narg narg, cl-object predicate, cl-object list,)
mapc	cl-object cl_mapc(cl_narg narg, cl-object function, ...)
mapcar	cl-object cl_mapcar(cl_narg narg, cl-object function, ...)
mapcan	cl-object cl_mapcan(cl_narg narg, cl-object function, ...)
mapl	cl-object cl_mapl(cl_narg narg, cl-object function, ...)
maplist	cl-object cl_maplist(cl_narg narg, cl-object function, ...)
mapcon	cl-object cl_mapcon(cl_narg narg, cl-object function, ...)
acons	cl-object cl_acons(cl-object key, cl-object datum, cl-object alist)
assoc	cl-object cl_assoc(cl_narg narg, cl-object item, cl-object alist, ...)
assoc-if	cl-object cl_assoc_if(cl_narg narg, cl-object predicate, cl-object alist, ...)
assoc-if-not	cl-object cl_assoc_if_not(cl_narg narg, cl-object predicate, cl-object alist, ...)
copy-alist	cl-object cl_copy_alist(cl-object alist)
pairlis	cl-object cl_pairlis(cl_narg narg, cl-object keys, cl-object data, ...)
rassoc	cl-object cl_rassoc(cl_narg narg, cl-object item, cl-object al- ist, ...)

rassoc-if	cl-object cl_rassoc_if(cl_narg nargs, cl-object predicate, cl-object alist, ...)
rassoc-if-not	cl-object cl_rassoc_if_not(cl_narg nargs, cl-object predicate, cl-object alist, ...)
get-properties	cl-object cl_get_properties(cl-object plist, cl-object indicator_list)
getf	cl-object cl_getf(cl_narg nargs, cl-object plist, cl-object indicator, ...)
intersection	cl-object cl_intersection(cl_narg nargs, cl-object list1, cl-object list2, ...)
nintersection	cl-object cl_nintersection(cl_narg nargs, cl-object list1, cl-object list2, ...)
adjoin	cl-object cl_adjoin(cl_narg nargs, cl-object item, cl-object list, ...)
set-difference	cl-object cl_set_difference(cl_narg nargs, cl-object list1, cl-object list2, ...)
nset-difference	cl-object cl_nset_difference(cl_narg nargs, cl-object list1, cl-object list2, ...)
set-exclusive-or	cl-object cl_set_exclusive_or(cl_narg nargs, cl-object list1, cl-object list2, ...)
nset-exclusive-or	cl-object cl_nset_exclusive_or(cl_narg nargs, cl-object list1, cl-object list2, ...)
subsetp	cl-object cl_subsetp(cl_narg nargs, cl-object list1, cl-object list2, ...)
union	cl-object cl_union(cl_narg nargs, cl-object list1, cl-object list2, ...)
nunion	cl-object cl_nunion(cl_narg nargs, cl-object list1, cl-object list2, ...)

2.13 Arrays

2.13.1 Array limits

ECL arrays can have up to 64 dimensions. Common-Lisp constants related to arrays have the following values in ECL.

Constant	Value
array-rank-limit	64
array-dimension-limit	most-positive-fixnum
array-total-size-limit	array-dimension-limit

2.13.2 Specializations

When the elements of an array are declared to have some precise type, such as a small or large integer, a character or a floating point number, ECL has means to store those elements in a more compact form, known as a *specialized array*. The list of types for which ECL

specializes arrays is platform dependent, but is summarized in the following table, together with the C type which is used internally and the expected size.

Specialized type	Element C type	Size
bit	-	1 bit
character	unsigned char or uint32_t	Depends on character range
base-char	unsigned char	
fixnum	cl_fixnum	Machine word (32 or 64 bits)
ext:cl-index	cl_index	Machine word (32 or 64 bits)
(signed-byte 8)	int8_t	8 bits
(unsigned-byte 8)	uint8_t	8 bits
(signed-byte 16)	int16_t	16 bits
(unsigned-byte 16)	uint16_t	16 bits
(signed-byte 32)	int32_t	32 bits
(unsigned-byte 32)	uint32_t	32 bits
(signed-byte 64)	int64_t	64 bits
(unsigned-byte 64)	uint64_t	64 bits
single-float or short-float	float	32-bits IEEE float
double-float	double	64-bits IEEE float
long-float	long double	Between 96 and 128 bits
(complex single-float)	float _Complex	64 bits
(complex double-float)	double _Complex	128 bits
(complex long-float)	long double _Complex	Between 192 and 256 bits
t	cl_object	Size of a pointer.

Let us remark that some of these specialized types might not exist in your platform. This is detected using conditional reading and features (See Section 2.10 [Numbers], page 39).

2.13.3 C Reference

2.13.3.1 Types and constants

C types, limits and enumerations

Constants and types

ECL_ARRAY_RANK_LIMIT	[Constant]
ECL_ARRAY_DIMENSION_LIMIT	[Constant]
ECL_ARRAY_TOTAL_LIMIT	[Constant]
cl_elttype { <i>ecl_aet_object</i> , ...}	[enum]

Lisp or C type	Enumeration value	Lisp or C type	Enumeration value
t	<i>ecl_aet_object</i>	(unsigned-byte 1)	<i>ecl_aet_bit</i>
cl_fixnum	<i>ecl_aet_fix</i>	cl_index	<i>ecl_aet_index</i>
(unsigned-byte 8)	<i>ecl_aet_b8</i>	(signed-byte 8)	<i>ecl_aet_i8</i>
(unsigned-byte 16)	<i>ecl_aet_b16</i>	(signed-byte 16)	<i>ecl_aet_i16</i>

(unsigned-byte 32)	<code>ecl_aet_b32</code>	(signed-byte 32)	<code>ecl_aet_i32</code>
(unsigned-byte 64)	<code>ecl_aet_b64</code>	(signed-byte 64)	<code>ecl_aet_i64</code>
<code>ecl_character</code>	<code>ecl_aet_ch</code>	<code>ecl_base_char</code>	<code>ecl_aet_bc</code>
single-float	<code>ecl_aet_sf</code>	double-float	<code>ecl_aet_df</code>
long-float	<code>ecl_aet_lf</code>	(complex long-float)	<code>ecl_aet_clf</code>
(complex single-float)	<code>ecl_aet_csf</code>	(complex double-float)	<code>ecl_aet_cdf</code>

Description This list contains the constants that limit the rank of an array (`ECL_ARRAY_RANK_LIMIT`), the maximum size of each dimension (`ECL_ARRAY_DIMENSION_LIMIT`) and the maximum number of elements in an array (`ECL_ARRAY_TOTAL_LIMIT`).

ECL uses also internally a set of constants to describe the different specialized arrays. The constants form up the enumeration type `cl_elttype`. They are listed in the table above, which associates enumeration values with the corresponding Common Lisp element type.

2.13.3.2 `ecl_aet_to_symbol`, `ecl_symbol_to_aet`

To and from element types

Functions

`cl_object ecl_aet_to_symbol (cl_elttype param)` [Function]

`cl_elttype ecl_symbol_to_aet (cl_object type)` [Function]

Description `ecl_aet_to_symbol` returns the Lisp type associated to the elements of that specialized array class. `ecl_symbol_to_aet` does the converse, computing the C constant that is associated to a Lisp element type.

The functions may signal an error if any of the arguments is an invalid C or Lisp type.

2.13.3.3 Constructors

Creating array and vectors

Functions

`cl_object ecl_alloc_simple_vector (cl_elttype element_type, cl_index length);` [Function]

`cl_object si_make_vector (cl_object element_type, cl_object length, cl_object adjustablep, cl_object fill_pointerp, cl_object displaced_to, cl_object displacement);` [Function]

`cl_object si_make_array (cl_object element_type, cl_object dimensions, cl_object adjustablep, cl_object fill_pointerp, cl_object displaced_to, cl_object displacement);` [Function]

Description The function `ecl_alloc_simple_vector` is the simplest constructor, creating a simple vector (i.e. non-adjustable and without a fill pointer), of the given size, preallocating the memory for the array data. The first argument, *element_type*, is a C constant that represents a valid array element type (See `cl_elttype`).

The function `si_make_vector` does the same job but allows creating an array with fill pointer, which is adjustable or displaced to another array.

- `element_type` is now a Common Lisp type descriptor, which is a symbol or list denoting a valid element type
- `dimension` is a non-negative fixnum with the vector size.
- `fill_pointerp` is either `ECL_NIL` or a non-negative fixnum denoting the fill pointer value.
- `displaced_to` is either `ECL_NIL` or a valid array to which the new array is displaced.
- `displacement` is either `ECL_NIL` or a non-negative value with the array displacement.

Finally, the function `si_make_array` does a similar job to `si_make_vector` but its second argument, *dimension*, can be a list of dimensions, to create a multidimensional array.

Examples Create one-dimensional `base-string` with room for 11 characters:

```
cl_object s = ecl_alloc_simple_vector(ecl_aet_bc, 11);
```

Create a one-dimensional array with a fill pointer

```
cl_object type = ecl_make_symbol("BYTE8","EXT");
cl_object a = si_make_vector(type, ecl_make_fixnum(16), ECL_NIL, /* adjustable */
                             ecl_make_fixnum(0) /* fill-pointer */,
                             ECL_NIL /* displaced_to */,
                             ECL_NIL /* displacement */);
```

An alternative formulation

```
cl_object type = ecl_make_symbol("BYTE8","EXT");
cl_object a = si_make_array(type, ecl_make_fixnum(16), ECL_NIL, /* adjustable */
                             ecl_make_fixnum(0) /* fill-pointer */,
                             ECL_NIL /* displaced_to */,
                             ECL_NIL /* displacement */);
```

Create a 2-by-3 two-dimensional array, specialized for an integer type:

```
cl_object dims = cl_list(2, ecl_make_fixnum(2), ecl_make_fixnum(3));
cl_object type = ecl_make_symbol("BYTE8","EXT");
cl_object a = si_make_array(dims, type, ECL_NIL, /* adjustable */
                             ECL_NIL /* fill-pointer */,
                             ECL_NIL /* displaced_to */,
                             ECL_NIL /* displacement */);
```

2.13.3.4 Accessors

Reading and writing array elements

Functions

<code>cl_object ecl_aref (cl_object array, cl_index row_major_index);</code>	[Function]
<code>cl_object ecl_aset (cl_object array, cl_index row_major_index, cl_object new_value);</code>	[Function]
<code>cl_object ecl_aref1 (cl_object vector, cl_index row_major_index);</code>	[Function]

`cl_object ecl_aset1 (cl_object vector, cl_index row_major_index, [Function]
cl_object new_value);`

Description `ecl_aset` accesses an array using the supplied *row_major_index*, checking the array bounds and returning a Lisp object for the value at that position. `ecl_aset` does the converse, storing a Lisp value at the given *row_major_index*.

The first argument to `ecl_aset` or `ecl_aset1` is an array of any number of dimensions. For an array of rank *N* and dimensions *d1*, *d2* ... up to *dN*, the row major index associated to the indices (*i1*, *i2*, ... *iN*) is computed using the formula $i1 + d1 * (i2 + d2 * (i3 + \dots))$.

`ecl_aset1` and `ecl_aset1` are specialized versions that only work with one-dimensional arrays or vectors. They verify that the first argument is indeed a vector.

All functions above check that the index does not exceed the array bounds, that the values match the array element type and that the argument is an array (or a vector). If these conditions are not met, a **type-error** is signaled.

2.13.3.5 Array properties

Array size, fill pointer, etc.

Functions

`cl_eltype ecl_array_eltype (cl_object array);` [Function]

`cl_index ecl_array_rank (cl_object array);` [Function]

`cl_index ecl_array_dimension (cl_object array, cl_index index);` [Function]

Description These functions query various properties of the arrays. Some of them belong to the list of functions in the Common Lisp package, without any need for specialized versions. More precisely

- `ecl_array_eltype` returns the array element type, with the encoding found in the enumeration `cl_eltype`.
- `ecl_array_rank` returns the number of dimensions of the vector or array.
- `ecl_array_dimension` queries the dimension of an array, where *index* is a non-negative integer between 0 and `ecl_array_dimension(array)-1`.

2.13.3.6 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function
make-array	<code>cl_object cl_make_array(cl_narg nargs, cl_object dimension...)</code>
adjust-array	<code>cl_object cl_adjust_array(cl_narg nargs, cl_object array, cl_object dimensions, ...)</code>
adjustable-array-p	<code>cl_object cl_adjustable_array_p(cl_object array)</code>
aref	<code>cl_object cl_aref(cl_narg nargs, cl_object array, ...)</code>
(setf aref)	<code>cl_object si_aset(cl_narg nargs, cl_object array, ...)</code>
array-dimension	<code>cl_object cl_array_dimension(cl_object array, cl_object index)</code>

array-dimensions	cl_object cl_array_dimensions(cl_object array)
array-element-type	cl_object cl_array_element_type(cl_object array)
array-has-fill-pointer-p	cl_object cl_array_has_fill_pointer_p(cl_object array)
array-displacement	cl_object cl_array_displacement(cl_object array)
array-in-bounds-p	cl_object cl_array_in_bounds_p(cl_narg nargs, cl_object array, ...)
array-rank	cl_object cl_array_rank(cl_object array)
array-row-major-index	cl_object cl_array_row_major_index(cl_narg nargs, cl_object array, ...)
array-total-size	cl_object cl_array_total_size(cl_object array)
arrayp	cl_object cl_arrayp(cl_object array)
fill-pointer	cl_object cl_fill_pointer(cl_object array)
(setf fill-pointer)	cl_object si_fill_pointer_set(cl_object array, cl_object fill_pointer)
row-major-aref	cl_object cl_row_major_aref(cl_object array, cl_object index)
(setf row-major-aref)	cl_object si_row_major_aset(cl_object array, cl_object index, cl_object value)
upgraded-array-element-type	cl_object cl_upgraded_array_element_type(cl_narg nargs, cl_object typespec, ...)
simple-vector-p	cl_object cl_simple_vector_p(cl_object object)
svref	cl_object cl_svref(cl_object simple_vector, cl_object index)
(setf svref)	cl_object si_svset(cl_object simple_vector, cl_object index, cl_object value)
vector	cl_object cl_vector(cl_narg nargs, ...)
vector-pop	cl_object cl_vector_pop(cl_object vector)
vector-push	cl_object cl_vector_push(cl_object new_element, cl_object vector)
vector-push-extend	cl_object cl_vector_push_extend(cl_narg nargs, cl_object new_element, cl_object vector, ...)
vectorp	cl_object cl_vectorp(cl_object object)
bit	cl_object cl_bit(cl_narg nargs, cl_object bit_array, ...)
(setf bit)	cl_object si_aset(cl_narg nargs, cl_object array, ...)
sbit	cl_object cl_sbit(cl_narg nargs, cl_object bit_array, ...)
(setf sbit)	cl_object si_aset(cl_narg nargs, cl_object array, ...)
bit-and	cl_object cl_bit_and(cl_narg nargs, cl_object array1, cl_object array2, ...)
bit-andc1	cl_object cl_bit_andc1(cl_narg nargs, cl_object array1, cl_object array2, ...)
bit-andc2	cl_object cl_bit_andc2(cl_narg nargs, cl_object array1, cl_object array2, ...)
bit-eqv	cl_object cl_bit_eqv(cl_narg nargs, cl_object array1, cl_object array2, ...)
bit-ior	cl_object cl_bit_ior(cl_narg nargs, cl_object array1, cl_object array2, ...)

bit-nand	cl_object cl_bit_nand(cl_narg nargs, cl_object array1, cl_object array2, ...)
bit-nor	cl_object cl_bit_nor(cl_narg nargs, cl_object array1, cl_object array2, ...)
bit-orc1	cl_object cl_bit_orc1(cl_narg nargs, cl_object array1, cl_object array2, ...)
bit-orc2	cl_object cl_bit_orc1(cl_narg nargs, cl_object array1, cl_object array2, ...)
bit-xor	cl_object cl_bit_xor(cl_narg nargs, cl_object array1, cl_object array2, ...)
bit-not	cl_object cl_bit_not(cl_narg nargs, cl_object array, ...)
bit-vector-p	cl_object cl_bit_vector_p(cl_object object)
simple-bit-vector-p	cl_object cl_simple_bit_vector_p(cl_object object)

2.14 Strings

2.14.1 String types & Unicode

The ECL implementation of strings is ANSI Common-Lisp compliant. There are basically four string types as shown in Table 2.7. As explained in Section 2.11 [Characters], page 47, when Unicode support is disabled, **character** and **base-character** are the same type and the last two string types are equivalent to the first two.

Abbreviation	Expanded type	Remarks
string	(array character (*))	8 or 32 bits per character, adjustable.
simple-string	(simple-array character (*))	8 or 32 bits per character, not adjustable nor displaced.
base-string	(array base-char (*))	8 bits per character, adjustable.
simple-base-string	(simple-array base-char (*))	8 bits per character, not adjustable nor displaced.

Table 2.7: Common Lisp string types

It is important to remember that strings with unicode characters can only be printed readably when the external format supports those characters. If this is not the case, ECL will signal a **serious-condition**. This condition will abort your program if not properly handled.

2.14.2 C reference

2.14.2.1 Base string constructors

Building strings of C data

Functions

cl_object ecl_alloc_adjustable_base_string (cl_index length); [Function]

```
cl_object ecl_alloc_simple_base_string (cl_index length);           [Function]
cl_object ecl_make_simple_base_string (const char* data,           [Function]
                                     cl_fixnum length);
cl_object ecl_make_constant_base_string (const char* data,        [Function]
                                     cl_fixnum length);
```

Description These are different ways to create a base string, which is a string that holds a small subset of characters, the **base-char**, with codes ranging from 0 to 255. **ecl_alloc_simple_base_string** creates an empty string with that much space for characters and a fixed length. The string does not have a fill pointer and cannot be resized, and the initial data is unspecified

ecl_alloc_adjustable_base_string is similar to the previous function, but creates an adjustable string with a fill pointer. This means that the length of the string can be changed and the string itself can be resized to accommodate more data.

The other constructors create strings but use some preexisting data. **ecl_make_simple_base_string** creates a string copying the data that the user supplies, and using freshly allocated memory. **ecl_make_constant_base_string** on the other hand, does not allocate memory, but simply uses the supplied pointer as buffer for the string. This last function should be used with care, ensuring that the supplied buffer is not deallocated. If the *length* argument of these functions is -1, the length is determined by **strlen**.

2.14.2.2 String accessors

Reading and writing characters into a string

Functions

```
ecl_character ecl_char (cl_object string, cl_index index);         [Function]
ecl_character ecl_char_set (cl_object string, cl_index index,      [Function]
                           ecl_character c);
```

Description Access to string information should be done using these two functions. The first one implements the equivalent of the **char** function from Common Lisp, returning the character that is at position *index* in the string *string*.

The counterpart of the previous function is **ecl_char_set**, which implements (**setf char**) and stores character *c* at the position *index* in the given string.

Both functions check the type of their arguments and verify that the indices do not exceed the string boundaries. Otherwise they signal a **serious-condition**.

2.14.2.3 ANSI dictionary

Common Lisp and C equivalence

Lisp symbol	C function
simple-string-p	cl_object cl_simple_string_p(cl_object string)
char	cl_object cl_char(cl_object string, cl_object index)
(setf char)	cl_object si_char_set(cl_object string, cl_object index, cl_object char)
schar	cl_object cl_schar(cl_object string, cl_object index)
(setf schar)	cl_object si_char_set(cl_object string, cl_object index, cl_object char)

string	cl-object cl_string(cl-object x)
string-upcase	cl-object cl_string_upcase(cl_narg nargs, cl-object string, ...)
string-downcase	cl-object cl_string_downcase(cl_narg nargs, cl-object string, ...)
string-capitalize	cl-object cl_string_capitalize(cl_narg nargs, cl-object string, ...)
nstring-upcase	cl-object cl_nstring_upcase(cl_narg nargs, cl-object string, ...)
nstring-downcase	cl-object cl_nstring_downcase(cl_narg nargs, cl-object string, ...)
nstring-capitalize	cl-object cl_nstring_capitalize(cl_narg nargs, cl-object string, ...)
string-trim	cl-object cl_string_trim(cl-object character_bag, cl-object string)
string-left-trim	cl-object cl_string_left_trim(cl-object character_bag, cl-object string)
string-right-trim	cl-object cl_string_right_trim(cl-object character_bag, cl-object string)
string	cl-object cl_string(cl-object x)
string=	cl-object cl_stringE(cl_narg nargs, cl-object string1, cl-object string2, ...)
string/=	cl-object cl_stringNE(cl_narg nargs, cl-object string1, cl-object string2, ...)
string<	cl-object cl_stringL(cl_narg nargs, cl-object string1, cl-object string2, ...)
string>	cl-object cl_stringG(cl_narg nargs, cl-object string1, cl-object string2, ...)
string<=	cl-object cl_stringLE(cl_narg nargs, cl-object string1, cl-object string2, ...)
string>=	cl-object cl_stringGE(cl_narg nargs, cl-object string1, cl-object string2, ...)
string-equal	cl-object cl_string_equal(cl_narg nargs, cl-object string1, cl-object string2, ...)
string-not-equal	cl-object cl_string_not_equal(cl_narg nargs, cl-object string1, cl-object string2, ...)
string-lessp	cl-object cl_string_lessp(cl_narg nargs, cl-object string1, cl-object string2, ...)
string-greaterp	cl-object cl_string_greaterp(cl_narg nargs, cl-object string1, cl-object string2, ...)
string-not-greaterp	cl-object cl_string_not_greaterp(cl_narg nargs, cl-object string1, cl-object string2, ...)
string-not-lessp	cl-object cl_string_not_lessp(cl_narg nargs, cl-object string1, cl-object string2, ...)
stringp	cl-object cl_stringp(cl-object x)
make-string	cl-object cl_make_string(cl_narg nargs, cl-object size, ...)

2.15 Sequences

2.15.1 C Reference

2.15.1.1 ANSI dictionary

Common Lisp and C equivalence

Lisp symbol	C function
concatenate	cl-object cl_concatenate(cl_narg nargs, cl-object result_type, ...)
copy-seq	cl-object cl_copy_seq(cl-object sequence)
count	cl-object cl_count(cl_narg nargs, cl-object item, cl-object sequence, ...)
count-if	cl-object cl_count_if(cl_narg nargs, cl-object predicate, cl-object sequence, ...)
count-if-not	cl-object cl_count_if_not(cl_narg nargs, cl-object predicate, cl-object sequence, ...)
delete	cl-object cl_delete(cl_narg nargs, cl-object item, cl-object sequence, ...)
delete-if	cl-object cl_delete_if(cl_narg nargs, cl-object test, cl-object sequence, ...)
delete-if-not	cl-object cl_delete_if_not(cl_narg nargs, cl-object test, cl-object sequence, ...)
delete-duplicates	cl-object cl_delete_duplicates(cl_narg nargs, cl-object sequence, ...)
elt	cl-object cl_elt(cl-object sequence, cl-object index)
(setf elt)	cl-object si_elt_set(cl-object sequence, cl-object index, cl-object value)
fill	cl-object cl_fill(cl_narg nargs, cl-object sequence, cl-object item, ...)
find	cl-object cl_find(cl_narg nargs, cl-object item, cl-object sequence, ...)
find-if	cl-object cl_find_if(cl_narg nargs, cl-object predicate, cl-object sequence, ...)
find-if-not	cl-object cl_find_if_not(cl_narg nargs, cl-object predicate, cl-object sequence, ...)
length	cl-object cl_length(cl-object x)
make-sequence	cl-object cl_make_sequence(cl_narg nargs, cl-object result_type, cl-object size, ...)
map	cl-object cl_map(cl_narg nargs, cl-object result_type, cl-object function, , ...)
map-into	cl-object cl_map_into(cl_narg nargs, cl-object result_sequence, cl-object function, ...)
merge	cl-object cl_merge(cl_narg nargs, cl-object result_type, cl-object sequence1, cl-object sequence2, cl-object predicate, ...)
mismatch	cl-object cl_mismatch(cl_narg nargs, cl-object sequence1, cl-object sequence2, ...)
nreverse	cl-object cl_nreverse(cl-object sequence)
nsubstitute	cl-object cl_nsubstitute(cl_narg nargs, cl-object newitem, cl-object olditem, cl-object sequence, ...)
nsubstitute-if	cl-object cl_nsubstitute_if(cl_narg nargs, cl-object newitem, cl-object predicate, cl-object sequence, ...)

nsubstitute-if-not	cl_object cl_nsubstitute_if_not(cl_narg nargs, cl_object newitem, cl_object predicate, cl_object sequence, ...)
position	cl_object cl_position(cl_narg nargs, cl_object item, cl_object sequence, ...)
position-if	cl_object cl_position_if(cl_narg nargs, cl_object predicate, cl_object sequence, ...)
position-if-not	cl_object cl_position_if_not(cl_narg nargs, cl_object predicate, cl_object sequence, ...)
reduce	cl_object cl_reduce(cl_narg nargs, cl_object function, cl_object sequence, ...)
remove	cl_object cl_remove(cl_narg nargs, cl_object item, cl_object sequence, ...)
remove-if	cl_object cl_remove_if(cl_narg nargs, cl_object test, cl_object sequence, ...)
remove-if-not	cl_object cl_remove_if_not(cl_narg nargs, cl_object test, cl_object sequence, ...)
remove-duplicates	cl_object cl_remove_duplicates(cl_narg nargs, cl_object sequence, ...)
replace	cl_object cl_replace(cl_narg nargs, cl_object sequence1, cl_object sequence2, ...)
reverse	cl_object cl_reverse(cl_object sequence)
search	cl_object cl_search(cl_narg nargs, cl_object sequence1, cl_object sequence2, ...)
sort	cl_object cl_sort(cl_narg nargs, cl_object sequence, cl_object predicate, ...)
stable-sort	cl_object cl_stable_sort(cl_narg nargs, cl_object sequence, cl_object predicate, ...)
subseq	cl_object cl_subseq(cl_narg nargs, cl_object sequence, cl_object start, ...)
substitute	cl_object cl_substitute(cl_narg nargs, cl_object newitem, cl_object olditem, cl_object sequence, ...)
substitute-if	cl_object cl_substitute_if(cl_narg nargs, cl_object newitem, cl_object predicate, cl_object sequence, ...)
substitute-if-not	cl_object cl_substitute_if_not(cl_narg nargs, cl_object newitem, cl_object predicate, cl_object sequence, ...)

2.16 Hash tables

2.16.1 Extensions

2.16.1.1 Weakness in hash tables

Weak hash tables allow the garbage collector to reclaim some of the entries if they are not strongly referenced elsewhere. ECL supports four kinds of weakness in hash tables: `:key`, `:value`, `:key-and-value` and `:key-or-value`.

To make hash table weak, programmer has to provide `:weakness` keyword argument to `cl:make-hash-table` with the desired kind of weakness value (`nil` means that the hash table has only strong references).

For more information see Weak References - Data Types and Implementation (<https://www.haible.de/bruno/papers/cs/weak/WeakDatastructures-writeup.html>) by Bruno Haible.

```
ext:hash-table-weakness ht [Function]
  Returns type of the hash table weakness. Possible return values are: :key, :value,
  :key-and-value, :key-or-value or nil.
```

2.16.1.2 Thread-safe hash tables

By default ECL doesn't protect hash tables from simultaneous access for performance reasons. Read and write access may be synchronized when `:synchronized` keyword argument to `make-hash-table` is `t` - (`make-hash-table :synchronized t`).

```
ext:hash-table-synchronized-p ht [Function]
  Predicate answering whether hash table is synchronized or not.
```

2.16.1.3 Hash tables serialization

```
ext:hash-table-content ht [Function]
  Returns freshly consed list of pairs (key . val) being contents of the hash table.
```

```
ext:hash-table-fill ht values [Function]
  Fills ht with values being list of (key . val). Hash table may have some content
  already, but conflicting keys will be overwritten.
```

2.16.1.4 Custom equivalence predicate

`make-hash-table` may accept arbitrary `:test` keyword for the equivalence predicate. If it is not one of the standard predicates (`:eq`, `:eql`, `:equal`, `:equalp`) a keyword argument `:hashing-function` must be a function accepting one argument and returning a positive fixnum. Otherwise the argument is ignored.

2.16.1.5 Example

```
CL-USER> (defparameter *ht*
           (make-hash-table :synchronized t
                           :weakness :key-or-value))

*HT*

CL-USER> (ext:hash-table-weakness *ht*)
:KEY-OR-VALUE

CL-USER> (ext:hash-table-synchronized-p *ht*)
T

CL-USER> (ext:hash-table-fill *ht* '((:foo 3) (:bar 4) (:quux 5)))
#<hash-table 000055b1229e0b40>
```

```
CL-USER> (ext:hash-table-content *ht*)
((#<weak-pointer 000055b121866350> . #<weak-pointer 000055b121866320>)
 (#<weak-pointer 000055b121866370> . #<weak-pointer 000055b121866360>)
 (#<weak-pointer 000055b121866390> . #<weak-pointer 000055b121866380>))
```

2.16.2 C Reference

2.16.2.1 ANSI dictionary

Common Lisp and C equivalence

Lisp symbol	C function
clrhash	cl_object cl_clrhash(cl_object hash_table)
gethash	cl_object cl_gethash(cl_narg nargs, cl_object key, cl_object hash_table, ...)
(setf gethash)	cl_object si_hash_set(cl_object key, cl_object hash_table, cl_object value)
hash-table-count	cl_object cl_hash_table_count(cl_object hash_table)
hash-table-p	cl_object cl_hash_table_p(cl_object hash_table)
hash-table-rehash-size	cl_object cl_hash_table_rehash_size(cl_object hash_table)
hash-table-rehash-threshold	cl_object cl_hash_table_rehash_threshold(cl_object hash_table)
hash-table-size	cl_object cl_hash_table_size(cl_object hash_table)
hash-table-test	cl_object cl_hash_table_test(cl_object hash_table)
make-hash-table	cl_object cl_make_hash_table(cl_narg nargs, ...)
maphash	cl_object cl_maphash(cl_object function, cl_object hash_table)
remhash	cl_object cl_remhash(cl_object key, cl_object hash_table)
sxhash	cl_object cl_sxhash(cl_object object)

2.17 Filenames

2.17.1 Syntax

A pathname in the file system of Common-Lisp consists of six elements: host, device, directory, name, type and version. Pathnames are read and printed using the `#P` reader macro followed by the namestring. A namestring is a string which represents a pathname. The syntax of namestrings for logical pathnames is well explained in the ANSI [ANSI, see [Bibliography], page 193] and it can be roughly summarized as follows:

```
[hostname:][;][directory-item;]0 or more[name][.type[.version]]
  hostname = word
  directory-item = wildcard-word
  type, name = wildcard-word without dots
```

Here, *wildcard-word* is a sequence of any character excluding `#\Null` and dots. *word* is like a *wildcard-word* but asterisks are excluded.

The way ECL parses a namestring is by first looking for the *hostname* component in the previous template. If it is found and it corresponds to a previously defined logical hostname, it assumes that the namestring corresponds to a logical pathname. If *hostname* is not found or it is not a logical hostname, then ECL tries the physical pathname syntax

```
[device:] [[//hostname/] [directory-item/] 0 or more[name] [.type]
  device, hostname = word
  directory-item = wildcard-word
  type = wildcard-word without dots
  name = [.]wildcard-word
```

If this syntax also fails, then the namestring is not a valid pathname string and a `parse-error` will be signaled.

It is important to remark that in ECL, all physical namestrings result into pathnames with a version equal to `:newest`. Pathnames which are not logical and have any other version (i. e. `nil` or a number), cannot be printed readably, but can produce a valid namestring which results of ignoring the version.

Finally, an important rule applies to physical namestrings: if a namestring contains one or more periods '.', the last period separates the namestring into the file name and the filetype. However, a namestring with a single leading period results in a name with a period in it. This is for compatibility with Unix filenames such as `.bashrc`, where the leading period indicates that the file is hidden.

The previous rule has an important consequence, because it means that if you want to create a pathname without a name, you have to do it explicitly. In other words, `".*"` is equivalent to `(make-pathname :name ".*" :type nil)`, while `(make-pathname :name nil :type :wild)` creates a pathname whose type is a wildcard.

The following table illustrates how the physical pathnames work with practical examples.

Namestring	Name	Type	Directory	Device
"foo.lsp"	"foo"	"lsp"	nil	nil
".bashrc"	".bashrc"	nil	nil	nil
".ecl.lsp"	".ecl"	"lsp"	nil	nil
"foo.*"	"foo"	:wild	nil	nil
"*.*"	:wild	:wild	nil	nil
"ecl/build/bare.lsp"	"bare"	"lsp"	(:relative "ecl" "build")	nil
"ecl/build/"	nil	nil	(:relative "ecl" "build")	nil
"../../ecl/build/"	nil	nil	(:relative :up :up "ecl" "build")	nil
"/etc/"	nil	nil	(:absolute "etc")	nil
"C:/etc/"	nil	nil	(:absolute "etc")	"C"
"*.*"	"*.*"	nil	nil	nil
#.(make-pathname :type "*")	nil	:wild	nil	nil

Table 2.8: Examples of physical namestrings

2.17.2 Wild pathnames and matching

ECL accepts four kind of wildcards in pathnames.

- A single wildcard in a directory component, file name, type or version is parsed as the `:wild` value. See for instance `"*.*"`, `"/home/*/bashrc"`, etc
- A double wildcard in a directory component, such as in `"/home/**/"` is parsed as the `:wild-inferiors`, and matches any number of directories, even nested ones, such as: `/home/`, `/home/jlr`, `/home/jlr/lib`, etc.
- An isolated wildcard `"log*.txt"` matches any number of characters: `log.txt`, `log_back.txt`, etc.
- A question mark `"log?.txt"` matches a single character: `log1.txt`, `log2.txt`...

The matching rules in Common Lisp and ECL are simple but have some unintuitive consequences when compared to Unix/DOS rules. The most important one is that directories must always end with a trailing slash `/`, as in `#p"/my/home/directory/"`. Second to that, `nil` values can only be matched by `nil` and `:wild`. Hence, `"*"` can only match files without file type. For some examples see Section 2.18.1 [Files - Dictionary], page 69.

2.17.3 C Reference

2.17.3.1 ANSI dictionary

Common Lisp and C equivalence

Lisp symbol	C function
directory-namestring	cl-object cl_directory_namestring(cl-object pathname)
enough-namestring	cl-object cl_enough_namestring(cl_narg nargs, cl-object pathname, ...)
file-namestring	cl-object cl_file_namestring(cl-object pathname)
host-namestring	cl-object cl_host_namestring(cl-object pathname)
load-logical-pathname-translations	cl-object cl_load_logical_pathname_translations(cl-object host)
logical-pathname-translations	cl-object cl_logical_pathname_translations(cl-object host)
logical-pathname	cl-object cl_logical_pathname(cl-object pathspec)
make-pathname	cl-object cl_make_pathname(cl_narg nargs, ...)
merge-pathnames	cl-object cl_merge_pathnames(cl_narg nargs, cl-object pathname,...)
namestring	cl-object cl_namestring(cl-object pathname)
parse-namestring	cl-object cl_parse_namestring(cl_narg nargs, cl-object thing, ...)
pathname	cl-object cl_pathname(cl-object pathspec)
pathname-device	cl-object cl_pathname_device(cl_narg nargs, cl-object pathname, ...)
pathname-directory	cl-object cl_pathname_directory(cl_narg nargs, cl-object pathname, ...)
pathname-host	cl-object cl_pathname_host(cl_narg nargs, cl-object pathname, ...)

<code>pathname-match-p</code>	<code>cl-object</code> <code>cl-pathname-match-p</code> (<code>cl-object</code> <code>pathname</code> , <code>cl-object</code> <code>wildcard</code>)
<code>pathname-name</code>	<code>cl-object</code> <code>cl-pathname-name</code> (<code>cl-narg</code> <code>narg</code> , <code>cl-object</code> <code>path-</code> <code>name</code> , ...)
<code>pathname-type</code>	<code>cl-object</code> <code>cl-pathname-type</code> (<code>cl-narg</code> <code>narg</code> , <code>cl-object</code> <code>path-</code> <code>name</code> , ...)
<code>pathname-version</code>	<code>cl-object</code> <code>cl-pathname-version</code> (<code>cl-object</code> <code>pathname</code>)
<code>pathnamep</code>	<code>cl-object</code> <code>cl-pathnamep</code> (<code>cl-object</code> <code>object</code>)
<code>translate-logical-pathname</code>	<code>cl-object</code> <code>cl-translate-logical-pathname</code> (<code>cl-narg</code> <code>narg</code> , <code>cl-object</code> <code>pathname</code> , ...)
<code>translate-pathname</code>	<code>cl-object</code> <code>cl-translate-pathname</code> (<code>cl-narg</code> <code>narg</code> , <code>cl-object</code> <code>source</code> , <code>cl-object</code> <code>from-wildcard</code> , <code>cl-object</code> <code>to-wildcard</code> , ...)
<code>wild-pathname-p</code>	<code>cl-object</code> <code>cl-wild-pathname-p</code> (<code>cl-narg</code> <code>narg</code> , <code>cl-object</code> <code>path-</code> <code>name</code> , ...)

2.18 Files

2.18.1 Dictionary

directory *paths**spec* [Function]

This function does not have any additional arguments other than the ones described in ANSI [ANSI, see [Bibliography], page 193]. To list files and directories, it follows the rules for matching pathnames described in Section 2.17.2 [Filenames - Wild pathnames and matching], page 68. In short, you have the following practical examples:

Argument	Meaning
<code>"/home/jlr/*.*)"</code>	List all files in directory <code>/home/jlr/</code> Note that it lists only files, not directories!
<code>"/home/jlr/*"</code>	Same as before, but only files without type.
<code>"/home/jlr/*/"</code>	List all directories contained in <code>/home/jlr/</code> . Nested directories are not navigated.
<code>"/home/jlr/**/*.*"</code>	List all files in all directories contained in <code>/home/jlr/</code> , recursively. Nested directories are navigated.

Table 2.9: Examples of using **directory**

rename-file *filespec* *new-name* **&key** (*if-exists* **:error**) [Function]

In addition to the arguments described in ANSI [ANSI, see [Bibliography], page 193], the **rename-file** function in ECL has an **:if-exists** keyword argument that specifies what happens when a file with the new name already exists. Valid values of this argument are:

Argument	Behaviour of the rename-file function
:error	Signal an error
:supersede , t	Overwrite the existing file
nil	Don't overwrite the existing file, don't signal an error

2.18.2 C Reference

2.18.2.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function
delete-file	cl_object cl_delete_file(cl_object filespec)
directory	cl_object cl_directory(cl_narg nargs, cl_object pathspec, ...)
ensure-directories-exist	cl_object cl_ensure_directories_exist(cl_narg nargs, cl_object pathspec, ...)
file-author	cl_object cl_file_author(cl_object pathspec)
file-error-pathname	[Only in Common Lisp]
file-write-date	cl_object cl_file_write_date(cl_object pathspec)
probe-file	cl_object cl_probe_file(cl_object pathspec)
rename-file	cl_object cl_rename_file(cl_narg nargs, cl_object filespec, cl_object new_name, ...)
truename	cl_object cl_truename(cl_object filespec)

2.19 Streams

2.19.1 ANSI Streams

2.19.1.1 Supported types

ECL implements all stream types described in ANSI [ANSI, see [Bibliography], page 193]. Additionally, when configured with option `--enable-clos-streams`, ECL includes a version of Gray streams where any object that implements the appropriate methods (`stream-input-p`, `stream-read-char`, etc) is a valid argument for the functions that expect streams, such as `read`, `print`, etc.

2.19.1.2 Element types

ECL distinguishes between two kinds of streams: character streams and byte streams. *Character streams* only accept and produce characters, written or read one by one, with `write-char` or `read-char`, or in chunks, with `write-sequence` or any of the Lisp printer functions. Character operations are conditioned by the external format, as described in Section 2.19.1.3 [Streams - External formats], page 70.

ANSI Common Lisp also supports binary streams. Here input and output is performed in chunks of bits. Binary streams are created with the function `open` passing as argument a subtype of integer and the implementation is free to round up that integer type to the closest size it supports. In particular ECL rounds up the size to a multiple of a byte. For example, the form `(open "foo.bin" :direction :output :element-type '(unsigned-byte 13))`, will open the file `foo.bin` for writing, using 16-bit words as the element type.

2.19.1.3 External formats

An *external format* is an encoding for characters that maps character codes to a sequence of bytes, in a one-to-one or one-to-many fashion. External formats are also known as

"character encodings" in the programming world and are an essential ingredient to be able to read and write text in different languages and alphabets.

ECL has one of the most complete supports for *external formats*, covering all of the usual codepages from the Windows and Unix world, up to the more recent UTF-8, UCS-2 and UCS-4 formats, all of them with big and small endian variants, and considering different encodings for the newline character.

However, the set of supported external formats depends on the size of the space of character codes. When ECL is built with Unicode support (the default option), it can represent all known characters from all known codepages, and thus all external formats are supported. However, when ECL is built with the restricted character set, it can only use one codepage (the one provided by the C library), with a few variants for the representation of end-of-line characters.

In ECL, an external format designator is defined recursively as either a symbol or a list of symbols. The grammar is as follows

```
external-format-designator :=  
    symbol |  
    ( {symbol}+ )
```

and the table of known symbols is shown below. Note how some symbols (`:cr`, `:little-endian`, etc.) just modify other external formats.

Symbols	Codepage or encoding	Unicode required
:cr	#\Newline is Carriage Return	No
:crlf	#\Newline is Carriage Return followed by Linefeed	No
:lf	#\Newline is Linefeed	No
:little-endian	Modify UCS to use little-endian encoding.	No
:big-endian	Modify UCS to use big-endian encoding.	No
:utf-8 :utf8	Unicode UTF-8	Yes
:ucs-2 :ucs2 :utf-16 :utf16	UCS-2 encoding with BOM. Defaults to big-endian when writing or if no BOM is detected when reading.	Yes
:unicode		
:ucs-2le :ucs2le :utf-16le	UCS-2 with little-endian encoding	Yes
:ucs-2be :ucs2be :utf-16be	UCS-2 with big-endian encoding	Yes
:ucs-4 :ucs4 :utf-32 :utf32	UCS-4 encoding with BOM. Defaults to big-endian when writing or if no BOM is detected when reading.	Yes
:ucs-4le :ucs4le :utf-32le	UCS-4 with little-endian encoding	Yes
:ucs-4be :ucs4be :utf-32be	UCS-4 with big-endian encoding	Yes
:iso-8859-1 :iso8859-1	Latin-1 encoding	Yes
:latin-1 :cp819 :ibm819		
:iso-8859-2 :iso8859-2	Latin-2 encoding	Yes
:latin-2 :latin2		
:iso-8859-3 :iso8859-3	Latin-3 encoding	Yes
:latin-3 :latin3		
:iso-8859-4 :iso8859-4	Latin-4 encoding	Yes
:latin-4 :latin4		
:iso-8859-5 :cyrillic	Latin-5 encoding	Yes
:iso-8859-6 :arabic :asmo-708	Latin-6 encoding	Yes
:ecma-114		
:iso-8859-7 :greek8 :greek	Greek encoding	Yes
:ecma-118		
:iso-8859-8 :hebrew	Hebrew encoding	Yes
:iso-8859-9 :latin-5 :latin5	Latin-5 encoding	Yes
:iso-8859-10 :iso8859-10	Latin-6 encoding	Yes
:latin-6 :latin6		
:iso-8859-13 :iso8859-13	Latin-7 encoding	Yes
:latin-7 :latin7		
:iso-8859-14 :iso8859-14	Latin-8 encoding	Yes
:latin-8 :latin8		
:iso-8859-15 :iso8859-15	Latin-7 encoding	Yes
:latin-9 :latin9		
:dos-cp437 :ibm-437	IBM CP 437	Yes
:dos-cp850 :ibm-850 :cp850	Windows CP 850	Yes
:dos-cp852 :ibm-852	IBM CP 852	Yes
:dos-cp855 :ibm-855	IBM CP 855	Yes

2.19.2 Dictionary

2.19.2.1 Sequence Streams

`ext:sequence-stream` [System Class]

Class Precedence List `ext:sequence-stream, stream, t`

Description Sequence streams work similar to string streams for vectors. The supplied vectors that the streams read from or write to must have a byte sized element type, i.e. (**signed-byte 8**), (**unsigned-byte 8**) or **base-char**.

The semantics depend on the vector element type and the external format of the stream. If no external format is supplied and the element type is an integer type, the stream is a binary stream and accepts only integers of the same type as the element type of the vector. Otherwise, the stream accepts both characters and integers and converts them using the given external format. If the element type is **base-char**, the elements of the vectors are treated as bytes. This means that writing a character may use multiple elements of the vector, whose **char-codes** will be equal to the values of the bytes comprising the character in the given external format.

`ext:make-sequence-input-stream` *vector* **&key** (*start* 0) (*end* **nil**) [Function]
(*external-format* **nil**)

Create a sequence input stream with the subsequence bounded by *start* and *end* of the given vector.

`ext:make-sequence-output-stream` *vector* **&key** (*external-format* **nil**) [Function]

Create a sequence output stream.

Example:

Using sequence streams to convert to a UTF8 encoded base string

```
CL-USER> (defvar *output* (make-array 20 :element-type 'base-char :adjustable t :fill-
*OUTPUT*
CL-USER> (defvar *stream* (ext:make-sequence-output-stream *output* :external-format :
*STREAM*
CL-USER> (write-string "Spätzle mit Soß'" *stream*)
"Spätzle mit Soß'"
CL-USER> *output*
"Sp  tzle mit So  \237'"
```

2.19.2.2 File Stream Extensions

`ext:set-buffering-mode` *stream mode* [Function]

Control the buffering mode of a stream

Synopsis

stream an ANSI stream

mode one of **nil**, **:none**, **:line**, **:line-buffered**, **:full** or **:full-buffered**

returns The supplied stream

Description If *mode* is `nil` or `:none`, *stream* will not be buffered, if it is `:line` or `:line-buffered` resp. `:full` or `:fully-buffered`, *stream* will be line resp. fully buffered. If the stream does not support buffering, nothing will happen.

`ext:file-stream-fd` *file-stream* [Function]
Return the POSIX file descriptor of *file-stream* as an integer

2.19.2.3 External Format Extensions

`ext:all-encodings` [Function]
Return a list of all supported external formats

`ext:character-coding-error` [Condition]
Character coding error

Class Precedence List `ext:character-coding-error`, `error`, `serious-condition`, `condition`, `t`

Methods

`ext:character-coding-error-external-format` *condition* [Function]
returns The external format of *condition*

Description Superclass of `ext:character-encoding-error` and `ext:character-decoding-error`.

`ext:character-encoding-error` [Condition]
Character encoding error

Class Precedence List `ext:character-encoding-error`, `ext:character-coding-error`, `error`, `serious-condition`, `condition`, `t`

Methods

`ext:character-encoding-error-code` *condition* [Function]
returns The character code of the character, which can't be encoded

Description Condition for characters, which can't be encoded with some external format.

`ext:character-decoding-error` [Condition]
Character decoding error

Class Precedence List `ext:character-decoding-error`, `ext:character-coding-error`, `error`, `serious-condition`, `condition`, `t`

Methods

`ext:character-decoding-error-octects` *condition* [Function]
returns A list of integers with the values of the unsigned char's which can't be decoded.

Description Condition for characters, which can't be decoded with some external format.

ext:stream-encoding-error [Condition]

Stream encoding error

Class Precedence List `ext:stream-encoding-error`, `ext:character-encoding-error`, `ext:character-coding-error`, `stream-error`, `error`, `serious-condition`, `condition`, `t`

Description This condition is signaled when trying to write a character to a stream, which can't be encoded with the streams external format.

ext:stream-decoding-error [Condition]

Stream decoding error

Class Precedence List `ext:stream-decoding-error`, `ext:character-decoding-error`, `ext:character-coding-error`, `stream-error`, `error`, `serious-condition`, `condition`, `t`

Description This condition is signaled when trying to read a character from a stream, which can't be decoded with the streams external format.

ext:encoding-error *stream external-format code* [Function]

Signal a `ext:stream-encoding-error` with the given *external-format* and *code*. Make a restart available so that the error can be ignored or the character can be replaced with a different one.

ext:decoding-error *stream external-format octets* [Function]

Signal a `ext:stream-decoding-error` with the given *external-format* and *octets*. Make a restart available so that the error can be ignored or the octets can be replaced with a character.

2.19.3 C Reference

2.19.3.1 ANSI dictionary

Common Lisp and C equivalence

Lisp symbol	C function
<code>broadcast-stream-streams</code>	<code>cl_object</code> <code>cl_broadcast_stream_streams(cl_object broadcast_stream)</code>
<code>clear-input</code>	<code>cl_object</code> <code>cl_clear_input(cl_narg nargs, ...)</code>
<code>clear-output</code>	<code>cl_object</code> <code>cl_clear_output(cl_narg nargs, ...)</code>
<code>close</code>	<code>cl_object</code> <code>cl_close(cl_narg nargs, cl_object stream, ...)</code>
<code>concatenated-stream-streams</code>	<code>cl_object</code> <code>cl_concatenated_stream_streams(cl_object concatenated_stream)</code>
<code>echo-stream-input-stream</code>	<code>cl_object</code> <code>cl_echo_stream_input_stream(cl_object echo_stream)</code>
<code>echo-stream-output-stream</code>	<code>cl_object</code> <code>cl_echo_stream_output_stream(cl_object echo_stream)</code>
<code>file-length</code>	<code>cl_object</code> <code>cl_file_position(cl_narg nargs, cl_object file_stream, ...)</code>
<code>file-position</code>	<code>cl_object</code> <code>cl_file_position(cl_object stream)</code>

file-string-length	cl-object cl_file_string_length(cl-object stream, cl-object object)
finish-output	cl-object cl_finish_output(cl_narg nargs, ...)
force-output	cl-object cl_force_output(cl_narg nargs, ...)
fresh-line	cl-object cl_fresh_line(cl_narg nargs, ...)
get-output-stream-string	cl-object cl_get_output_stream_string(cl-object string_output_stream)
input-stream-p	cl-object cl_input_stream_p(cl-object stream)
interactive-stream-p	cl-object cl_interactive_stream_p(cl-object stream)
listen	cl-object cl_listen(cl_narg nargs, cl-object stream, ...)
make-broadcast-stream	cl-object cl_make_broadcast_stream(cl_narg nargs, ...)
make-concatenated-stream	cl-object cl_make_concatenated_stream(cl_narg nargs,)
make-echo-stream	cl-object cl_make_echo_stream(cl-object input, cl-object output)
make-string-input-stream	cl-object cl_make_string_input_stream(cl_narg nargs, cl-object string, ...)
make-string-output-stream	cl-object cl_make_string_output_stream(cl_narg nargs, ...)
make-two-way-stream	cl-object cl_make_two_way_stream(cl-object input, cl-object output)
make-synonym-stream	cl-object cl_make_synonym_stream(cl-object symbol)
open	cl-object cl_open(cl_narg nargs, cl-object filespec, ...)
open-stream-p	cl-object cl_open_stream_p(cl-object stream)
output-stream-p	cl-object cl_output_stream_p(cl-object stream)
peek-char	cl-object cl_peek_char(cl_narg nargs, ...)
read-byte	cl-object cl_read_byte(cl_narg nargs, cl-object stream, ...)
read-char	cl-object cl_read_char(cl_narg nargs, ...)
read-char-no-hang	cl-object cl_read_char_no_hang(cl_narg nargs, ...)
read-line	cl-object cl_read_line(cl_narg nargs, ...)
read-sequence	cl-object cl_read_sequence(cl_narg nargs, cl-object sequence, cl-object stream, ...)
stream-element-type	cl-object cl_stream_element_type(cl-object stream)
stream-error-stream	[Only in Common Lisp]
stream-external-format	cl-object cl_stream_external_format(cl-object stream)
(setf stream-external-format)	cl-object si_stream_external_format_set(cl-object stream, cl-object format)
stream-p	cl-object cl_stream_p(cl-object object)
synonym-stream-symbol	cl-object cl_synonym_stream_symbol(cl-object synonym_stream)
terpri	cl-object cl_terpri(cl_narg nargs, ...)
two-way-stream-input-stream	cl-object cl_two_way_stream_input_stream(cl-object two_way_stream)
two-way-stream-output-stream	cl-object cl_two_way_stream_output_stream(cl-object two_way_stream)
unread-char	cl-object cl_unread_char(cl_narg nargs, cl-object character, ...)
write-byte	cl-object cl_write_byte(cl-object byte, cl-object stream)
write-char	cl-object cl_write_char(cl_narg nargs, cl-object character, ...)

write-line	cl-object cl_write_line(cl_narg nargs, cl-object string, ...)
write-string	cl-object cl_write_string(cl_narg nargs, cl-object string, ...)
write-sequence	cl-object cl_write_sequence(cl_narg nargs, cl-object sequence, cl-object stream, ...)
y-or-n-p	cl-object cl_y_or_n_p(cl_narg nargs, ...)
yes-or-no-p	cl-object cl_yes_or_no_p(cl_narg nargs, ...)

2.20 Printer

In all situations where the rules are well specified, ECL prints objects according to ANSI [ANSI, see [Bibliography], page 193]. The specification leaves however a number of cases as implementation dependent behavior. The output of ECL in those cases is summarized in Table 2.11. Except for the types **character** and **random-state**, most of those examples regard non-standard written forms **#<...>** cannot be read back using **read**. These printed representations are just informative and should not be used to reconstruct or compare objects.

Lisp type	Format	Remarks
package	#<package <i>name</i> >	
random-state	#<random-state array>	
bitvector	#<bit-vector <i>unique-id</i> >	Only when <i>*print-array*</i> is false.
vector	#<vector <i>unique-id</i> >	Only when <i>*print-array*</i> is false.
array	#<array <i>unique-id</i> >	Only when <i>*print-array*</i> is false.
hash-table	#<hash-table <i>unique-id</i> >	
readtable	#<readtable <i>unique-id</i> >	
interpreted function	#<bytecompiled-function <i>name-or-id</i> >	Name is a symbol.
machine compiled function	#<compiled-function <i>name</i> >	Name is a symbol.
input-stream	#<input stream "filename">	An stream that reads from <i>filename</i> .
output-stream	#<output stream "filename">	An stream that writes to <i>filename</i> .
probe-stream	#<probe stream "filename">	
string-input-stream	#<string-input stream from " <i>string-piece</i> ">	The string is the text left to be read.
string-output-stream	#<string-output stream <i>unique-id</i> >	
two-way-stream	#<two-way stream <i>unique-id</i> >	
echo-stream	#<echo stream <i>unique-id</i> >	
synonym-stream	#<synonym stream to <i>symbol</i> >	
broadcast-stream	#<broadcast stream <i>unique-id</i> >	
concatenated-stream	#<concatenated stream <i>unique-id</i> >	
closed-stream	#<closed ...>	The dots denote any of the above stream forms.

Table 2.11: Implementation-specific printed representation

2.20.1 C Reference

2.20.1.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function
copy-pprint-dispatch	cl_object cl_copy_pprint_dispatch(cl_narg nargs, ...)
pprint-dispatch	cl_object cl_pprint_dispatch(cl_narg nargs, cl_object object, ...)
pprint-fill	cl_object cl_pprint_fill(cl_narg nargs, cl_object stream, cl_object object, ...)
pprint-linear	cl_object cl_pprint_linear(cl_narg nargs, cl_object stream, cl_object object, ...)
pprint-tabular	cl_object cl_pprint_tabular(cl_narg nargs, cl_object stream, cl_object object, ...)
pprint-indent	cl_object cl_pprint_indent(cl_narg nargs, cl_object relative_to, cl_object n, ...)
pprint-newline	cl_object cl_pprint_newline(cl_narg nargs, cl_object kind, ...)
pprint-tab	cl_object cl_pprint_tab(cl_narg nargs, cl_object kind, cl_object colnum, cl_object colinc, ...)
print-object	[Only in Common Lisp]
set-pprint-dispatch	cl_object cl_set_pprint_dispatch(cl_narg nargs, cl_object type_spec, cl_object function, ...)
write	cl_object cl_write(cl_narg nargs, cl_object object, ...)
prin1	cl_object cl_prin1(cl_narg nargs, cl_object object, ...)
princ	cl_object cl_princ(cl_narg nargs, cl_object object, ...)
print	cl_object cl_print(cl_narg nargs, cl_object object, ...)
pprint	cl_object cl_pprint(cl_narg nargs, cl_object object, ...)
write-to-string	cl_object cl_write_to_string(cl_narg nargs, cl_object object, ...)
prin1-to-string	cl_object cl_prin1_to_string(cl_object object)
princ-to-string	cl_object cl_princ_to_string(cl_object object)
print-not-readable-object	[Only in Common Lisp]
format	cl_object cl_format(cl_narg nargs, cl_object stream, cl_object string, ...)

2.21 Reader

2.21.1 *read-supress*

The behaviour of ***read-supress*** is not fully specified in the standard. ECL tries to be as liberal as possible in the syntax that it accepts when ***read-suppress*** is true. Errors are only signaled in the following cases:

- End of file
- Unbalanced parantheses
- Invalid dispatching macro characters such as < or) (Undefined dispatching macro characters don't produce errors)

2.21.2 C Reference

2.21.2.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function
copy-readtable	cl_object cl_copy_readtable(cl_narg nargs, ...)
make-dispatch-macro-character	cl_object cl_make_dispatch_macro_character(cl_narg nargs, cl_object char, ...)
read	cl_object cl_read(cl_narg nargs, ...)
read-preserving-whitespace	cl_object cl_read_preserving_whitespace(cl_narg nargs, ...)
read-delimited-list	cl_object cl_read_delimited_list(cl_narg nargs, cl_object char, ...)
read-from-string	cl_object cl_read_from_string(cl_narg nargs, cl_object string, ...)
readtable-case	cl_object cl_readtable_case(cl_object readtable)
(setf readtable-case)	cl_object si_readtable_case_set(cl_object readtable, cl_object mode)
readtablep	cl_object cl_readtablep(cl_object object)
get-dispatch-macro-character	cl_object cl_get_dispatch_macro_character(cl_narg nargs, cl_object disp_char, cl_object sub_char, ...)
set-dispatch-macro-character	cl_object cl_set_dispatch_macro_character(cl_narg nargs, cl_object disp_char, cl_object sub_char, cl_object function, ...)
get-macro-character	cl_object cl_get_macro_character(cl_narg nargs, cl_object char, ...)
set-macro-character	cl_object cl_set_macro_character(cl_narg nargs, cl_object char, cl_object function, ...)
set-syntax-from-char	cl_object cl_set_syntax_from_char(cl_narg nargs, cl_object to_char, cl_object from_char, ...)

2.22 System construction

2.22.1 C Reference

2.22.1.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function
compile-file	[Only in Common Lisp]
compile-file-pathname	[Only in Common Lisp]
load	cl_object cl_load(cl_narg nargs, cl_object pathname, ...)
provide	cl_object cl_provide(cl_object module_name)
require	cl_object cl_require(cl_narg nargs, cl_object module_name, ...)

2.23 Environment

2.23.1 Dictionary

disassemble *function-designator** [Function]

Display the assembly code of a function

Synopsis

function-designator

A symbol which is bound to a function in the global environment, or a lambda form

Description As specified in ANSI [ANSI, see [Bibliography], page 193] this function outputs the internal representation of a compiled function, or of a lambda form, as it would look after being compiled.

ECL only has a particular difference: it has two different compilers, one based on bytecodes and one based on the C language. The output will thus depend on the arguments and on which compiler is active at the moment in which this function is run.

- If the argument is a bytecompiled function or a lambda form, it will be processed by the active compiler and the appropriate output (bytecodes or C) will be shown.
- If the argument is a C-compiled form, disassembling the function by showing its C source code is not possible, since that would require saving not only the lambda form of the function, but also the precise configuration of the compiler when the function was compiled. Hence no output will be shown.

trace *function-name** [Macro]

Follow the execution of functions

Synopsis (trace *function-name**)

function-name

{*symbol* | (*symbol* [*option form*]*)}

symbol A symbol which is bound to a function in the global environment. Not evaluated.

option One of :break, :break-after, :cond-before, :cond-after, :cond, :print, :print-after, :step

form A lisp form evaluated in an special environment.

returns List of symbols with traced functions.

Description Causes one or more functions to be traced. Each *function-name* can be a symbol which is bound to a function, or a list containing that symbol plus additional options. If the function bound to that symbol is called, information about the arguments and output of this function will be printed. Trace options will modify the amount of information and when it is printed.

Not that if the function is called from another function compiled in the same file, tracing might not be enabled. If this is the case, to enable tracing, recompile the caller with a **notinline** declaration for the called function.

`trace` returns a name list of those functions that were traced by the call to `trace`. If no *function-name* is given, `trace` simply returns a name list of all the currently traced functions.

Trace options cause the normal printout to be suppressed, or cause extra information to be printed. Each option is a pair of an option keyword and a value form. If an already traced function is traced again, any new options replace the old options and a warning might be printed. The lisp *form* accompanying the option is evaluated in an environment where `sys::args` contains the list of arguments to the function.

The following options are defined:

`:cond`, `:cond-before`, `:cond-after`

If `:cond-before` is specified, then `trace` does nothing unless *form* evaluates to true at the time of the call. `:cond-after` is similar, but suppresses the initial printout, and is tested when the function returns. `:cond` tries both before and after.

`:step` If *form* evaluates to true, the stepper is entered.

`:break`, `:break-after`

If specified, and *form* evaluates to true, then the debugger is invoked at the start of the function or at the end of the function according to the respective option.

`:print`, `:print-after`

In addition to the usual printout, the result of evaluating *form* is printed at the start of the function or at the end of the function, depending on the option. Multiple print options cause multiple values to be output, in the order in which they were introduced.

See also the following example:

```
> (defun abc (x)
    (if (>= x 10)
        x
        (abc (+ x (abc (1+ x))))))
> (trace abc)

> (abc 9)
1> (ABC 9)
| 2> (ABC 10)
| <2 (ABC 10)
| 2> (ABC 19)
| <2 (ABC 19)
<1 (ABC 19)
19
> (untrace abc)

(ABC)
;; Break if the first argument of the function is greater than 10
> (trace (abc :break (>= (first si::args) 10)))
```

```

((ABC :BREAK (>= (FIRST SI::ARGS) 10)))
> (abc 9)
1> (ABC 9)
| 2> (ABC 10)

Condition of type: SIMPLE-CONDITION
tracing ABC
Available restarts:

1. (CONTINUE) Return from BREAK.
2. (RESTART-TOPLEVEL) Go back to Top-Level REPL.

Broken at ABC. In: #<process TOP-LEVEL 0x1842f80>.
>>

```

2.23.2 C Reference

2.23.2.1 ANSI Dictionary

Common Lisp and C equivalence

Lisp symbol	C function
decode-universal-time	cl_object cl_decode_universal_time(cl_narg nargs, cl_object universal_time, ...)
encode-universal-time	cl_object cl_encode_universal_time(cl_narg nargs, cl_object second, cl_object minute, cl_object hour, cl_object date, cl_object month, cl_object year, ...)
get-universal-time	cl_object cl_get_universal_time(void)
get-decoded-time	cl_object cl_get_decoded_time(void)
sleep	cl_object cl_sleep(cl_object seconds)
apropos	cl_object cl_apropos(cl_narg nargs, cl_object string, ...)
apropos-list	cl_object cl_apropos_list(cl_narg nargs, cl_object string, ...)
describe	cl_object cl_describe(cl_narg nargs, cl_object object, ...)
describe-object	[Only in Common Lisp]
get-internal-real-time	cl_object cl_get_internal_real_time(void)
get-internal-run-time	cl_object cl_get_internal_run_time(void)
disassemble	[Only in Common Lisp]
documentation	[Only in Common Lisp]
room	[Only in Common Lisp]
ed	[Only in Common Lisp]
inspect	cl_object cl_inspect(cl_object object)
dribble	cl_object cl_dribble(cl_narg nargs, ...)
lisp-implementation-type	cl_object cl_lisp_implementation_type(void)
lisp-implementation-version	cl_object cl_lisp_implementation_version(void)
short-site-name	cl_object cl_short_site_name()
long-site-name	cl_object cl_long_site_name()

machine-instance	cl_object cl_machine_instance()
machine-type	cl_object cl_machine_type()
machine-version	cl_object cl_machine_version()
software-type	cl_object cl_software_type()
software-version	cl_object cl_software_version()
user-homedir-pathname	cl_object cl_user_homedir_pathname(cl_narg nargs, ...)

3 Extensions

3.1 System building

3.1.1 Compiling with ECL

In this section we will introduce topics on compiling Lisp programs. ECL is especially powerful on combining lisp programs with C programs. You can embed ECL as a lisp engine in C programs, or call C functions via Section 3.3 [Foreign Function Interface], page 98. We explain file types generated by some compilation approaches. For the examples, a GNU/Linux system and gcc as a development environment are assumed.

You can generate the following files with ECL:

1. Portable FASL file (.fasc)
2. Native FASL file (.fas, .fasb)
3. Object file (.o)
4. Static library
5. Shared library
6. Executable file

Relations among them are depicted below:

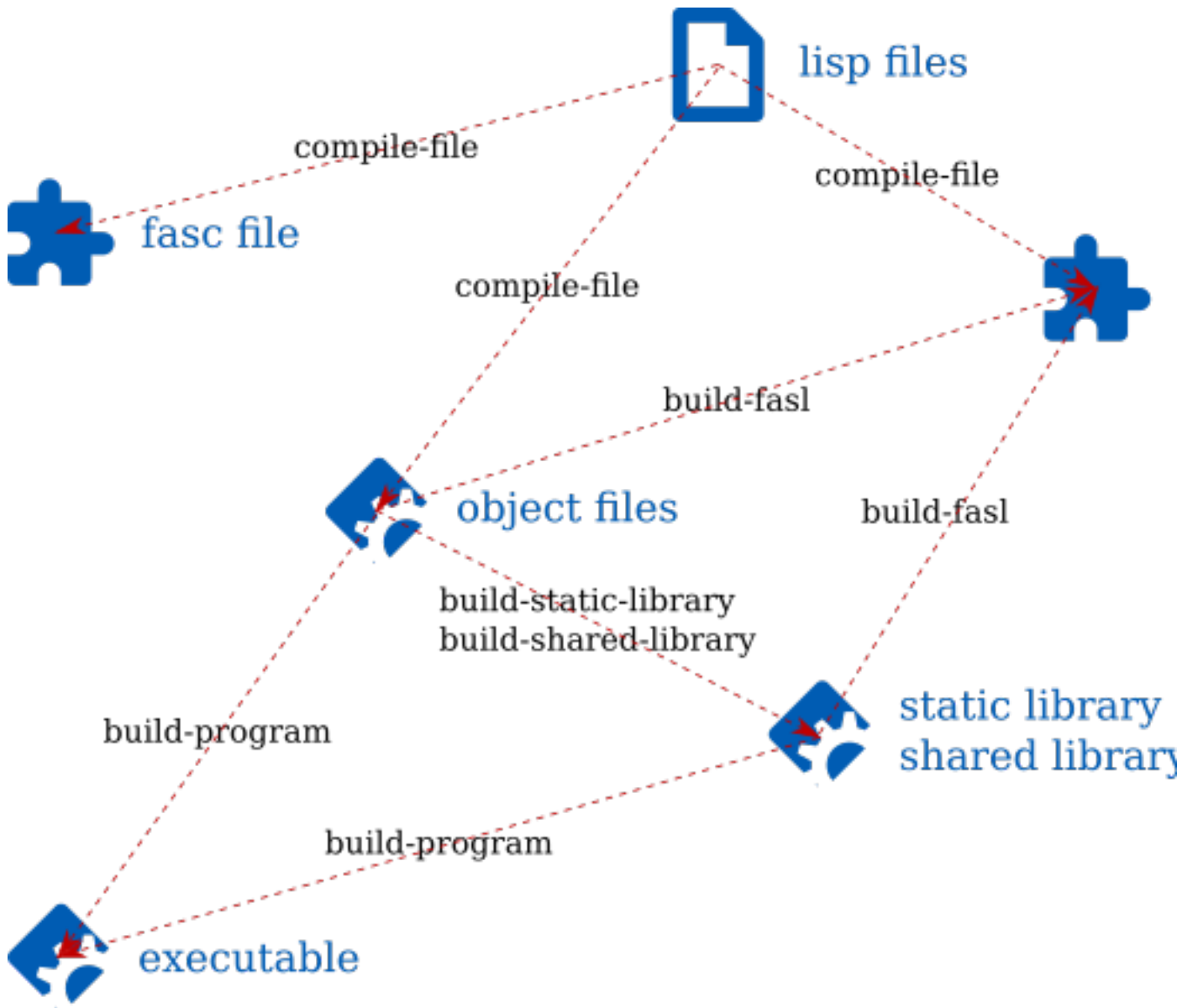


Figure 3.1: Build file types

3.1.1.1 Portable FASL

ECL provides two compilers (bytecodes compiler, and C/C++ compiler). Portable FASL files are built from source lisp files by the bytecodes compiler. Generally FASC files are portable across architectures and operating systems providing a convenient way of shipping portable modules. Portable FASL files may be concatenated, what leads to bundles. FASC files are faster to compile, but generally slower to run.

```
;; install bytecodes compiler
```

```

(ext:install-bytecodes-compiler)

;; compile hello.lisp file to hello.fasc
(compile-file "hello1.lisp")
(compile-file "hello2.lisp")

;; reinitialize C/C++ compiler back
(ext:install-c-compiler)

;; FASC file may be loaded dynamically from lisp program
(load "hello1.fasc")

;; ... concatenated into a bundle with other FASC
(with-open-file (output "hello.fasc"
                       :direction :output
                       :if-exists :supersede)
  (ext:run-program
    "cat" '("hello1.fasc" "hello2.fasc") :output output))

;; ... and loaded dynamically from lisp program
(load "hello.fasc")

```

3.1.1.2 Native FASL

If you want to make a library which is loaded dynamically from a lisp program, you should choose the fasl file format. Under the hood native fasls are just shared library files.

This means you can load fasl files with `dlopen` and initialize it by calling a `init` function from C programs, but this is not an intended usage. The recommended usage is to load fasl files by calling the `load` lisp function. To work with *Native FASL files* ECL has to be compiled with `--enable-shared` configure option (enabled by default).

Creating a fasl file from one lisp file is very easy.

```
(compile-file "hello.lisp")
```

To create a fasl file from more lisp files, firstly you have to compile each lisp file into an object file, and then combine them with `c:build-fasl`.

```

;; generates hello.o
(compile-file "hello.lisp" :system-p t)
;; generates goodbye.o
(compile-file "goodbye.lisp" :system-p t)

;; generates hello-goodbye.fas
(c:build-fasl "hello-goodbye"
  :lisp-files '("hello.o" "goodbye.o"))

;; fasls may be built from mix of objects and libraries (both shared and
;; static)
(c:build-fasl "mixed-bundle"
  :lisp-files '("hello1.o" "hello2.a" "hello3.so"))

```

3.1.1.3 Object file

Object files work as an intermediate file format. If you want to compile more than two lisp files, you might better to compile with a `:system-p t` option, which generates object files (instead of a fasl).

On linux systems, ECL invokes `gcc -c` to generate object files.

An object file consists of some functions in C:

- Functions corresponding to Lisp functions
- The initialization function which registers defined functions on the lisp environment

Consider the example below.

```
(defun say-hello ()
  (print "Hello, world"))
```

During compilation, this simple lisp program is translated into the C program, and then compiled into the object file. The C program contains two functions:

- `static cl_object L1say_hello: 'say-hello' function`
- `ECL_DLLEXPORT void _eclwm2nNauJEfEnD_CLSxi0z(cl_object flag): initialization function`

In order to use these object files from your C program, you have to call initialization functions before using lisp functions (such as `say-hello`). However the name of an init function is seemed to be randomized and not user-friendly. This is because object files are not intended to be used directly.

ECL provides other user-friendly ways to generate compiled lisp programs (as static/shared libraries or executables), and in each approach, object files act as intermediate files.

3.1.1.4 Static library

ECL can compile lisp programs to static libraries, which can be linked with C programs. A static library is created by `c:build-static-library` with some compiled object files.

```
;; generates hello.o
(compile-file "hello.lsp" :system-p t)
;; generates goodbye.o
(compile-file "goodbye.lsp" :system-p t)

;; generates libhello-goodbye.a
(c:build-static-library "hello-goodbye"
  :lisp-files '("hello.o" "goodbye.o")
  :init-name "init_hello_goodbye")
```

When you use a static/shared library, you have to call its init function. The name of this function is specified by the `:init-name` option. In this example, it is then `init_hello_goodbye`. The usage of this function is shown below:

```
#include <ecl/ecl.h>
extern void init_hello_goodbye(cl_object cblock);

int
main(int argc, char **argv)
```



```

{
    /* setup the lisp runtime */
    cl_boot(argc, argv);

    /* call the init function via ecl_init_module */
    ecl_init_module(NULL, init_hello_goodbye);

    /* ... */

    /* shutdown the lisp runtime */
    cl_shutdown();

    return 0;
}

```

Because the program itself does not know the type of the `init` function, a prototype declaration is inserted. After booting up the lisp environment, it invokes `init_hello_goodbye` via `ecl_init_module`. `init_hello_goodbye` takes an argument, and `ecl_init_module` supplies an appropriate one. Now that the initialization is finished, we can use functions and other stuff defined in the library.

DEPRECATED `read_VV` - equivalent to `ecl_init_module`

3.1.1.5 Shared library

Almost the same as with a static library. The user has to use `c:build-shared-library`:

```

;; generates hello.o
(compile-file "hello.lsp" :system-p t)
;; generates goodbye.o
(compile-file "goodbye.lsp" :system-p t)

;; generates libhello-goodbye.so
(c:build-shared-library "hello-goodbye"
    :lisp-files '("hello.o" "goodbye.o")
    :init-name "init_hello_goodbye")

```

3.1.1.6 Executable

ECL supports the generation of executable files. To create a standalone executable from a lisp program, compile all lisp files to object files. After that, calling `c:build-program` creates the executable:

```

;; generates hello.o
(compile-file "hello.lsp" :system-p t)
;; generates goodbye.o
(compile-file "goodbye.lsp" :system-p t)

;; generates hello-goodbye
(c:build-program "hello-goodbye"
    :lisp-files '("hello.o" "goodbye.o"))

```

Like with native FASL, the program may be built also from libraries.

3.1.1.7 Summary

In this section, some file types that can be compiled with ECL were introduced. Each file type has an adequate purpose:

- Object file: intermediate file format for others
- Fasl files: loaded dynamically via the `load` lisp function
- Static library: linked with and used from C programs
- Shared library: loaded dynamically and used from C programs
- Executable: standalone executable

ECL provides a high-level interface `c:build-*` for each native format. In case of *Portable FASL* the bytecodes compiler is needed.

3.1.2 Compiling with ASDF

First, let's disregard the simple situation in which we write Lisp without depending on any other Lisp libraries. A more practical example is to build a library that depends on other asdf (<https://common-lisp.net/project/asdf/>), systems. ECL provides a useful extension for asdf called `asdf:make-build`, which offers an abstraction for building libraries directly from system definitions.

To download dependencies you may use Quicklisp (<https://www.quicklisp.org>) to load your system (with dependencies defined). Make sure you can successfully load and run your library in the ECL REPL (or `*slime-repl*`). Don't worry about other libraries loaded in your image – ECL will only build and pack libraries your project depends on (that is, all dependencies you put in your `.asd` file, and their dependencies - nothing more, despite the fact that other libraries may be loaded).

3.1.2.1 Example code to build

We use a simple project that depends on `alexandria` to demonstrate the interface. The example consists of `example-with-dep.asd`, `package.lisp` and `example.lisp` (included in the `examples/asdf_with_dependence/` directory in the ECL source tree). Before any kind of build you need to push the full path of this directory to `asdf:*central-registry*` (or link it in a location already recognized by ASDF).

3.1.2.2 Build it as an single executable

Use this in the REPL to make an executable:

```
(asdf:make-build :example-with-dep
                 :type :program
                 :move-here #P"./"
                 :epilogue-code '(progn (example:test-function 5)
                                         (si:exit)))
```

Here the `:epilogue-code` is executed after loading our library; we can use arbitrary Lisp forms here. You can also put this code in your Lisp files and directly build them without this `:epilogue-code` option to achieve the same result. Running the program in a console will display the following and exit:

```
Factorial of 5 is: 120
```

3.1.2.3 Build it as shared library and use in C

Use this in the REPL to make a shared library:

```
(asdf:make-build :example-with-dep
                  :type :shared-library
                  :move-here #P"./"
                  :monolithic t
                  :init-name "init_example")
```

Here `:monolithic t` means that ECL will compile the library and all its dependencies into a single library named `example-with-dep--all-systems.so`. The `:move-here` parameter is self-explanatory. `:init-name` sets the name of the initialization function. Each library linked from C/C++ code must be initialized, and this is a mechanism to specify the initialization function's name.

To use it, we write a simple C program:

```
/* test.c */
#include <ecl/ecl.h>
extern void init_dll_example(cl_object);

int main (int argc, char **argv) {

    cl_boot(argc, argv);
    ecl_init_module(NULL, init_dll_example);

    /* do things with the Lisp library */
    cl_eval(c_string_to_object("(example:test-function 5)"));

    cl_shutdown();
    return 0;
}
```

Compile the file using a standard C compiler (note we're linking to `libecl.so` with `-lecl`, which provides the lisp runtime¹):

```
gcc test.c example-with-dep--all-systems.so -o test -lecl
```

If ECL is installed in a non-standard location you may need to provide flags for the compiler and the linker. You may read them with:

```
ecl-config --cflags
ecl-config --libs
```

Since our shared object is not in the standard location, you need to provide `LD_LIBRARY_PATH` pointing to the current directory to run the application:

```
LD_LIBRARY_PATH='pwd' ./test
```

This will show:

```
Factorial of 5 is: 120
```

¹ You may also link ECL runtime statically. That is not covered in this walkthrough.

You can also build all dependent libraries separately as a few `.so` files and link them together. For example, if you are building a library called `complex-example`, that depends on `alexandria` and `cl-fad`, you can do the following (in the REPL):

```
(asdf:make-build :complex-example
                  :type :shared-library
                  :move-here #P"./"
                  :init-name "init_example")

(asdf:make-build :alexandria
                  :type :shared-library
                  :move-here #P"./"
                  :init-name "init_alexandria")

(asdf:make-build :cl-fad
                  :type :shared-library
                  :move-here #P"./"
                  :init-name "init_fad")

(asdf:make-build :bordeaux-threads
                  :type :shared-library
                  :move-here #P"./"
                  :init-name "init_bt")
```

Note that we haven't specified `:monolithic t`, so we need to build `bordeaux-threads` as well because `cl-fad` depends on it. The building sequence doesn't matter and the resultant `.so` files can also be used in your future programs if these libraries are not modified.

We need to initialize all these modules using `ecl_init_module` in the correct order. (`bordeaux-threads` must be initialized before `cl-fad`; `cl-fad` and `alexandria` must be initialized before `complex-example`.)

Here is a code snippet (not a full program):

```
extern void init_fad(cl_object);
extern void init_alexandria(cl_object);
extern void init_bt(cl_object);
extern void init_example(cl_object);

/* call these *after* cl_boot(argc, argv);
   if B depends on A, you should first init A then B. */
ecl_init_module(NULL, init_bt);
ecl_init_module(NULL, init_fad);
ecl_init_module(NULL, init_alexandria);
ecl_init_module(NULL, init_example);
```

3.1.2.4 Build it as static library and use in C

To build a static library, use:

```
(asdf:make-build :example-with-dep
                  :type :static-library
```

```

:move-here #P"./"
:monolithic t
:init-name "init_example")

```

This will generate `example-with-dep--all-systems.a` in the current directory which we need to initialize with the `init_example` function. Compile it using:

```
gcc test.c example-with-dep--all-systems.a -o test-static -lecl
```

Then run it:

```
./test-static
```

This will show:

```
Factorial of 5 is: 120
```

Note we don't need to pass the current path in `LD_LIBRARY_PATH` here, since our Lisp library is statically bundled with the executable. The result is the same as the shared library example above. You can also build all dependent libraries separately as static libraries.

3.1.3 C compiler configuration

ECL provides some global variables to customize which C compiler and compiler options to use:

3.1.3.1 Compiler flags

It is not required to surround the compiler flags with quotes or use slashes before special characters.

string c:*user-cc-flags* [Variable]
Flags and options to be passed to the C compiler when building FASL, shared libraries and standalone programs.

string c:*user-ld-flags* [Variable]
Flags and options to be passed to the linker when building FASL, shared libraries and standalone programs.

string c:*cc-optimize* [Variable]
Optimize options to be passed to the C compiler.

3.1.3.2 Compiler & Linker programs

string c:*cc* [Variable]
This variable controls how the C compiler is invoked by ECL. One can set the variable appropriately adding for instance flags which the C compiler may need to exploit special hardware features (e.g. a floating point coprocessor).

string c:*ld* [Variable]
This variable controls the linker which is used by ECL.

string c:*ranlib* [Variable]
Name of the 'ranlib' program on the hosting platform.

string c:*ar* [Variable]
Name of the 'ar' program on the hosting platform.

string c::*ecl-include-directory* [Variable]
 Directory where the ECL header files for the target platform are located.

string c::*ecl-library-directory* [Variable]
 Directory where the ECL library files for the target platform are located.

3.2 Operating System Interface

3.2.1 Command line arguments

string ext:*help-message* [Variable]
 Command line help message. Initial value is ECL help message. This variable contains the help message which is output when ECL is invoked with the `--help`.

list-of-pathname-designators ext:*lisp-init-file-list* [Variable]
 ECL initialization files. Initial value is '("~/ecl" "~/eclrc"). This variable contains the names of initialization files that are loaded by ECL or embedding programs. The loading of initialization files happens automatically in ECL unless invoked with the option `--norc`. Whether initialization files are loaded or not is controlled by the command line options rules, as described in `ext:process-command-args`.

list-of-lists ext:+default-command-arg-rules+ [Variable]
 ECL command line options. This constant contains a list of rules for parsing the command line arguments. This list is made of all the options which ECL accepts by default. It can be passed as first argument to `ext:process-command-args`, and you can use it as a starting point to extend ECL.

ext:command-args [Function]
 Original list of command line arguments. This function returns the list of command line arguments passed to either ECL or the program it was embedded in. The output is a list of strings and it corresponds to the `argv` vector in a C program. Typically, the first argument is the name of the program as it was invoked. You should not count on the filename to be resolved.

ext:process-command-args &key args rules [Function]
args A list of strings. Defaults to the output of `ext:command-args`.
rules A list of lists. Defaults to the value of `ext:+default-command-arg-rules+`.

This function processes the command line arguments passed to either ECL or the program that embeds it. It uses the list of rules `rules`, which has the following syntax:

(option-name nargs template [:stop | :noloadrc | :loadrc]*)

option-name
 A string with the option prefix as typed by the user. For instance `--help`, `-?`, `--compile`, etc.

nargs
 A non-negative integer denoting the number of arguments taken by this option.

template A lisp form, not evaluated, where numbers from 0 to nargs will be replaced by the corresponding option argument.

:stop If present, parsing of arguments stops after this option is found and processed. The list of remaining arguments is passed to the rule. ECL's top-level uses this option with the `--` command line option to set `ext:*unprocessed-ecl-command-args*` to the list of remaining arguments.

:noloadrc, :loadrc
Determine whether the lisp initialization files in `ext:*lisp-init-file-list*` will be loaded before processing all forms.

`ext:process-command-args` works as follows. First of all, it parses all the command line arguments, except for the first one, which is assumed to contain the program name. Each of these arguments is matched against the rules, sequentially, until one of the patterns succeeds.

A special name `*default*`, matches any unknown command line option. If there is no `*default*` rule and no match is found, an error is signaled. For each rule that succeeds, the function constructs a lisp statement using the template.

After all arguments have been processed, `ext:process-command-args`, and there were no occurrences of `:noloadrc`, the first existing file listed in `ext:*lisp-init-file-list*` will be loaded. Finally, the list of lisp statements will be evaluated.

The following piece of code implements the `ls` command using lisp. Instructions for building this program are found under `examples/cmdline/ls.lsp`.

```
(setq ext:*help-message* "
ls [--help | -?] filename*
    Lists the file that match the given patterns.
")

(defun print-directory (pathnames)
  (format t "~{~A~%~}"
    (mapcar #'(lambda (x) (enough-namestring x (si::getcwd)))
      (mapcan #'directory (or pathnames '(*.*" */"))))))

(defconstant +ls-rules+
  '("--help" 0 (progn (princ ext:*help-message* *standard-output*) (ext:quit 0)))
  ("-?" 0 (progn (princ ext:*help-message* *standard-output*) (ext:quit 0)))
  ("*DEFAULT*" 1 (print-directory 1) :stop)))

(let ((ext:*lisp-init-file-list* NIL)) ; No initialization files
  (handler-case (ext:process-command-args :rules +ls-rules+)
    (error (c)
      (princ ext:*help-message* *error-output*)
      (ext:quit 1))))
(ext:quit 0)
```

3.2.2 External processes

ECL provides several facilities for invoking and communicating with external processes. If one just wishes to execute some program, without caring for its output, then probably `ext:system` is the best function. In all other cases it is preferable to use `ext:run-program`, which opens pipes to communicate with the program and manipulate it while it runs on the background.

External process is a structure created with `ext:run-program` (returned as third value). It is programmer responsibility, to call `ext:external-process-wait` on finished processes, however during garbage collection object will be finalized.

`ext:external-process-pid process` [Function]
Returns process PID or `nil` if already finished.

`ext:external-process-status process` [Function]
Updates process status. `ext:external-process-status` calls `ext:external-process-wait` if process has not finished yet (non-blocking call). Returns two values:
`status` - member of `(:abort :error :exited :signalled :stopped :resumed :running)`
`code` - if process exited it is a returned value, if terminated it is a signal code. Otherwise `NIL`.

`ext:external-process-wait process wait` [Function]
If the second argument is non-`NIL`, function blocks until external process is finished. Otherwise status is updated. Returns two values (see `ext:external-process-status`).

`ext:terminate-process process &optional force` [Function]
Terminates external process.

`ext:external-process-input process` [Function]
`ext:external-process-output process` [Function]
`ext:external-process-error-stream process` [Function]
Process stream accessors (read-only).

`ext:run-program command argv &key input output error wait` [Function]
environ if-input-does-not-exist if-output-exists if-error-exists
external-format #+windows escape-arguments
`ext:run-program` creates a new process specified by the *command* argument. *argv* are the standard arguments that can be passed to a program. For no arguments, use `nil` (which means that just the name of the program is passed as arg 0).
`ext:run-program` will return three values - two-way stream for communication, return code or `nil` (if process is called asynchronously), and `ext:external-process` object holding process state.

It is programmer responsibility to call `ext:external-process-wait` on finished process, however ECL associates Section 3.6.5 [Finalization], page 140, with the object calling it when the object is garbage collected. If process didn't finish but is not referenced, finalizer will be invoked once more during next garbage collection.

The **&key** arguments have the following meanings:

input Either **t**, **nil**, a pathname, a string, a stream or **:stream**. If **t** the standard input for the current process is inherited. If **nil**, **/dev/null** is used. If a pathname (or a string), the file so specified is used. If a stream, all the input is read from that stream and sent to the subprocess. If **:stream**, the **ext:external-process-input** slot is filled in with a stream that sends its output to the process. Defaults to **:stream**.

if-input-does-not-exist

Can be one of: **:error** to generate an error **:create** to create an empty file **nil** (the default) to return nil from **ext:run-program**

output Either **t**, **nil**, a pathname, a string, a stream, or **:stream**. If **t**, the standard output for the current process is inherited. If **nil**, **/dev/null** is used. If a pathname (or as string), the file so specified is used. If a stream, all the output from the process is written to this stream. If **:stream**, the **ext:external-process-output** slot is filled in with a stream that can be read to get the output. Defaults to **:stream**.

if-output-exists

Can be one of: **:error** (the default) to generate an error, **:supersede** to supersede the file with output from the program, **:append** to append output from the program to the file or **nil** to return nil from **ext:run-program**.

error Same as **:output**, except that **:error** can also be specified as **:output** in which case all error output is routed to the same place as normal output. Defaults to **:output**.

if-error-exists

Same as **:if-output-exists**.

wait If non-**nil** (default), wait until the created process finishes. If **nil**, continue running Lisp until the program finishes.

environ A list of STRINGS describing the new Unix environment (as in "man environ"). The default is to copy the environment of the current process. To extend existing environment (instead of replacing it), use **:environ** (**append *my-env* (ext:environ)**).

If non-**nil** **environ** argument is supplied, then first argument to **ext:run-program**, **command**, must be full path to the file.

external-format

The external-format to use for **:input**, **:output**, and **:error** STREAMs.

Windows specific options:

escape-arguments

Controls escaping of the arguments passed to **CreateProcess**.

3.2.3 FIFO files (named pipes)

Named pipe (known as fifo) may be created on UNIX with a shell command `mkfifo`. ECL opens such files in non-blocking mode. `ext:file-kind` will return for such file `:fifo`. Since it is impossible to guess how many characters are available in this special file `file-length` function will return NIL.

3.2.4 Operating System Interface Reference

`ext:system command` [Function]
Run shell command ignoring its output. Uses fork.

`ext:make-pipe` [Function]
Creates a pipe and wraps it in a two way stream.

`ext:quit &optional exit-code kill-all-threads` [Function]
This function abruptly stops the execution of the program in which ECL is embedded. Depending on the platform, several other functions will be invoked to free resources, close loaded modules, etc.
The exit code is the code seen by the parent process that invoked this program. Normally a code other than zero denotes an error.
If *kill-all-threads* is non-nil, tries to gently kill and join with running threads.

`ext:environ` [Function]
`ext:getenv variable` [Function]
`ext:setenv variable value` [Function]
Environment accessors.

`ext:getpid` [Function]
`ext:getuid` [Function]
`ext:getcwd &optional (change-default-pathname-defaults NIL)` [Function]
`ext:chdir directory &optional (change-default-pathname-defaults T)` [Function]
`ext:file-kind filename follow-symlinks-p` [Function]
`ext:copy-file filename destination-filename` [Function]
`ext:chmod filename mode` [Function]
Common operating system functions.

3.3 Foreign Function Interface

3.3.1 What is a FFI?

A Foreign Function Interface, or FFI for short, is a means for a programming language to interface with libraries written in other programming languages, the foreign code. You will see this concept most often being used in interpreted environments, such as Python, Ruby or Lisp, where one wants to reuse the big number of libraries written in C and C++ for dealing with graphical interfaces, networking, filesystems, etc.

A FFI is made of at least three components:

Foreign objects management

This is the data that the foreign code will use. A FFI needs to provide means to build and manipulate foreign data, with automatic conversions to and from lisp

data types whenever possible, and it also has to deal with issues like garbage collection and finalization.

Foreign code loader

To actually use a foreign routine, the code must reside in memory. The process of loading this code and finding out the addresses of the routines we want to use is normally done by an independent component.

Foreign function invocation

This is the part of the FFI that deals with actually calling the foreign routines we want to use. For that one typically has to tell the FFI what are the arguments that these routines expect, what are the calling conventions and where are these routines to be found.

On top of these components sits a higher level interface written entirely in lisp, with which you will actually declare and use foreign variables, functions and libraries. In the following sections we describe both the details of the low-level components (See Section 3.3.2 [Two kinds of FFI], page 99, and Section 3.3.3 [Foreign objects], page 100), and of the higher level interface (See Section 3.3.4 [Higher level interfaces], page 101). It is highly recommended that you read all sections.

3.3.2 Two kinds of FFI

ECL allows for two different approaches when building a FFI. Both approaches have a different implementation philosophy and affect the places where you can use the FFI and how.

Static FFI (SFFI)

For every foreign function and variable you might need to use, a wrapper is automatically written in C with the help of `ffi:c-inline`. These wrappers are compiled using an ordinary C compiler and linked against both the foreign libraries you want to use and against the ECL library. The result is a FASL file that can be loaded from ECL and where the wrappers appear as ordinary lisp functions and variables that the user may directly invoke.

Dynamic FFI (DFFI)

First of all, the foreign libraries are loaded in memory using the facilities of the operating system. Similar routines are used to find out and register the memory location of all the functions and variables we want to use. Finally, when actually accessing these functions, a little piece of assembly code does the job of translating the lisp data into foreign objects, storing the arguments in the stack and in CPU registers, calling the function and converting back the output of the function to lisp.

ECL for this purpose utilizes *libffi* (<https://sourceware.org/libffi/>), a portable foreign-function interface library.

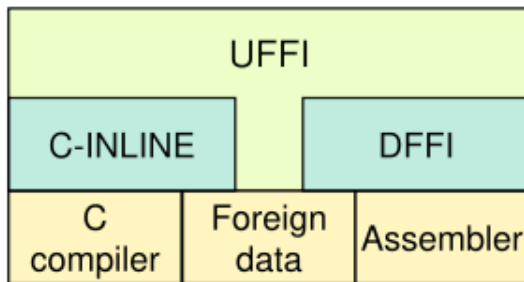


Figure 3.2: FFI components

As you see, the first approach uses rather portable techniques based on a programming language (C, C++) which is strongly supported by the operating system. The conversion of data is performed by a calling routines in the ECL library and we need not care about the precise details (organizing the stack, CPU registers, etc) when calling a function: the compiler does this for us.

On the other hand, the dynamic approach allows us to choose the libraries we load at any time, look for the functions and invoke them even from the toplevel, but it relies on unportable techniques and requires the developers to know very well both the assembly code of the machine the code runs on and the calling conventions of that particular operating system. For these reasons ECL doesn't maintain it's own implementation of the DFFI but rather relies on the third party library.

ECL currently supports the static method on all platforms, and the dynamical one a wide range of the most popular ones, shown in the section *Supported Platforms* at <https://sourceware.org/libffi/>.

You can test if your copy of ECL was built with DFFI by inspecting whether the symbol `:dffi` is present in the list from variable `*features*`.

3.3.3 Foreign objects

While the foreign function invocation protocols differ strongly between platforms and implementations, foreign objects are pretty easy to handle portably. For ECL, a foreign object is just a bunch of bytes stored in memory. The lisp object for a foreign object encapsulates several bits of information:

- A list or a symbol specifying the C type of the object.
- The pointer to the region of memory where data is stored.
- A flag determining whether ECL can automatically manage that piece of memory and deallocated when no longer in use.

A foreign object may contain many different kinds of data: integers, floating point numbers, C structures, unions, etc. The actual type of the object is stored in a list or a symbol which is understood by the higher level interface (See Section 3.3.4 [Higher level interfaces], page 101).

The most important component of the object is the memory region where data is stored. By default ECL assumes that the user will perform manual management of this memory,

deleting the object when it is no longer needed. The first reason is that this block may have been allocated by a foreign routine using `malloc()`, or `mmap()`, or statically, by referring to a C constant. The second reason is that foreign functions may store references to this memory which ECL is not aware of and, in order to keep these references valid, ECL should not attempt to automatically destroy the object.

In many cases, however, it is desirable to automatically destroy foreign objects once they have been used. The higher level interfaces UFFI and CFFI (<https://common-lisp.net/project/cffi/>) provide tools for doing this. For instance, in the following example adapted from the UFFI documentation, the string *name* is automatically deallocated

```
(ffi:def-function ("gethostname" c-gethostname)
  ((name (* :unsigned-char))
   (len :int))
  :returning :int)

(ffi:with-foreign-object (name '(:array :unsigned-char 256))
  (if (zerop (c-gethostname (ffi:char-array-to-pointer name) 256))
      (format t "Hostname: ~S" (ffi:convert-from-foreign-string name))
      (error "gethostname() failed.")))
```

3.3.4 Higher level interfaces

Up to now we have only discussed vague ideas about how a FFI works, but you are probably more interested on how to actually code all these things in lisp. You have here three possibilities:

- ECL supplies a high level interface which is compatible with UFFI up to version 1.8 (api for $\geq v2.0$ is provided by `cffi-uffi-compat` system shipped with CFFI). Code designed for UFFI library should run mostly unchanged with ECL. Note, that api resides in `ffi` package, not `uffi`, to prevent conflicts with `cffi-uffi-compat`. New code shouldn't use this interface preferring CFFI (<https://common-lisp.net/project/cffi/>).
- The CFFI (<https://common-lisp.net/project/cffi/>) library features a complete backend for ECL. This method of interfacing with the foreign libraries is preferred over using UFFI.
- ECL's own low level interface. Only to be used if ECL is your deployment platform. It features some powerful constructs that allow you to mix arbitrary C and lisp code.

In the following two subsections we will discuss two practical examples of using the native UFFI and the CFFI library.

UFFI example

The example below shows how to use UFFI in an application. There are several important ingredients:

- You need to specify the libraries you use and do it at the toplevel, so that the compiler may include them at link time.
- Every function you will use has to be declared using `ffi:def-function`.

```
#!
Build and load this module with (compile-file "uffi.lsp" :load t)
```

```

|#
;;
;; This toplevel statement notifies the compiler that we will
;; need this shared library at runtime. We do not need this
;; statement in windows.
;;
#-(or ming32 windows)
(ffi:load-foreign-library #+darwin "/usr/lib/libm.dylib"
  #-darwin "/usr/lib/libm.so")
;;
;; With this other statement, we import the C function sin(),
;; which operates on IEEE doubles.
;;
(ffi:def-function ("sin" c-sin) ((arg :double))
  :returning :double)
;;
;; We now use this function and compare with the lisp version.
;;
(format t "~%Lisp sin:~t~d~%C sin:~t~d~%Difference:~t~d"
  (sin 1.0d0) (c-sin 1.0d0) (- (sin 1.0d0) (c-sin 1.0d0)))

```

CFFI example

The CFFI (<https://common-lisp.net/project/cffi/>) library is an independent project and it is not shipped with ECL. If you wish to use it you can go to their homepage, download the code and build it using ASDF.

CFFI differs slightly from UFFI in that functions may be used even without being declared beforehand.

```

#|
Build and load this module with (compile-file "cffi.lisp" :load t)
|#
;;
;; This toplevel statement notifies the compiler that we will
;; need this shared library at runtime. We do not need this
;; statement in windows.
;;
#-(or ming32 windows)
(cffi:load-foreign-library #+darwin "/usr/lib/libm.dylib"
  #-darwin "/usr/lib/libm.so")
;;
;; With this other statement, we import the C function sin(),
;; which operates on IEEE doubles.
;;
(cffi:defcfun ("sin" c-sin) :double '(:double))
;;
;; We now use this function and compare with the lisp version.
;;

```

```
(format t "~%Lisp sin:~t~d~%C sin:~t~d~%Difference:~t~d"
(sin 1.0d0) (c-sin 1.0d0) (- (sin 1.0d0) (c-sin 1.0d0)))
;;
;; The following also works: no declaration!
;;
(let ((c-cos (cffi:foreign-funcall "cos" :double 1.0d0 :double)))
  (format t "~%Lisp cos:~t~d~%C cos:~t~d~%Difference:~t~d"
(cos 1.0d0) c-cos (- (cos 1.0d0) c-cos)))
```

SFFI example (low level inlining)

To compare with the previous pieces of code, we show how the previous programs would be written using `ffi:c-lines` and `ffi:c-inline`.

```
#|
Build and load this module with (compile-file "ecl.lisp" :load t)
|#
;;
;; With this other statement, we import the C function sin(), which
;; operates on IEEE doubles. Notice that we include the C header to
;; get the full declaration.
;;
(defun c-sin (x)
  (ffi:c-lines "#include <math.h>")
  (ffi:c-inline (x) (:double) :double "sin(#0)" :one-liner t))
;;
;; We now use this function and compare with the lisp version.
;;
(format t "~%Lisp sin:~t~d~%C sin:~t~d~%Difference:~t~d"
(sin 1.0d0) (c-sin 1.0d0) (- (sin 1.0d0) (c-sin 1.0d0)))
```

3.3.5 SFFI Reference

Reference

`ffi:c-lines` *c/c++-code** [Special Form]

Insert C declarations and definitions

c/c++-code

One or more strings with C definitions. Not evaluated.

returns

No value.

Description This special form inserts C code from strings passed in the *arguments* directly in the file that results from compiling lisp sources. Contrary to `ffi:c-inline`, this function may have no executable statements, accepts no input value and returns no value.

The main use of `ffi:c-lines` is to declare or define C variables and functions that are going to be used later in other FFI statements. All statements from *arguments* are grouped at the beginning of the produced header file.

ffi:clines is a special form that can only be used in lisp compiled files as a toplevel form. Other uses will lead to an error being signaled, either at the compilation time or when loading the file.

Examples In this example the **ffi:clines** statement is required to get access to the C function **cos**:

```
(ffi:clines "#include <math.h>")
(defun cos (x)
  (ffi:c-inline (x) (:double) :double "cos(#0)" :one-liner t))
```

ffi:c-inline (*lisp-values*) (*arg-c-types*) *return-type* *c/c++-code* [Special Form]
&key (*side-effects* *t*) (*one-liner* *nil*)

Inline C code in a lisp form

lisp-values One or more lisp expressions. Evaluated.

arg-c-types

One or more valid FFI types. Evaluated.

return-type

Valid FFI type or (**values** *ffi-type**).

c/c++-code

String containing valid C code plus some valid escape forms.

one-liner

Boolean indicating, if the expression is a valid R-value. Defaults to **nil**.

side-effects

Boolean indicating, if the expression causes side effects. Defaults to **t**.

returns

One or more lisp values.

Description This is a special form which can be only used in compiled code and whose purpose is to execute some C code getting and returning values from and to the lisp environment.

The first argument *lisp-values* is a list of lisp forms. These forms are going to be evaluated and their lisp values will be transformed to the corresponding C types denoted by the elements in the list *arg-c-types*.

The input values are used to create a valid C expression using the template in *C/C++-code*. This is a string of arbitrary size which mixes C expressions with two kind of escape forms.

The first kind of escape form are made of a hash and a letter or a number, as in: **#0**, **#1**, ..., until **#z**. These codes are replaced by the corresponding input values. The second kind of escape form has the format **@(return [n])**, it can be used as lvalue in a C expression and it is used to set the n-th output value of the **ffi:c-inline** form.

When the parameter *one-liner* is true, then the C template must be a simple C statement that outputs a value. In this case the use of **@(return)** is not allowed. When the parameter *one-liner* is false, then the C template may be a more complicated block form, with braces, conditionals, loops and spanning multiple lines. In this case the output of the form can only be set using **@(return)**.

Parameter *side-effects* set to false will indicate, that the functions causes no side-effects. This information is used by the compiler to optimize the resulting code. If

side-effects is set to false, but the function may cause the side effects, then results are undefined.

Note that the conversion between lisp arguments and FFI types is automatic. Note also that `ffi:c-inline` cannot be used in interpreted or bytecompiled code! Such usage will signal an error.

Examples The following example implements the transcendental function SIN using the C equivalent:

```
(ffi:c-lines "#include <math.h>")
(defun mysin (x)
  (ffi:c-inline (x) (:double) :double
    "sin(#0)"
    :one-liner t
    :side-effects nil))
```

This function can also be implemented using the `@(return)` form as follows:

```
((defun mysin (x)
  (ffi:c-inline (x) (:double) :double
    "@(return)=sin(#0);"
    :side-effects nil))
```

The following example is slightly more complicated as it involves loops and two output values:

```
((defun sample (x)
  (ffi:c-inline (x (+ x 2)) (:int :int) (values :int :int) "{
    int n1 = #0, n2 = #1, out1 = 0, out2 = 1;
    while (n1 <= n2) {
      out1 += n1;
      out2 *= n1;
      n1++;
    }
    @(return 0)= out1;
    @(return 1)= out2;
  }"
    :side-effects nil))
```

ffi:c-progn *args* **&body** *body* [Special Form]

Interleave C statements with the Lisp code

args Lisp arguments. Evaluated.

returns No value.

Description This form is used for its side effects. It allows for interleaving C statements with the Lisp code. The argument types doesn't have to be declared – in such case the objects type in the C world will be `cl_object`.

Examples

```
(lambda (i)
  (let* ((limit i)
        (iterator 0)
```

```

      (custom-var (cons 1 2)))
(declare (:int limit iterator))
(ffl:c-progn (limit iterator custom-var)
  "cl_object cv = #2;"
  "ecl_print(cv, ECL_T);"
  "for (#1 = 0; #1 < #0; #1++) {"
  (format t "~&Iterator: ~A, I: ~A~%" iterator i)
  "}")

```

ffi:defcallback *name ret-type arg-desc &body body* [Special Form]

name Name of the lisp function.

ret-type Declaration of the return type which function returns.

arg-desc List of pairs (*arg-name arg-type*).

body Function body.

returns Pointer to the defined callback.

Description Defines Lisp function and generates a callback for the C world, which may be passed to these functions. Note, that this special operator has also a dynamic variant (with the same name and interface).

ffi:defcbody *name arg-types result-type c-expression* [Macro]

Define C function under the lisp name

name Defined function name.

arg-types Argument types of the defined Lisp function.

result-type Result type of the C function (may be (*values ...*)).

returns Defined function name.

Description The compiler defines a Lisp function named by *name* whose body consists of the C code of the string *c-expression*. In the *c-expression* one can reference the arguments of the function as #0, #1, etc.

The interpreter ignores this form.

ffi:defentry *name arg-types c-name &key no-interrupts* [Macro]

name Lisp name for the function.

arg-types Argument types of the C function.

c-name If *c-name* is a list, then C function result type is declared as (*car c-name*) and its name is (*string (cdr c-name)*).
If it's an atom, then the result type is :object, and function name is (*string c-name*).

returns Lisp function *name*.

Description The compiler defines a Lisp function named by *name* whose body consists of a calling sequence to the C language function named by *c-name*.

The interpreter ignores this form.

ext:with-backend *&key* *bytecodes* *c/c++* [Special Form]

Use different code depending on the backend.

Description Depending on whether the bytecodes or C compiler is used, this form will emit the code given in the corresponding keyword argument.

Examples

```
CL-USER> (defmacro test ()
           '(ext:with-backend :c/c++ "c/c++" :bytecodes "bytecodes"))
TEST
CL-USER> (test)
"bytecodes"
CL-USER> (funcall (compile nil (lambda () (test))))

;;; OPTIMIZE levels: Safety=2, Space=0, Speed=3, Debug=3
"c/c++"
```

ffi:defla *name* *args* **&body** *body* [Macro]

Provide Lisp alternative for interpreted code.

Description Used to DEFINE Lisp Alternative. For the interpreter, **ffi:defla** is equivalent to **defun**, but the compiler ignores this form.

3.3.6 DFFI Reference

ffi:*use-dffi* [Variable]

This variable controls whether DFFI is used or not.

3.3.7 UFFI Reference

3.3.7.1 Primitive Types

Primitive types have a single value, these include characters, numbers, and pointers. They are all symbols in the keyword package.

':char'

':unsigned-char'

Signed/unsigned 8-bits. Dereferenced pointer returns a character.

':byte'

':unsigned-byte'

Signed/unsigned 8-bits. Dereferenced pointer returns an integer.

':short'

':unsigned-short'

':int'

':unsigned-int'

':long'

':unsigned-long'

Standard integer types (16-bit, 32-bit and 32/64-bit).

`‘:int16_t’`
`‘:uint16_t’`
`‘:int32_t’`
`‘:uint32_t’`
`‘:int64_t’`
`‘:uint64_t’`
 Integer types with guaranteed bitness.

`‘:float’`
`‘:double’` Floating point numerals (32-bit and 64-bit).

`‘:long-double’`
 Floating point numeral (usually 80-bit, at least 64-bit, exact bitness is compiler/architecture/platform dependent).

`‘:csfloat’`
`‘:cdfloat’`
`‘:clfloat’`
 Complex floating point numerals. These types exist only when ECL is built with c99complex support.

`‘:cstring’`
 A NULL terminated string used for passing and returning characters strings with a C function.

`‘:void’` The absence of a value. Used to indicate that a function does not return a value.

`‘:pointer-void’`
 Points to a generic object.

`‘*’` Used to declare a pointer to an object.

`‘:object’` A generic lisp object (i.e. a `cl_object` in C)

Reference

ffi:def-constant *name value &key* (*export nil*) [Macro]

Binds a symbol to a constant.

name A symbol that will be bound to the value.

value An evaluated form that is bound the the name.

export When `t`, the name is exported from the current package. Defaults to `nil`.

returns Constant name.

Description This is a thin wrapper around `defconstant`. It evaluates at compile-time and optionally exports the symbol from the package.

Examples

```
(ffi:def-constant pi2 (* 2 pi))
```

```
(ffi:def-constant exported-pi2 (* 2 pi) :export t)
```

Side Effects Creates a new special variable.

ffi:def-foreign-type *name definition* [Macro]

Defines a new foreign type

name A symbol naming the new foreign type.

value A form that is not evaluated that defines the new foreign type.

returns Foreign type designator (*value*).

Description Defines a new foreign type

Examples

```
(ffi:def-foreign-type my-generic-pointer :pointer-void)
(ffi:def-foreign-type a-double-float :double-float)
(ffi:def-foreign-type char-ptr (* :char))
```

Side effects Defines a new foreign type.

ffi:null-char-p *char* [Function]

Tests a character for NULL value

char A character or integer.

returns A boolean flag indicating if *char* is a NULL value.

Description A predicate testing if a character or integer is NULL. This abstracts the difference in implementations where some return a character and some return a integer whence dereferencing a C character pointer.

Examples

```
(ffi:def-array-pointer ca :unsigned-char)
(let ((fs (ffi:convert-to-foreign-string "ab")))
  (values (ffi:null-char-p (ffi:deref-array fs 'ca 0))
          (ffi:null-char-p (ffi:deref-array fs 'ca 2))))
;; => NIL T
```

3.3.7.2 Aggregate Types

Overview

Aggregate types are comprised of one or more primitive types.

Reference

ffi:def-enum *name fields &key separator-string* [Macro]

Defines a C enumeration

name A symbol that names the enumeration.

fields A list of field definitions. Each definition can be a symbol or a list of two elements. Symbols get assigned a value of the current counter which starts at 0 and increments by 1 for each subsequent symbol. If the field definition is a list, the first position is the symbol and the second position is the value to assign to the symbol. The current counter gets set to 1+ this value.

returns A string that governs the creation of constants. The default is "#".

Description Declares a C enumeration. It generates constants with integer values for the elements of the enumeration. The symbols for these constant values are created by the concatenation of the enumeration name, separator-string, and field symbol. Also creates a foreign type with the name *name* of type `:int`.

Examples

```
(ffi:def-enum abc (:a :b :c))
;; Creates constants abc#a (1), abc#b (2), abc#c (3) and defines
;; the foreign type "abc" to be :int

(ffi:def-enum efoo (:e1 (:e2 10) :e3) :separator-string "-")
;; Creates constants efoo-e1 (1), efoo-e2 (10), efoo-e3 (11) and defines
;; the foreign type efoo to be :int
```

Side effects Creates a `:int` foreign type, defines constants.

ffi:def-struct *name* &*rest fields* [Macro]
 Defines a C structure

name A symbol that names the structure.

fields A variable number of field definitions. Each definition is a list consisting of a symbol naming the field followed by its foreign type.

Description Declares a structure. A special type is available as a slot in the field. It is a pointer that points to an instance of the parent structure. Its type is `:pointer-self`.

Examples

```
(ffi:def-struct foo (a :unsigned-int)
  (b (* :char))
  (c (:array :int 10))
  (next :pointer-self))
```

Side effects Creates a foreign type.

ffi:get-slot-value *obj type field* [Function]
 Retrieves a value from a slot of a structure

obj A pointer to the foreign structure.

type The name of the foreign structure.

field The name of the desired field in the foreign structure.

returns The value of the *field* in the structure *obj*.

Description Accesses a slot value from a structure. This is generalized and can be used with `setf`.

Examples

```
(ffi:get-slot-value foo-ptr 'foo-structure 'field-name)
(setf (ffi:get-slot-value foo-ptr 'foo-structure 'field-name) 10)
```

ffi:get-slot-pointer *obj type field* [Function]

Retrieves a pointer from a slot of a structure

obj A pointer to the foreign structure.

type The name of the foreign structure.

field The name of the desired field in the foreign structure.

returns The value of the pointer *field* in the structure *obj*.

Description This is similar to `ffi:get-slot-value`. It is used when the value of a slot is a pointer type.

Examples

```
(ffi:get-slot-pointer foo-ptr 'foo-structure 'my-char-ptr)
```

ffi:def-array-pointer *name type* [Macro]

Defines a pointer to an array of *type*

name A name of the new foreign type.

type The foreign type of the array elements.

Description Defines a type that is a pointer to an array of *type*.

Examples

```
(ffi:def-array-pointer byte-array-pointer :unsigned-char)
```

Side effects Defines a new foreign type.

ffi:deref-array *array type position* [Function]

Dereference an array

array A foreign array.

type The foreign type of the *array*.

position An integer specifying the position to retrieve from the *array*.

returns The value stored in the *position* of the *array*.

Description Dereferences (retrieves) the value of the foreign array element. `setf`-able.

Examples

```
(ffi:def-array-pointer ca :char)
  (let ((fs (ffi:convert-to-foreign-string "ab")))
    (values (ffi:null-char-p (ffi:deref-array fs 'ca 0))
            (ffi:null-char-p (ffi:deref-array fs 'ca 2))))
;; => NIL T
```

ffi:def-union *name &rest fields* [Macro]

Defines a foreign union type

name A name of the new union type.

fields A list of fields of the union in form (field-name field-type).

Description Defines a foreign union type.

Examples

```
(ffi:def-union test-union
  (a-char :char)
  (an-int :int))

(let ((u (ffi:allocate-foreign-object 'test-union)))
  (setf (ffi:get-slot-value u 'test-union 'an-int) (+ 65 (* 66 256))))
(prog1
  (ffi:ensure-char-character (ffi:get-slot-value u 'test-union 'a-char))
  (ffi:free-foreign-object u)))
;; => #\A
```

Side effects Defines a new foreign type.

3.3.7.3 Foreign Objects

Overview

Objects are entities that can be allocated, referred to by pointers, and can be freed.

Reference

ffi:allocate-foreign-object *type* &**optional** *size* [Function]

Allocates an instance of a foreign object

type The type of foreign object to allocate. This parameter is evaluated.

size An optional size parameter that is evaluated. If specified, allocates and returns an array of *type* that is *size* members long. This parameter is evaluated.

returns A pointer to the foreign object.

Description Allocates an instance of a foreign object. It returns a pointer to the object.

Examples

```
(ffi:def-struct ab (a :int) (b :double))
;; => (:STRUCT (A :INT) (B :DOUBLE))
(ffi:allocate-foreign-object 'ab)
;; => #<foreign AB>
```

ffi:free-foreign-object *ptr* [Function]

Frees memory that was allocated for a foreign object

ptr A pointer to the allocated foreign object to free.

Description Frees memory that was allocated for a foreign object.

ffi:with-foreign-object (*var type*) &**body** *body* [Macro]

Wraps the allocation, binding and destruction of a foreign object around a body of code

var Variable name to bind.

type Type of foreign object to allocate. This parameter is evaluated.
body Code to be evaluated.
returns The result of evaluating the body.

Description This function wraps the allocation, binding, and destruction of a foreign object around the body of code.

Examples

```
(defun gethostname2 ()
  "Returns the hostname"
  (ffi:with-foreign-object (name '(:array :unsigned-char 256))
    (if (zerop (c-gethostname (ffi:char-array-to-pointer name) 256))
      (ffi:convert-from-foreign-string name)
      (error "gethostname() failed."))))
```

ffi:size-of-foreign-type *ftype* [Macro]

Returns the number of data bytes used by a foreign object type

ftype A foreign type specifier. This parameter is evaluated.
returns Number of data bytes used by a foreign object *ftype*.

Description Returns the number of data bytes used by a foreign object type. This does not include any Lisp storage overhead.

Examples

```
(ffi:size-of-foreign-type :unsigned-byte)
;; => 1
(ffi:size-of-foreign-type 'my-100-byte-vector-type)
;; => 100
```

ffi:pointer-address *ptr* [Function]

Returns the address of a pointer

ptr A pointer to a foreign object.
returns An integer representing the pointer's address.

Description Returns the address as an integer of a pointer.

ffi:deref-pointer *ptr ftype* [Function]

Dereferences a pointer

ptr Pointer to a foreign object.
ftype Foreign type of the object being pointed to.
returns The value of the object where the pointer points.

Description Returns the object to which a pointer points. **setf**-able.

Notes Casting of the pointer may be performed with **ffi:with-cast-pointer** together with **ffi:deref-pointer**/**ffi:deref-array**.

Examples

```
(let ((intp (ffi:allocate-foreign-object :int)))
```

```
(setf (ffi:deref-pointer intp :int) 10)
(prog1
  (ffi:deref-pointer intp :int)
  (ffi:free-foreign-object intp)))
;; => 10
```

ffi:ensure-char-character *object* [Function]

Ensures that a dereferenced `:char` pointer is a character

object Either a character or a integer specifying a character code.

returns A character.

Description Ensures that an objects obtained by dereferencing `:char` and `:unsigned-char` pointers is a lisp character.

Examples

```
(let ((fs (ffi:convert-to-foreign-string "a")))
  (prog1
    (ffi:ensure-char-character (ffi:deref-pointer fs :char))
    (ffi:free-foreign-object fs)))
;; => #\a
```

Exceptional Situations Depending upon the implementation and what UFFI expects, this macro may signal an error if the object is not a character or integer.

ffi:ensure-char-integer *object* [Function]

Ensures that a dereferenced `:char` pointer is an integer

object Either a character or a integer specifying a character code.

returns An integer.

Description Ensures that an objects obtained by dereferencing `:char` and `:unsigned-char` pointers is a lisp integer.

Examples

```
(let ((fs (ffi:convert-to-foreign-string "a")))
  (prog1
    (ffi:ensure-char-integer (ffi:deref-pointer fs :char))
    (ffi:free-foreign-object fs)))
;; => 96
```

Exceptional Situations Depending upon the implementation and what UFFI expects, this macro may signal an error if the object is not a character or integer.

ffi:make-null-pointer *ftype* [Function]

Create a NULL pointer of a specified type

ftype A type of object to which the pointer refers.

returns The NULL pointer of type *ftype*.

ffi:null-pointer-p *ptr* [Function]

Tests a pointer for NULL value

ptr A foreign object pointer.

returns The boolean flag.

ffi:+null-cstring-pointer+ [Variable]

A NULL cstring pointer. This can be used for testing if a cstring returned by a function is NULL.

ffi:with-cast-pointer (*var ptr ftype*) **&body** *body* [Macro]

Wraps a body of code with a pointer cast to a new type

var Symbol which will be bound to the casted object.

ptr Pointer to a foreign object.

ftype A foreign type of the object being pointed to.

returns The value of the object where the pointer points.

Description Executes *body* with *ptr* cast to be a pointer to type *ftype*. *var* will be bound to this value during the execution of *body*.

Examples

```
(ffi:with-foreign-object (size :int)
  ;; F00 is a foreign function returning a :POINTER-VOID
  (let ((memory (foo size)))
    (when (mumble)
      ;; at this point we know for some reason that MEMORY points
      ;; to an array of unsigned bytes
      (ffi:with-cast-pointer (memory :unsigned-byte)
        (dotimes (i (deref-pointer size :int))
          (do-something-with
            (ffi:deref-array memory '(:array :unsigned-byte) i)))))))■
```

ffi:def-foreign-var *name type module* [Macro]

Defines a symbol macro to access a variable in foreign code

name A string or list specifying the symbol macro's name. If it is a string, that names the foreign variable. A Lisp name is created by translating #_ to #_ and by converting to upper-case.

If it is a list, the first item is a string specifying the foreign variable name and the second it is a symbol stating the Lisp name.

type A foreign type of the foreign variable.

module Either a string specifying the module (or library) the foreign variable resides in, **:default** if no module needs to be loaded or **nil** to use SFFI.

Description Defines a symbol macro which can be used to access (get and set) the value of a variable in foreign code.

Examples

C code defining foreign structure, standalone integer and the accessor:

```
int baz = 3;

typedef struct {
    int x;
    double y;
} foo_struct;

foo_struct the_struct = { 42, 3.2 };

int foo () {
    return baz;
}
```

Lisp code defining C structure, function and a variable:

```
(ffi:def-struct foo-struct
  (x :int)
  (y :double))

(ffi:def-function ("foo" foo) ()
  :returning :int
  :module "foo")

(ffi:def-foreign-var ("baz" *baz*) :int "foo")
(ffi:def-foreign-var ("the_struct" *the-struct*) foo-struct "foo")

*baz*          ;; => 3
(incf *baz*)    ;; => 4
(foo)           ;; => 4
```

3.3.7.4 Foreign Strings

Overview

UFFI has functions to two types of C-compatible strings: cstrings and foreign strings. cstrings are used only as parameters to and from functions. In some implementations a cstring is not a foreign type but rather the Lisp string itself. On other platforms a cstring is a newly allocated foreign vector for storing characters. The following is an example of using cstrings to both send and return a value.

```
(ffi:def-function ("getenv" c-getenv)
  ((name :cstring))
  :returning :cstring)

(defun my-getenv (key)
  "Returns an environment variable, or NIL if it does not exist"
  (check-type key string)
  (ffi:with-cstring (key-native key)
    (ffi:convert-from-cstring (c-getenv key-native))))
```

In contrast, foreign strings are always a foreign vector of characters which have memory allocated. Thus, if you need to allocate memory to hold the return value of a string, you must use a foreign string and not a cstring. The following is an example of using a foreign string for a return value.

```
(ffi:def-function ("gethostname" c-gethostname)
  ((name (* :unsigned-char))
   (len :int))
  :returning :int)

(defun gethostname ()
  "Returns the hostname"
  (let* ((name (ffi:allocate-foreign-string 256))
         (result-code (c-gethostname name 256))
         (hostname (when (zerop result-code)
                       (ffi:convert-from-foreign-string name))))
    ;; UFFI does not yet provide a universal way to free
    ;; memory allocated by C's malloc. At this point, a program
    ;; needs to call C's free function to free such memory.
    (unless (zerop result-code)
      (error "gethostname() failed."))))
```

Foreign functions that return pointers to freshly allocated strings should in general not return cstrings, but foreign strings. (There is no portable way to release such cstrings from Lisp.) The following is an example of handling such a function.

```
(ffi:def-function ("readline" c-readline)
  ((prompt :cstring))
  :returning (* :char))

(defun readline (prompt)
  "Reads a string from console with line-editing."
  (ffi:with-cstring (c-prompt prompt)
    (let* ((c-str (c-readline c-prompt))
           (str (ffi:convert-from-foreign-string c-str)))
      (ffi:free-foreign-object c-str)
      str)))
```

Reference

ffi:convert-from-cstring *object* [Macro]

Converts a cstring to a Lisp string

object A cstring

returns A Lisp string

Description Converts a Lisp string to a cstring. This is most often used when processing the results of a foreign function that returns a cstring.

ffi:convert-to-cstring *object* [Macro]

Converts a Lisp string to a cstring

object A Lisp string

returns A `cstring`

Description Converts a Lisp string to a `cstring`. The `cstring` should be freed with `ffi:free-cstring`.

Side Effects This function allocates memory.

`ffi:convert-from-cstring` *cstring* [Macro]

Free memory used by *cstring*

cstring `cstring` to be freed.

Description Frees any memory possibly allocated by `ffi:convert-to-cstring`. On ECL, a `cstring` is just the Lisp string itself.

`ffi:with-cstring` (*cstring string*) **&body** *body* [Macro]

Binds a newly created `cstring`

cstring A symbol naming the `cstring` to be created.

string A Lisp string that will be translated to a `cstring`.

body The body of where the *cstring* will be bound.

returns Result of evaluating the *body*.

Description Binds a symbol to a `cstring` created from conversion of a *string*. Automatically frees the *cstring*.

Examples

```
(ffi:def-function ("getenv" c-getenv)
  ((name :cstring))
  :returning :cstring)

(defun getenv (key)
  "Returns an environment variable, or NIL if it does not exist"
  (check-type key string)
  (ffi:with-cstring (key-cstring key)
    (ffi:convert-from-cstring (c-getenv key-cstring)))))
```

`ffi:with-cstrings` *bindings* **&body** *body* [Macro]

Binds a newly created `cstrings`

bindings List of pairs (*cstring string*), where *cstring* is a name for a `cstring` translated from Lisp string *string*.

body The body of where the *bindings* will be bound.

returns Result of evaluating the *body*.

Description Binds a symbols to a `cstrings` created from conversion of a *strings*. Automatically frees the *cstrings*. This macro works similar to `let*`. Based on `with-cstring`.

ffi:convert-from-foreign-string *foreign-string* &**key** *length* [Macro]
 (*null-terminated-p* **t**)

Converts a foreign string into a Lisp string

foreign-string

A foreign string.

length The length of the foreign string to convert. The default is the length of the string until a NULL character is reached.

null-terminated-p

A boolean flag with a default value of **t**. When true, the string is converted until the first NULL character is reached.

returns A Lisp string.

Description Returns a Lisp string from a foreign string. Can translate ASCII and binary strings.

ffi:convert-to-foreign-string *string* [Macro]

Converts a Lisp string to a foreign string

string A Lisp string.

returns A foreign string.

Description Converts a Lisp string to a foreign string. Memory should be freed with **ffi:free-foreign-object**.

ffi:allocate-foreign-string *size* &**key** *unsigned* [Macro]

Allocates space for a foreign string

size The size of the space to be allocated in bytes.

unsigned A boolean flag with a default value of **t**. When true, marks the pointer as an **:unsigned-char**.

returns A foreign string which has undefined contents.

Description Allocates space for a foreign string. Memory should be freed with **ffi:free-foreign-object**.

ffi:with-foreign-string (*foreign-string string*) &**body** *body* [Macro]

Binds a newly allocated **foreign-string**

foreign-string

A symbol naming the **foreign string** to be created.

string A Lisp string that will be translated to a **foreign string**.

body The body of where the *foreign-string* will be bound.

returns Result of evaluating the *body*.

Description Binds a symbol to a **foreign-string** created from conversion of a *string*. Automatically deallocates the *foreign-string*.

Examples

ffi:with-foreign-strings *bindings &body body* [Macro]

Binds a newly created foreign string

bindings List of pairs (*foreign-string string*), where *foreign-string* is a name for a foreign string translated from Lisp string *string*.

body The body of where the *bindings* will be bound.

returns Result of evaluating the *body*.

Description Binds a symbols to a **foreign-strings** created from conversion of a *strings*. Automatically frees the *foreign-strings*. This macro works similar to **let***. Based on **ffi:with-foreign-string**.

3.3.7.5 Functions and Libraries

Reference

ffi:def-function *name args &key module (returning :void) (call :cdecl)* [Macro]

name A string or list specifying the function name. If it is a string, that names the foreign function. A Lisp name is created by translating #_ to #_- and by converting to upper-case in case-insensitive Lisp implementations. If it is a list, the first item is a string specifying the foreign function name and the second it is a symbol stating the Lisp name.

args A list of argument declarations. If *nil*, indicates that the function does not take any arguments.

module Either a string specifying which module (or library) that the foreign function resides, **:default** if no module needs to be loaded or **nil** to use SFFI.

call Function calling convention. May be one of **:default**, **:cdecl**, **:sysv**, **:stdcall**, **:win64** and **:unix64**.

This argument is used only when we're using the dynamic function interface. If ECL is built without the DFFI support, then it uses SFFI the *call* argument is ignored.

returning A declaration specifying the result type of the foreign function. **:void** indicates that the function does not return any value.

Description Declares a foreign function.

Examples

```
(ffi:def-function "gethostname"
  ((name (* :unsigned-char))
   (len :int))
  :returning :int)
```

ffi:load-foreign-library *filename &key module supporting-libraries force-load system-library* [Macro]

filename A string or pathname specifying the library location in the filesystem.

module **IGNORED** A string designating the name of the module to apply to functions in this library.

supporting-libraries

IGNORED A list of strings naming the libraries required to link the foreign library.

force-load **IGNORED** Forces the loading of the library if it has been previously loaded.

system-library

Denotes if the loaded library is a system library (accessible with the correct linker flags). If **t**, then SFFI is used and the linking is performed after compilation of the module. Otherwise (default) both SFFI and DFFI are used, but SFFI only during the compilation.

returns A generalized boolean *true* if the library was able to be loaded successfully or if the library has been previously loaded, otherwise **nil**.

Description Loads a foreign library. Ensures that a library is only loaded once during a session.

Examples

```
(ffi:load-foreign-library #p"/usr/lib/libmagic.so.1")
;; => #<codeblock "/usr/lib/libmagic.so">
```

Side Effects Loads the foreign code into the Lisp system.

Affected by Ability to load the file.

ffi:find-foreign-library *names directories &key drive-letters* [Function]
types

Finds a foreign library file

names A string or list of strings containing the base name of the library file.

directories A string or list of strings containing the directory the library file.

drive-letters

A string or list of strings containing the drive letters for the library file.

types A string or list of strings containing the file type of the library file. Default is **nil**. If **nil**, will use a default type based on the currently running implementation.

returns A path containing the path to the *first* file found, or **nil** if the library file was not found.

Description Finds a foreign library by searching through a number of possible locations. Returns the path of the first found file.

Examples

```
(ffi:find-foreign-library '("libz" "libmagic")
                          '("/usr/local/lib/" "/usr/lib/")
                          :types '("so" "dll"))
;; => #P"/usr/lib/libz.so.1.2.8"
```

3.4 Native threads

3.4.1 Tasks, threads or processes

On most platforms, ECL supports native multithreading. That means there can be several tasks executing lisp code on parallel and sharing memory, variables and files. The interface for multitasking in ECL, like those of most other implementations, is based on a set of functions and types that resemble the multiprocessing capabilities of old Lisp Machines.

This backward compatibility is why tasks or threads are called "processes". However, they should not be confused with operating system processes, which are made of programs running in separate contexts and without access to each other's memory.

The implementation of threads in ECL is purely native and based on Posix Threads wherever available. The use of native threads has advantages. For instance, they allow for non-blocking file operations, so that while one task is reading a file, a different one is performing a computation.

As mentioned above, tasks share the same memory, as well as the set of open files and sockets. This manifests on two features. First of all, different tasks can operate on the same lisp objects, reading and writing their slots, or manipulating the same arrays. Second, while threads share global variables, constants and function definitions they can also have thread-local bindings to special variables that are not seen by other tasks.

The fact that different tasks have access to the same set of data allows both for flexibility and a greater risk. In order to control access to different resources, ECL provides the user with locks, as explained in the next section.

3.4.2 Processes (native threads)

Process is a primitive representing native thread.

3.4.3 Processes dictionary

`cl-object mp_all_processes ()` [Function]

`mp:all-processes` [Function]

Returns the list of processes associated to running tasks. The list is a fresh new one and can be destructively modified. However, it may happen that the output list is not up to date, because some of the tasks have expired before this copy is returned.

`cl-object mp_exit_process () ecl_attr_noreturn` [Function]

`mp:exit-process` [Function]

When called from a running task, this function immediately causes the task to finish. When invoked from the main thread, it is equivalent to invoking `ext:quit` with exit code 0.

`cl-object mp_interrupt_process (cl-object process, cl-object function)` [Function]

`mp:interrupt-process process function` [Function]

Interrupt a task. This function sends a signal to a running *process*. When the task is free to process that signal, it will stop whatever it is doing and execute the given function.

WARNING: Use with care! Interrupts can happen anywhere, except in code regions explicitly protected with `mp:without-interrupts`. This can lead to dangerous situations when interrupting functions which are not thread safe. In particular, one has to consider:

- Reentrancy: Functions, which usually are not called recursively can be re-entered during execution of the interrupt.
- Stack unwinding: Non-local jumps like `throw` or `return-from` in the interrupting code will handle `unwind-protect` forms like usual. However, the cleanup forms of an `unwind-protect` can still be interrupted. In that case the execution flow will jump to the next `unwind-protect`.

Example:

Kill a task that is doing nothing (See `mp:process-kill`).

```
(flet ((task-to-be-killed ()
      ;; Infinite loop
      (loop (sleep 1))))
  (let ((task (mp:process-run-function 'background #'task-to-be-killed)))
    (sleep 10)
    (mp:interrupt-process task 'mp:exit-process)))
```

`cl_object mp_make_process (cl_narg nargs, ...)` [Function]

`mp:make-process &key name initial-bindings` [Function]

Create a new thread. This function creates a separate task with a name set to *name* and no function to run. See also `mp:process-run-function`. Returns newly created process.

If *initial-bindings* is false, the new process inherits local bindings to special variables (i.e. binding a special variable with `let` or `let*`) from the current thread, otherwise the new thread possesses no local bindings.

`cl_object mp_process_active_p (cl_object process)` [Function]

`mp:process-active-p process` [Function]

Returns `t` when *process* is active, `nil` otherwise. Signals an error if *process* doesn't designate a valid process.

`cl_object mp_process_enable (cl_object process)` [Function]

`mp:process-enable process` [Function]

The argument to this function should be a process created by `mp:make-process`, which has a function associated as per `mp:process-preset` but which is not yet running. After invoking this function a new thread will be created in which the associated function will be executed. Returns *process* if the thread creation was successful and `nil` otherwise.

```
(defun process-run-function (process-name process-function &rest args)
  (let ((process (mp:make-process name)))
    (apply #'mp:process-preset process function args)
    (mp:process-enable process)))
```

`cl_object mp_process_yield ()` [Function]

`mp:process-yield` [Function]

Yield the processor to other threads.

`cl_object mp_process_join (cl_object process)` [Function]

`mp:process-join process` [Function]

Suspend current thread until *process* exits. Return the result values of the *process* function.

`cl_object mp_process_kill (cl_object process)` [Function]

`mp:process-kill process` [Function]

Try to stop a running task. Killing a process may fail if the task has disabled interrupts.

Example:

Kill a task that is doing nothing

```
(flet ((task-to-be-killed ()
      ;; Infinite loop
      (loop (sleep 1))))
  (let ((task (mp:process-run-function 'background #'task-to-be-killed)))
    (sleep 10)
    (mp:process-kill task)))
```

`cl_object mp_process_suspend (cl_object process)` [Function]

`mp:process-suspend process` [Function]

Suspend a running *process*. May be resumed with `mp:process-resume`.

Example:

```
(flet ((ticking-task ()
      ;; Infinite loop
      (loop
        (sleep 1)
        (print :tick))))
  (print "Running task (one tick per second)")
  (let ((task (mp:process-run-function 'background #'ticking-task)))
    (sleep 5)
    (print "Suspending task for 5 seconds")
    (mp:process-suspend task)
    (sleep 5)
    (print "Resuming task for 5 seconds")
    (mp:process-resume task)
    (sleep 5)
    (print "Killing task")
    (mp:process-kill task)))
```

`cl_object mp_process_resume (cl_object process)` [Function]

`mp:process-resume process` [Function]

Resumes a suspended *process*. See example in `mp:process-suspend`.

`cl_object mp_process_name (cl_object process)` [Function]

`mp:process-name process` [Function]

Returns the name of a *process* (if any).

`cl_object mp_process_preset (cl_narg nargs, cl_object process, cl_object function, ...)` [Function]

`mp:process-preset process function &rest function-args` [Function]

Associates a *function* to call with the arguments *function-args*, with a stopped *process*. The function will be the entry point when the task is enabled in the future.

See `mp:process-enable` and `mp:process-run-function`.

`cl_object mp_process_run_function (cl_narg nargs, cl_object name, cl_object function, ...)` [Function]

`mp:process-run-function name function &rest function-args` [Function]

Create a new process using `mp:make-process`, associate a function to it and start it using `mp:process-preset`.

Example:

```
(flet ((count-numbers (end-number)
      (dotimes (i end-number)
        (format t "~%;;; Counting: ~i" i)
        (terpri)
        (sleep 1))))
  (mp:process-run-function 'counter #'count-numbers 10))
```

`cl_object mp_current_process ()` [Function]

`mp:*current-process*` [Variable]

Returns/holds the current process of a caller.

`cl_object mp_block_signals ()` [Function]

`mp:block-signals` [Function]

Blocks process for interrupts and returns the previous sigmask.

See `mp:interrupt-process`.

`cl_object mp_restore_signals (cl_object sigmask)` [Function]

`mp:restore-signals sigmask` [Function]

Enables the interrupts from *sigmask*.

See `mp:interrupt-process`.

`mp:without-interrupts &body body` [Macro]

Executes *body* with all deferrable interrupts disabled. Deferrable interrupts arriving during execution of the *body* take effect after *body* has been executed.

Deferrable interrupts include most blockable POSIX signals, and `mp:interrupt-process`. Does not interfere with garbage collection, and unlike in many traditional Lisps using userspace threads, in ECL `mp:without-interrupts` does not inhibit scheduling of other threads.

Binds `mp:allow-with-interrupts`, `mp:with-local-interrupts` and `mp:with-restored-interrupts` as a local macros.

`mp:with-restored-interrupts` executes the body with interrupts enabled if and only if the `mp:without-interrupts` was in an environment in which interrupts were allowed.

`mp:allow-with-interrupts` allows the `mp:with-interrupts` to take effect during the dynamic scope of its body, unless there is an outer `mp:without-interrupts` without a corresponding `mp:allow-with-interrupts`.

`mp:with-local-interrupts` executes its body with interrupts enabled provided that there is an `mp:allow-with-interrupts` for every `mp:without-interrupts` surrounding the current one. `mp:with-local-interrupts` is equivalent to:

```
(mp:allow-with-interrupts (mp:with-interrupts ...))
```

Care must be taken not to let either `mp:allow-with-interrupts` or `mp:with-local-interrupts` appear in a function that escapes from inside the `mp:without-interrupts` in:

```
(mp:without-interrupts
  ;; The body of the lambda would be executed with WITH-INTERRUPTS allowed
  ;; regardless of the interrupt policy in effect when it is called.
  (lambda () (mp:allow-with-interrupts ...)))
```

```
(mp:without-interrupts
  ;; The body of the lambda would be executed with interrupts enabled
  ;; regardless of the interrupt policy in effect when it is called.
  (lambda () (mp:with-local-interrupts ...)))
```

`mp:with-interrupts &body body` [Macro]

Executes *body* with deferrable interrupts conditionally enabled. If there are pending interrupts they take effect prior to executing *body*.

As interrupts are normally allowed `mp:with-interrupts` only makes sense if there is an outer `mp:without-interrupts` with a corresponding `mp:allow-with-interrupts`: interrupts are not enabled if any outer `mp:without-interrupts` is not accompanied by `mp:allow-with-interrupts`.

3.4.4 Locks (mutexes)

Locks are used to synchronize access to the shared data. Lock may be owned only by a single thread at any given time. Recursive locks may be re-acquired by the same thread multiple times (and non-recursive locks can't).

3.4.5 Locks dictionary

`cl_object ecl_make_lock (cl_object name, bool recursive)` [Function]

C/C++ equivalent of `mp:make-lock` without *key* arguments.

See `mp:make-lock`.

`mp:make-lock &key name (recursive nil)` [Function]

Creates a lock named *name*. If *recursive* is true, a recursive lock is created that can be locked multiple times by the same thread.

<code>cl_object mp_recursive_lock_p (cl_object lock)</code>	[Function]
<code>mp:recursive-lock-p lock</code> Predicate verifying if <i>lock</i> is recursive.	[Function]
<code>cl_object mp_holding_lock_p (cl_object lock)</code>	[Function]
<code>mp:holding-lock-p lock</code> Predicate verifying if the current thread holds <i>lock</i> .	[Function]
<code>cl_object mp_lock_name (cl_object lock)</code>	[Function]
<code>mp:lock_name lock</code> Returns the name of <i>lock</i> .	[Function]
<code>cl_object mp_lock_owner (cl_object lock)</code>	[Function]
<code>mp:lock-owner lock</code> Returns the process owning <i>lock</i> . For testing whether the current thread is holding a lock see <code>mp:holding-lock-p</code> .	[Function]
<code>cl_object mp_lock_count (cl_object lock)</code>	[Function]
<code>mp:lock-count lock</code> Returns number of processes waiting for <i>lock</i> .	[Function]
<code>cl_object mp_get_lock_wait (cl_object lock)</code> Grabs a lock (blocking if <i>lock</i> is already taken). Returns <code>ECL_T</code> .	[Function]
<code>cl_object mp_get_lock_nowait</code> Grabs a lock if free (non-blocking). If <i>lock</i> is already taken returns <code>ECL_NIL</code> , otherwise <code>ECL_T</code> .	[Function]
<code>mp:get-lock lock &optional (wait t)</code> Tries to acquire a lock. <i>wait</i> indicates whether function should block or give up if <i>lock</i> is already taken. If <i>wait</i> is <code>nil</code> and <i>lock</i> can't be acquired returns <code>nil</code> . Successful operation returns <code>t</code> .	[Function]
<code>cl_object mp_giveup_lock (cl_object lock)</code>	[Function]
<code>mp:giveup-lock lock</code> Releases <i>lock</i> .	[Function]
<code>mp:with-lock (lock-form) &body body</code> Acquire lock for the dynamic scope of <i>body</i> , which is executed with the lock held by current thread. Returns the values of <i>body</i> .	[Macro]

3.4.6 Readers-writer locks

Readers-writer (or shared-exclusive) locks allow concurrent access for read-only operations, while write operations require exclusive access. `mp:rwlock` is non-recursive.

Readers-writers locks are an optional feature, which is available if `*features*` includes `:ecl-read-write-lock`.

3.4.7 Read-Write locks dictionary

- `cl_object ecl_make_rwlock (cl_object name)` [Function]
 C/C++ equivalent of `mp:make-rwlock` without `key` arguments.
 See `mp:make-rwlock`.
- `mp:make-rwlock &key name` [Function]
 Creates a `rwlock` named `name`.
- `cl_object mp_rwlock_name (cl_object lock)` [Function]
- `mp:rwlock-name lock` [Function]
 Returns the name of `lock`.
- `cl_object mp_get_rwlock_read_wait (cl_object lock)` [Function]
 Acquires `lock` (blocks if `lock` is already taken with `mp:get-rwlock-write`. Lock may be acquired by multiple readers). Returns `ECL_T`.
- `cl_object mp_get_rwlock_read_nowait` [Function]
 Tries to acquire `lock`. If `lock` is already taken with `mp:get-rwlock-write` returns `ECL_NIL`, otherwise `ECL_T`.
- `mp:get-rwlock-read lock &optional (wait t)` [Function]
 Tries to acquire `lock`. `wait` indicates whenever function should block or give up if `lock` is already taken with `mp:get-rwlock-write`.
- `cl_object mp_get_rwlock_write_wait (cl_object lock)` [Function]
 Acquires `lock` (blocks if `lock` is already taken). Returns `ECL_T`.
- `cl_object mp_get_rwlock_write_nowait` [Function]
 Tries to acquire `lock`. If `lock` is already taken returns `ECL_NIL`, otherwise `ECL_T`.
- `mp:get-rwlock-write lock &optional (wait t)` [Function]
 Tries to acquire `lock`. `wait` indicates whenever function should block or give up if `lock` is already taken.
- `cl_object mp_giveup_rwlock_read (cl_object lock)` [Function]
`cl_object mp_giveup_rwlock_write (cl_object lock)` [Function]
- `mp:giveup-rwlock-read lock` [Function]
`mp:giveup-rwlock-write lock` [Function]
 Release `lock`.
- `mp:with-rwlock (lock operation) &body body` [Macro]
 Acquire `rwlock` for the dynamic scope of `body` for operation `operation`, which is executed with the lock held by current thread. Returns the values of `body`.
 Valid values of argument `operation` are `:read` or `:write` (for reader and writer access accordingly).

3.4.8 Condition variables

Condition variables are used to wait for a particular condition becoming true (e.g new client connects to the server).

3.4.9 Condition variables dictionary

<code>cl_object mp_make_condition_variable ()</code>	[Function]
<code>mp:make-condition-variable</code> Creates a condition variable.	[Function]
<code>cl_object mp_condition_variable_wait (cl_object cv, cl_object lock)</code>	[Function]
<code>mp:condition-variable-wait cv lock</code> Release <i>lock</i> and suspend thread until condition <code>mp:condition-variable-signal</code> is called on <i>cv</i> . When thread resumes re-acquire <i>lock</i> .	[Function]
<code>cl_object mp_condition_variable_timedwait (cl_object cv, cl_object lock, cl_object seconds)</code>	[Function]
<code>mp:condition-variable-timedwait cv lock seconds</code> <code>mp:condition-variable-wait</code> which timeouts after <i>seconds</i> seconds.	[Function]
<code>cl_object mp_condition_variable_signal (cl_object cv)</code>	[Function]
<code>mp:condition-variable-signal cv</code> Signal <i>cv</i> (wakes up only one waiter). After signal, signaling thread keeps lock, waking thread goes on the queue waiting for the lock. See <code>mp:condition-variable-wait</code> .	[Function]
<code>cl_object mp_condition_variable_broadcast (cl_object cv)</code>	[Function]
<code>mp:condition-variable-broadcast cv</code> Signal <i>cv</i> (wakes up all waiters). See <code>mp:condition-variable-wait</code> .	[Function]

3.4.10 Semaphores

Semaphores are objects which allow an arbitrary resource count. Semaphores are used for shared access to resources where number of concurrent threads allowed to access it is limited.

3.4.11 Semaphores dictionary

<code>cl_object ecl_make_semaphore (cl_object name, cl_fixnum count)</code> C/C++ equivalent of <code>mp:make-semaphore</code> without <i>key</i> arguments. See <code>mp:make-semaphore</code> .	[Function]
<code>mp:make-semaphore &key name count</code> Creates a counting semaphore <i>name</i> with a resource count <i>count</i> .	[Function]
<code>cl_object mp_semaphore_name (cl_object semaphore)</code>	[Function]
<code>mp:semaphore-name semaphore</code> Returns the name of <i>semaphore</i> .	[Function]

<code>cl_object mp_semaphore_count (cl_object semaphore)</code>	[Function]
<code>mp:semaphore-count semaphore</code>	[Function]
Returns the resource count of <i>semaphore</i> .	
<code>cl_object mp_semaphore_wait_count (cl_object semaphore)</code>	[Function]
<code>mp:semaphore-wait-count semaphore</code>	[Function]
Returns the number of threads waiting on <i>semaphore</i> .	
<code>cl_object mp_wait_on_semaphore (cl_object semaphore)</code>	[Function]
<code>mp:wait-on-semaphore semaphore</code>	[Function]
Waits on semaphore until it can grab the resource (blocking). Returns resource count before semaphore was acquired.	
<code>cl_object mp_try_get_semaphore (cl_object semaphore)</code>	[Function]
<code>mp:try-get-semaphore semaphore</code>	[Function]
Tries to get a semaphore (non-blocking). If there is no resource left returns <code>nil</code> , otherwise returns resource count before semaphore was acquired.	
<code>cl_object mp_signal_semaphore (cl_narg n, cl_object sem, ...);</code>	[Function]
<code>mp:signal-semaphore semaphore &optional (count 1)</code>	[Function]
Releases <i>count</i> units of a resource on <i>semaphore</i> .	

3.4.12 Atomic operations

ECL supports both compare-and-swap and fetch-and-add (which may be faster on some processors) atomic operations on a number of different places. The compare-and-swap macro is user extensible with a protocol similar to `setf`.

3.4.13 Atomic operations dictionary

C Reference

<code>cl_object ecl_compare_and_swap (cl_object *slot, cl_object old, cl_object new)</code>	[Function]
Perform an atomic compare and swap operation on <i>slot</i> and return the previous value stored in <i>slot</i> . If the return value is equal to <i>old</i> (comparison by <code>==</code>), the operation has succeeded. This is a inline-only function defined in "ecl/ecl_atomics.h".	
<code>cl_object ecl_atomic_inc (cl_object *slot, cl_object increment)</code>	[Function]
<code>cl_object ecl_atomic_inc_by_fixnum (cl_object *slot, cl_fixnum increment)</code>	[Function]
Atomically increment <i>slot</i> by the given increment and return the previous value stored in <i>slot</i> . The consequences are undefined if the value of <i>slot</i> is not of type <code>fixnum</code> . <code>ecl_atomic_inc</code> signals an error if <i>increment</i> is not of type <code>fixnum</code> . This is a inline-only function defined in "ecl/ecl_atomics.h".	
<code>cl_index ecl_atomic_index_inc (cl_index *slot);</code>	[Function]
Atomically increment <i>slot</i> by 1 and return the new value stored in <i>slot</i> .	

`cl_object ecl_atomic_get (cl_object *slot)` [Function]
 Perform a volatile load of the object in *slot* and then atomically set *slot* to `ECL_NIL`.
 Returns the value previously stored in *slot*.

`void ecl_atomic_push (cl_object *slot, cl_object o)` [Function]

`cl_object ecl_atomic_pop (cl_object *slot)` [Function]
 Like push/pop but atomic.

Lisp Reference

`mp:atomic-incf place &optional (increment 1)` [Macro]

`mp:atomic-decf place &optional (increment 1)` [Macro]

Atomically increments/decrements the fixnum stored in *place* by the given *increment* and returns the value of *place* before the increment. Incrementing and decrementing is done using modular arithmetic, so that `mp:atomic-incf` of a place whose value is `most-positive-fixnum` by 1 results in `most-negative-fixnum` stored in *place*.

Currently the following places are supported:

`car`, `cdr`, `first`, `rest`, `svref`, `symbol-value`, `slot-value`, `clos:standard-instance-access`, `clos:funcallable-standard-instance-access`.

For `slot-value`, the object should have no applicable methods defined for `slot-value-using-class` or `(setf slot-value-using-class)`.

The consequences are undefined if the value of *place* is not of type `fixnum`.

`mp:compare-and-swap place old new` [Macro]

Atomically stores *new* in *place* if *old* is `eq` to the current value of *place*. Returns the previous value of *place*: if the returned value is `eq` to *old*, the swap was carried out.

Currently, the following places are supported:

`car`, `cdr`, `first`, `rest`, `svref`, `symbol-plist`, `symbol-value`, `slot-value`, `clos:standard-instance-access`, `clos:funcallable-standard-instance-access`, a structure slot accessor² or any other place for which a compare-and-swap expansion was defined by `mp:defcas` or `mp:define-cas-expander`.

For `slot-value`, `slot-unbound` is called if the slot is unbound unless *old* is `eq` to `si:unbound`, in which case *old* is returned and *new* is assigned to the slot. Additionally, the object should have no applicable methods defined for `slot-value-using-class` or `(setf slot-value-using-class)`.

`mp:atomic-update place update-fn &rest arguments` [Macro]

Atomically updates the CAS-able *place* to the value returned by calling *update-fn* with *arguments* and the old value of *place*. *update-fn* must be a function accepting `(1+ (length arguments))` arguments. Returns the new value which was stored in *place*.

place may be read and *update-fn* may be called more than once if multiple threads are trying to write to *place* at the same time.

² The creation of atomic structure slot accessors can be deactivated by supplying a `(:atomic-accessors nil)` option to `defstruct`.

Example:

Atomic update of a structure slot. If the update would not be atomic, the result would be unpredictable.

```
(defstruct test-struct
  (slot1 0))
(let ((struct (make-test-struct)))
  (mapc #'mp:process-join
    (loop repeat 100
      collect (mp:process-run-function
        ""
        (lambda ()
          (loop repeat 1000 do
            (mp:atomic-update (test-struct-slot1 struct) #'1+)
            (sleep 0.00001))))))
    (test-struct-slot1 struct))
=> 100000
```

`mp:atomic-push` *obj place* [Macro]

`mp:atomic-pop` *place* [Macro]

Like `push/pop`, but atomic. *place* must be CAS-able and may be read multiple times before the update succeeds.

`mp:define-cas-expander` *accessor lambda-list &body body* [Macro]

Define a compare-and-swap expander similar to `define-setf-expander`. Defines the compare-and-swap-expander for generalized-variables (*accessor ...*). When a form `(mp:compare-and-swap (accessor arg1 ... argn) old new)` is evaluated, the forms given in the body of `mp:define-cas-expander` are evaluated in order with the parameters in *lambda-list* bound to *arg1 ... argn*. The body must return six values

```
(var1 ... vark)
(form1 ... formk)
old-var
new-var
compare-and-swap-form
volatile-access-form
```

in order (Note that *old-var* and *new-var* are single variables, unlike in `define-setf-expander`). The whole `compare-and-swap` form is then expanded into

```
(let* ((var1 from1) ... (vark formk)
      (old-var old-form)
      (new-var new-form))
  compare-and-swap-form).
```

Note that it is up to the user of this macro to ensure atomicity for the resulting compare-and-swap expansions.

Example

`mp:define-cas-expander` can be used to define a more convenient compare-and-swap expansion for a class slot. Consider the following class:

```
(defclass food ()
  ((name :initarg :name)
   (deliciousness :initform 5 :type '(integer 0 10)
                  :accessor food-deliciousness)))

(defvar *spätzle* (make-instance 'food :name "Spätzle"))
```

We can't just use `mp:compare-and-swap` on `*spätzle*`:

```
> (mp:compare-and-swap (food-deliciousness *x*) 5 10)
```

Condition of type: SIMPLE-ERROR

Cannot get the compare-and-swap expansion of (FOOD-DELICIOUSNESS *X*).

We can use `symbol-value`, but let's define a more convenient compare-and-swap expander:

```
(mp:define-cas-expander food-deliciousness (food)
  (let ((old (gensym))
        (new (gensym)))
    (values nil nil old new
            '(progn (check-type ,new (integer 0 10))
                    (mp:compare-and-swap (slot-value ,food 'deliciousness)
                                         ,old ,new))
            '(food-deliciousness ,food))))
```

Now finally, we can safely store our rating:

```
> (mp:compare-and-swap (food-deliciousness *spätzle*) 5 10)
```

5

mp:defcas *accessor cas-fun &optional documentation* [Macro]

Define a compare-and-swap expansion similar to the short form of `defsetf`. Defines an expansion

```
(compare-and-swap (accessor arg1 ... argn) old new)
=> (cas-fun arg1 ... argn old new)
```

Note that it is up to the user of this macro to ensure atomicity for the resulting compare-and-swap expansions.

mp:remcas *symbol* [Function]

Remove a compare-and-swap expansion. It is an equivalent of `fmakunbound (setf symbol)` for cas expansions.

mp:get-cas-expansion *place &optional environment* [Function]

Returns the compare-and-swap expansion forms and variables as defined in `mp:define-cas-expander` for *place* as six values.

3.5 Signals and Interrupts

3.5.1 Problems associated to signals

POSIX contemplates the notion of "signals", which are events that cause a process or a thread to be interrupted. Windows uses the term "exception", which includes also a more general kind of errors.

In both cases the consequence is that a thread or process may be interrupted at any time, either by causes which are intrinsic to them (synchronous signals), such as floating point exceptions, or extrinsic (asynchronous signals), such as the process being aborted by the user.

Of course, those interruptions are not always welcome. When the interrupt is delivered and a handler is invoked, the thread or even the whole program may be in an inconsistent state. For instance the thread may have acquired a lock, or it may be in the process of filling the fields of a structure. Furthermore, sometimes the signal that a process receives may not even be related to it, as in the case when a user presses *Ctrl-C* and a `SIGINT` signal is delivered to an arbitrary thread, or when the process receives the Windows exception `CTRL_CLOSE_EVENT` denoting that the terminal window is being closed.

Understanding this, POSIX restricts severely what functions can be called from a signal handler, thereby limiting its usefulness. However, Common Lisp users expect to be able to handle floating point exceptions and to gracefully manage user interrupts, program exits, etc. In an attempt to solve this seemingly impossible problem, ECL has taken a pragmatic approach that works, it is rather safe, but involves some work on the ECL maintainers and also on users that want to embed ECL as a library.

3.5.2 Kinds of signals

3.5.2.1 Synchronous signals

The name derives from POSIX and it denotes interrupts that occur due to the code that a particular thread executes. They are largely equivalent to C++ and Java exceptions, and in Windows they are called "unchecked exceptions."

Common Lisp programs may generate mostly three kinds of synchronous signals:

- Floating point exceptions, that result from overflows in computations, division by zero, and so on.
- Access violations, such as dereferencing NULL pointers, writing into regions of memory that are protected, etc.
- Process interrupts.

The first family of signals are generated by the floating point processing hardware in the computer, and they typically happen when code is compiled with low security settings, performing mathematical operations without checks.

The second family of signals may seem rare, but unfortunately they still happen quite often. One scenario is wrong code that handles memory directly via FFI. Another one is undetected stack overflows, which typically result in access to protected memory regions. Finally, a very common cause of these kind of exceptions is invoking a function that has been compiled with very low security settings with arguments that are not of the expected type – for instance, passing a float when a structure is expected.

The third family is related to the multiprocessing capabilities in Common Lisp systems and more precisely to the `mp:interrupt-process` function which is used to kill, interrupt and

inspect arbitrary threads. In POSIX systems ECL informs a given thread about the need to interrupt its execution by sending a particular signal from the set which is available to the user.

Note that in neither of these cases we should let the signal pass unnoticed. Access violations and floating point exceptions may propagate through the program causing more harm than expected, and without process interrupts we will not be able to stop and cancel different threads. The only question that remains, though, is whether such signals can be handled by the thread in which they were generated and how.

3.5.2.2 Asynchronous signals

In addition to the set of synchronous signals or "exceptions", we have a set of signals that denote "events", things that happen while the program is being executed, and "requests". Some typical examples are:

- Request for program termination (`SIGKILL`, `SIGTERM`).
- Indication that a child process has finished.
- Request for program interruption (`SIGINT`), typically as a consequence of pressing a key combination, e.g. `Ctrl-C`.

The important difference with synchronous signals is that we have no thread that causes the interrupt and thus there is no preferred way of handling them. Moreover, the operating system will typically dispatch these signals to an arbitrary thread, unless we set up mechanisms to prevent it. This can have nasty consequences if the incoming signal interrupt a system call, or leaves the interrupted thread in an inconsistent state.

3.5.3 Signals and interrupts in ECL

The signal handling facilities in ECL are constrained by two needs. First of all, we can not ignore the synchronous signals mentioned in Section 3.5.2.1 [Signals and Interrupts - Synchronous signals], page 134. Second, all other signals should cause the least harm to the running threads. Third, when a signal is handled synchronously using a signal handler, the handler should do almost nothing unless we are completely sure that we are in an interruptible region, that is outside system calls, in code that ECL knows and controls.

The way in which this is solved is based on the existence of both synchronous and asynchronous signal handling code, as explained in the following two sections.

3.5.3.1 Handling of asynchronous signals

In systems in which this is possible, ECL creates a signal handling thread to detect and process asynchronous signals (See Section 3.5.2.2 [Signals and Interrupts - Asynchronous signals], page 135). This thread is a trivial one and does not process the signals itself: it communicates with, or launches new signal handling threads to act accordingly to the denoted events.

The use of a separate thread has some nice consequences. The first one is that those signals will not interrupt any sensitive code. The second one is that the signal handling thread will be able to execute arbitrary lisp or C code, since it is not being executed in a sensitive context. Most important, this style of signal handling is the recommended one by the POSIX standards, and it is the one that Windows uses.

The installation of the signal handling thread is dictated by a boot time option, `ECL_OPT_SIGNAL_HANDLING_THREAD` (see Table 1.1 for a summary of boot options), and it will only be possible in systems that support either POSIX or Windows threads.

Systems which embed ECL as an extension language may wish to deactivate the signal handling thread using the previously mentioned option. If this is the case, then they should take appropriate measures to avoid interrupting the code in ECL when such signals are delivered.

Systems which embed ECL and do not mind having a separate signal handling thread can control the set of asynchronous signals which is handled by this thread. This is done again using the appropriate boot options such as `ECL_OPT_TRAP_SIGINT`, `ECL_OPT_TRAP_SIGTERM`, etc. Note that in order to detect and handle those signals, ECL must block them from delivery to any other thread. This means changing the `sigprocmask()` in POSIX systems or setting up a custom `SetConsoleCtrlHandler()` in Windows.

3.5.3.2 Handling of synchronous signals

We have already mentioned that certain synchronous signals and exceptions can not be ignored and yet the corresponding signal handlers are not able to execute arbitrary code. To solve this seemingly impossible contradiction, ECL uses a simple solution, which is to mark the sections of code which are interruptible, and in which it is safe for the handler to run arbitrary code. All other regions would be considered "unsafe" and would be protected from signals and exceptions.

In principle this "marking" of safe areas can be done using POSIX functions such as `pthread_sigmask()` or `sigprocmask()`. However in practice this is slow, as it involves at least a function call, resolving thread-local variables, etc, etc, and it will not work in Windows.

Furthermore, sometimes we want signals to be detected but not to be immediately processed. For instance, when reading from the terminal we want to be able to interrupt the process, but we can not execute the code from the handler, since the C function which is used to read from the terminal, `read()`, may have left the input stream in an inconsistent, or even locked state.

The approach in ECL is more lightweight: we install our own signal handler and use a thread-local variable as a flag that determines whether the thread is executing interrupt safe code or not. More precisely, if the variable `ecl_process_env()->disable_interrupts` is set, signals and exceptions will be postponed and then the information about the signal is queued. Otherwise the appropriate code is executed: for instance invoking the debugger, jumping to a condition handler, quitting, etc.

Systems that embed ECL may wish to deactivate completely these signal handlers. This is done using the boot options, `ECL_OPT_TRAP_SIGFPE`, `ECL_OPT_TRAP_SIGSEGV`, `ECL_OPT_TRAP_SIGBUS`, `ECL_OPT_TRAP_INTERRUPT_SIGNAL`.

Systems that embed ECL and want to allow handling of synchronous signals should take care to also trap the associated lisp conditions that may arise. This is automatically taken care of by functions such as `si_safe_eval`, and in all other cases it can be solved by enclosing the unsafe code in a `ECL_CATCH_ALL` frame.

3.5.4 Considerations when embedding ECL

There are several approaches when handling signals and interrupts in a program that uses ECL. One is to install your own signal handlers. This is perfectly fine, but you should respect the same restrictions as ECL. Namely, you may not execute arbitrary code from those signal handlers, and in particular it will not always be safe to execute Common Lisp code from there.

If you want to use your own signal handlers then you should set the appropriate options before invoking `cl_boot`, as explained in `ecl_set_option`. Note that in this case ECL will not always be able to detect floating point exceptions.

The other option is to let ECL handle signals itself. This would be safer when the dominant part of the code is Common Lisp, but you may need to protect the code that embeds ECL from being interrupted using either the macros `ecl_disable_interrupts` and `ecl_enable_interrupts` or the POSIX functions `pthread_sigmask` and `sigprocmask`.

3.5.5 Signals Reference

`mp:with-interrupts &body body` [Macro]
`mp:without-interrupts &body body` [Macro]

Execute code with interrupts optionally enabled/disabled, See Section 3.4.3 [Processes dictionary], page 122.

`ext:unix-signal-received` [Condition]
 Unix signal condition

Class Precedence List condition, t

Methods

`ext:unix-signal-received-code condition` [Function]
 Returns the signal code of *condition*

`ext:get-signal-handler code` [Function]
 Queries the currently active signal handler for *code*.

`ext:set-signal-handler code handler` [Function]
 Arranges for the signal *code* to be caught in all threads and sets the signal handler for it. The value of *handler* modifies the signal handling behaviour as follows:

handler is a function designator

The function designated by *handler* will be invoked with no arguments

handler is a symbol denoting a condition type

A continuable error of the given type will be signaled

handler is equal to *code*

A condition of type `ext:unix-signal-received` with the corresponding signal code will be signaled

handler is `nil`

The signal will be caught but no handler will be called

ext:catch-signal *code flag &key process* [Function]

Changes the action taken on receiving the signal *code*. *flag* can be one of the following:

nil or **:ignore**

Ignore the signal

:default Use the default signal handling strategy of the operating system

t or **:catch**

Catch the signal and invoke the signal handler as given by **ext:get-signal-handler**

:mask, **:unmask**

Change the signal mask of either a) the not yet enabled *process* or b) the current process, if *process* is not supplied

Returns **t** on success and **nil** on failure.

Example:

```
CL-USER> (ext:catch-signal ext:+SIGPIPE+ :catch)
```

```
T
```

```
CL-USER> (ext:get-signal-handler ext:+SIGPIPE+)
```

```
NIL
```

```
CL-USER> (ext:set-signal-handler ext:+SIGPIPE+
```

```
      #'(lambda ()
```

```
          (format t "SIGPIPE detected in process: ~a~%" mp:
```

```
      #<bytecompiled-function 0x25ffca8>)
```

Passing the SIGPIPE signal to the ECL program with `killall -s SIGPIPE ecl` results in the output:

```
SIGPIPE detected in process: #<process TOP-LEVEL 0x1ecdfe0>
```

3.6 Memory Management

3.6.1 Introduction

ECL relies on the Boehm-Weiser garbage collector for handling memory, creating and destroying objects, and handling finalization of objects that are no longer reachable. The use of a garbage collector, and in particular the use of a portable one, imposes certain restrictions that may appear odd for C/C++ programmers.

In this section we will discuss garbage collection, how ECL configures and uses the memory management library, what users may expect, how to handle the memory and how to control the process by which objects are deleted.

3.6.2 Boehm-Weiser garbage collector

First of all, the garbage collector must be able to determine which objects are alive and which are not. In other words, the collector must be able to find all references to an object. One possibility would be to know where all variables of a program reside, and where is the stack of the program and its size, and parse all data there, discriminating references to lisp objects. To do this precisely one would need a very precise control of the data and stack

segments, as well as how objects are laid out by the C compiler. This is beyond ECL's scope and wishes and it can make coexistence with other libraries (C++, Fortran, etc) difficult.

The Boehm-Weiser garbage collector, on the other hand, is a conservative garbage collector. When scanning memory looking for references to live data, it guesses, conservatively, whether a word is a pointer or not. In case of doubt it will consider it to be a pointer and add it to the list of live objects. This may cause certain objects to be retained longer than what an user might expect but, in our experience, this is the best of both worlds and ECL uses certain strategies to minimize the amount of misinterpreted data.

More precisely, ECL uses the garbage collector with the following settings:

- The collector will not scan the data sectors. If you embed ECL in another program, or link libraries with ECL, you will have to notify ECL which variables point to lisp objects.
- The collector is configured to ignore pointers that point to the middle of allocated objects. This minimizes the risk of misinterpreting integers as pointers to live objects.
- It is possible to register finalizers that are invoked when an object is destroyed, but for that you should use ECL's API and understand the restriction described later in Section 3.6.5 [Finalization], page 140.

Except for finalization, which is a questionable feature, the previous settings are not very relevant for Common Lisp programmers, but are crucial for people interested in embedding in or cooperating with other C, C++ or Fortran libraries. Care should be taken when manipulating directly the GC library to avoid interfering with ECL's expectations.

3.6.3 Memory limits

Beginning with version 9.2.1, ECL operates a tighter control of the resources it uses. In particular, it features explicit limits in the four stacks and in the amount of live data. These limits are optional, can be changed at run time, but they allow users to better control the evolution of a program, handling memory and stack overflow gracefully via the Common Lisp condition system.

The customizable limits are listed in Table 3.1, but they need a careful description.

- **ext:heap-size** limits the total amount of memory which is available for lisp objects. This is the memory used when you create conses, arrays, structures, etc.
- **ext:c-stack** controls the size of the stack for compiled code, including ECL's library itself. This limit is less stringent than the others. For instance, when code is compiled with low safety settings, checks for this stack limit are usually omitted, for performance reasons.
- **ext:binding-stack** controls the number of nested bindings for special variables. The current value is usually safe enough, unless you have deep recursive functions that bind special variables, which is not really a good idea.
- **ext:frame-stack** controls the number of nested blocks, tagbodies and other control structures. It affects both interpreted and compiled code, but quite often compiled code optimizes away these stack frames, saving memory and not being affected by this limit.
- **ext:lisp-stack** controls the size of the interpreter stack. It only affects interpreted code.

If you look at Table 3.1, some of these limits may seem very stringent, but they exist to allow detecting and correcting both stack and memory overflow conditions. Larger values can be set systematically either in the `~/.eclrc` initialization file, or using the command line options from the table.

3.6.4 Memory conditions

When ECL surpasses or approaches the memory limits it will signal a Common Lisp condition. There are two types of conditions, `ext:stack-overflow` and `ext:storage-exhausted`, for stack and heap overflows, respectively. Both errors are correctable, as the following session shows:

```
> (defun foo (x) (foo x))

FOO
> (foo 1)
C-STACK overflow at size 1654784. Stack can probably be resized.
Broken at SI:BYTECODES.Available restarts:
1. (CONTINUE) Extend stack size
Broken at FOO.
>> :r1
C-STACK overflow at size 2514944. Stack can probably be resized.
Broken at SI:BYTECODES.Available restarts:
1. (CONTINUE) Extend stack size
Broken at FOO.
>> :q
Top level.
```

3.6.5 Finalization

As we all know, Common-Lisp relies on garbage collection for deleting unreachable objects. However, it makes no provision for the equivalent of a C++ Destructor function that should be called when the object is eliminated by the garbage collector. The equivalent of such methods in a garbage collected environment is normally called a *finalizer*.

ECL includes a simple implementation of finalizers which makes the following promises.

- The finalizer can be any lisp function, let it be compiled or interpreter.
- Finalizers are not invoked during garbage collection. Instead, if an unreachable object is found to have an associated finalizer, it is pushed into a list and *before the next garbage collection cycle*, the finalizer will be invoked.
- If the finalizer is invoked and it makes the object reachable, for instance, by assigning it to a variable, it will not be destroyed, but it will have no longer a finalizer associated to it.
- ECL will strive to call finalizers before the environment is closed and the program is finished, but this mechanism may fail when exiting in a non ordinary way.

The implementation is based on two functions, `ext:set-finalizer` and `ext:get-finalizer`, which allow setting and querying the finalizer functions for certain objects.

3.6.6 Memory Management Reference

Reference

`ext:stack-overflow` [Condition]

Stack overflow condition

Class **Precedence** **List** `ext:stack-overflow,` `storage-condition,`
`serious-condition, condition, t`

Methods

`ext:stack-overflow-size` *condition* [Function]

returns A non-negative integer.

`ext:stack-overflow-type` *condition* [Function]

returns A symbol from Table 3.1, except `ext:heap-size`.

Description This condition is signaled when one of the stack limits in Table 3.1 are violated or dangerously approached. It can be handled by resetting the limits and continuing, or jumping to an outer control point.

`ext:storage-exhausted` [Condition]

Memory overflow condition

Class **Precedence** **List** `ext:storage-exhausted,` `storage-condition,`
`serious-condition, condition, t`

Description This condition is signaled when ECL exhausts the `ext:heap-size` limit from Table 3.1. In handling this condition ECL follows this logic:

- If the heap size limit was set to 0 (that is no limit), but there is some free space in the safety region ECL frees this space and issues a non-restartable error. The user may jump to an outer point or quit.
- If the heap size had a finite limit, ECL offers the user the chance to resize it, issuing a restartable condition. The user may at this point use (`ext:set-limit 'ext:heap-size 0`) to remove the heap limit and avoid further messages, or use the (`continue`) restart to let ECL enlarge the heap by some amount.
- Independently of the heap size limit, if ECL finds that there is no space to free or to grow, ECL simply quits. There will be no chance to do some cleanup because there is no way to allocate any additional data.

`ext:get-finalizer` *object* [Function]

object Any lisp object.

Description This function returns the finalizer associated to an object, or `nil`.

`ext:get-limit` *concept* [Function]

concept A symbol.

Description Queries the different memory and stack limits that condition ECL's behavior. The value to be queried is denoted by the symbol *concept*, which should be one from the list: Table 3.1

ext:set-finalizer *object function* [Function]
 Associate a finalizer to an object.

object Any lisp object.

function A function or closure that takes one argument or `nil`.

Description If *function* is `nil`, no finalizer is associated to the object. Otherwise *function* must be a function or a closure of one argument, which will be invoked before the object is destroyed.

Example Close a file associated to an object.

```
(defclass my-class () ((file :initarg :file :initform nil)))

(defun finalize-my-class (x)
  (let ((s (slot-value x 'file)))
    (when s (format t "~%;;; Closing" s) (close s))))

(defmethod initialize-instance :around ((my-instance my-class) &rest args)
  (ext:set-finalizer my-instance #'finalize-my-class)
  (call-next-method))

(progn
  (make-instance 'my-class :file (open "~/ecl.old" :direction :input))
  nil)

(si::gc t)
(si::gc t)

;; Closing
```

ext:set-limit *concept value* [Function]
 Set a memory or stack limit.

concept A symbol.

value A positive integer.

Changes the different memory and stack limits that condition ECL's behavior. The value to be changed is denoted by the symbol *concept*, while the *value* is the new maximum size. The valid symbols and units are listed in Table 3.1.

Note that the limit has to be positive, but it may be smaller than the previous value of the limit. However, if the supplied value is smaller than what ECL is using at the moment, the new value will be silently ignored.

Concept	Units	Default	Command line
ext:frame-stack	Nested frames	2048	<code>--frame-stack</code>
ext:binding-stack	Bindings	8192	
ext:c-stack	Bytes	128 kilobytes	<code>--c-stack</code>
ext:heap-size	Bytes	256 megabytes	<code>--heap-size</code>
ext:lisp-stack	Bytes	32 kilobytes	<code>--lisp-stack</code>

Table 3.1: Customizable memory limits

3.7 Meta-Object Protocol (MOP)

3.7.1 Introduction

The Meta-Object Protocol is an extension to Common Lisp which provides rules, functions and a type structure to handle the object system. It is a reflective system, where classes are also objects and can be created and manipulated using very well defined procedures.

The Meta-Object Protocol associated to Common Lisp's object system was introduced in a famous book, The Art of the Metaobject Protocol AMOP [AMOP, see [Bibliography], page 193], which was probably intended for the ANSI [ANSI, see [Bibliography], page 193] specification but was dropped because of its revolutionary and then not too well tested ideas.

The AMOP is present, in one way or another, in most Common Lisp implementations, either using proprietary systems or because their implementation of CLOS descended from PCL (Portable CommonLoops). It has thus become a de facto standard and ECL should not be without it.

Unfortunately ECL's own implementation originally contained only a subset of the AMOP. This was a clever decision at the time, since the focus was on performance and on producing a stable and lean implementation of Common Lisp. Nowadays it is however not an option, especially given that most of the AMOP can be implemented with little cost for both the implementor and the user.

So ECL has an almost complete implementation of the AMOP. However, since it was written from scratch and progressed according to user's request and our own innovations, there might still be some missing functionality which we expect to correct in the near future. Please report any feature you miss as a bug through the appropriate channels.

3.8 Gray Streams

3.9 Tree walker

3.10 Local package nicknames

3.10.1 Overview

ECL allows giving packages local nicknames: they allow short and easy-to-use names to be used without fear of name conflict associated with normal nicknames.

A local nickname is valid only when inside the package for which it has been specified. Different packages can use same local nickname for different global names, or different local nickname for same global name.

The keyword `:package-local-nicknames` in `*features*` indicates the support for this feature.

3.10.2 Package local nicknames dictionary

`cl:defpackage name [[options]]*` [Macro]

Options are extended to include

`:local-nicknames (local-nickname actual-package-name)*`

The package has the specified local nicknames for the corresponding actual packages.

Example:

```
(defpackage :bar (:intern "X"))
(defpackage :foo (:intern "X"))
(defpackage :quux (:use :cl) (:local-nicknames (:bar :foo) (:foo :bar)))
(find-symbol "X" :foo) ; => FOO::X
(find-symbol "X" :bar) ; => BAR::X
(let ((*package* (find-package :quux)))
  (find-symbol "X" :foo)) ; => BAR::X
(let ((*package* (find-package :quux)))
  (find-symbol "X" :bar)) ; => FOO::X
```

`ext:package-local-nicknames package-designator` [Function]

`cl_object si_package_local_nicknames (cl_object package-designator)` [Function]

Returns an alist of (`local-nickname` . `actual-package`) describing the nicknames local to the designated package.

When in the designated package, calls to `find-package` with any of the local-nicknames will return the corresponding actual-package instead. This also affects all implied calls to `find-package`, including those performed by the reader.

When printing a package prefix for a symbol with a package local nickname, the local nickname is used instead of the real name in order to preserve print-read consistency.

`ext:package-locally-nicknamed-by-list package-designator` [Function]

`cl_object si_package_locally_nicknamed_by_list (cl_object package-designator)` [Function]

Returns a list of packages which have a local nickname for the designated package.

`ext:add-package-local-nickname local-nickname actual-package &optional package-designator` [Function]

`cl_object si_add_package_local_nickname (cl_object local-nickname, cl_object actual-package, cl_object package-designator)` [Function]

Adds *local-nickname* for *actual-package* in the designated package, defaulting to current package. *local-nickname* must be a string designator, and *actual-package* must be a package designator.

Returns the designated package.

Signals a continuable error if *local-nickname* is already a package local nickname for a different package.

When in the designated package, calls to `find-package` with the *local-nickname* will return the package the designated *actual-package* instead. This also affects all implied calls to `find-package`, including those performed by the reader.

When printing a package prefix for a symbol with a package local nickname, the local nickname is used instead of the real name in order to preserve print-read consistency.

`ext:remove-package-local-nickname` *old-nickname* &optional *package-designator* [Function]

`cl_object` `si_remove_package_local_nickname` (*cl_object* *old_nickname*, *cl_object* *package_designator*) [Function]

If the designated package had *old-nickname* as a local nickname for another package, it is removed. Returns true if the nickname existed and was removed, and `nil` otherwise.

3.11 Package locks

3.11.1 Package Locking Overview

ECL borrows parts of the protocol and documentation from SBCL for compatibility. Interface is the same except that the home package for locking is `ext` and that ECL doesn't implement Implementation Packages and a few constructs. To load the extension you need to require `package-locks`:

```
(require '#:package-locks)
```

Package locks protect against unintentional modifications of a package: they provide similar protection to user packages as is mandated to `common-lisp` package by the ANSI specification. They are not, and should not be used as, a security measure.

Newly created packages are by default unlocked (see the `:lock` option to `defpackage`).

The package `common-lisp` and ECL internal implementation packages are locked by default, including `ext`.

It may be beneficial to lock `common-lisp-user` as well, to ensure that various libraries don't pollute it without asking, but this is not currently done by default.

3.11.2 Operations Violating Package Locks

The following actions cause a package lock violation if the package operated on is locked, and `*package*` is not an implementation package of that package, and the action would cause a change in the state of the package (so e.g. exporting already external symbols is never a violation). Package lock violations caused by these operations signal errors of type `package-error`.

1. Shadowing a symbol in a package.
2. Importing a symbol to a package.
3. Uninterning a symbol from a package.
4. Exporting a symbol from a package.

5. Unexporting a symbol from a package.
6. Changing the packages used by a package.
7. Renaming a package.
8. Deleting a package.
9. Attempting to redefine a function in a locked package.
10. Adding a new package local nickname to a package.
11. Removing an existing package local nickname to a package.

3.11.3 Package Lock Dictionary

ext:package-locked-p *package* [Function]
 Returns *t* when *package* is locked, *nil* otherwise. Signals an error if *package* doesn't designate a valid package.

ext:lock-package *package* [Function]
 Locks *package* and returns *t*. Has no effect if package was already locked. Signals an error if package is not a valid *package* designator

ext:unlock-package *package* [Function]
 Unlocks *package* and returns *t*. Has no effect if *package* was already unlocked. Signals an error if *package* is not a valid package designator.

ext:without-package-locks **&body** *body* [Macro]
 Ignores all runtime package lock violations during the execution of *body*. *Body* can begin with declarations.

ext:with-unlocked-packages (**&rest** *packages*) **&body** *body* [Macro]
 Unlocks *packages* for the dynamic scope of the *body*. Signals an error if any of *packages* is not a valid package designator.

cl:defpackage *name* **[[option]]*** \Rightarrow *package* [Macro]
 Options are extended to include

:lock *boolean*

If the argument to **:lock** is *t*, the package is initially locked. If **:lock** is not provided it defaults to *nil*.

Example:

```
(defpackage "F00" (:export "BAR") (:lock t))
```

;;; is equivalent to

```
(defpackage "F00") (:export "BAR")
(lock-package "F00")
```

3.12 CDR Extensions

ECL currently implements the following specifications of the Common Lisp Document Repository (<https://common-lisp.net/project/cdr/>):

Number	Comments
1	In <code>clos</code> package; partial implementation, see Section 3.7 [Meta-Object Protocol (MOP)], page 143,
5	In <code>ext</code> package
7	Only if ECL is compiled with <code>--with-cmuformat</code> configure option
14	

Table 3.2: Implemented CDR extensions

4 Developer's guide

4.1 Sources structure

4.1.1 src/c

<code>alloc_2.d</code>	memory allocation based on the Boehm GC
<code>all_symbols.d</code>	name mangler and symbol initialization
<code>apply.d</code>	interface to C call mechanism
<code>arch/*</code>	architecture dependant code
<code>array.d</code>	array routines
<code>assignment.c</code>	assignment
<code>backq.d</code>	backquote mechanism
<code>big.d</code>	bignum routines based on the GMP
<code>big_ll.d</code>	bignum emulation with long long
<code>cfun.d</code>	compiled functions
<code>cfun_ dispatch.d</code>	trampolines for functions
<code>character.d</code>	character routines
<code>char_ctype.d</code>	character properties.
<code>cinit.d</code>	lisp initialization
<code>clos/accessor.d</code>	dispatch for slots
<code>clos/cache.d</code>	thread-local cache for a variety of operations
<code>clos/gfun.d</code>	dispatch for generic functions
<code>clos/instance.d</code>	CLOS interface
<code>cmpaux.d</code>	auxiliaries used in compiled Lisp code
<code>compiler.d</code>	bytecode compiler

<code>cons.d</code>	list manipulation macros & functions (auto generated)
<code>disassembler.d</code>	bytecodes disassembler utilities
<code>dpp.c</code>	defun preprocessor
<code>ecl_</code> <code>constants.h</code>	constant values for <code>all_symbols.d</code>
<code>ecl_features.h</code>	names of features compiled into ECL
<code>error.d</code>	error handling
<code>eval.d</code>	evaluation
<code>ffi/backtrace.d</code>	C backtraces
<code>ffi/cdata.d</code>	data for compiled files
<code>ffi/libraries.d</code>	shared library and bundle opening / copying / closing
<code>ffi/mmap.d</code>	mapping of binary files
<code>ffi.d</code>	user defined data types and foreign functions interface
<code>file.d</code>	file interface (implementation dependent)
<code>format.d</code>	format (this isn't ANSI compliant, we need it for bootstrapping though)
<code>hash.d</code>	hash tables
<code>interpreter.d</code>	bytecode interpreter
<code>iso_latin_</code> <code>names.h</code>	character names in ISO-LATIN-1
<code>list.d</code>	list manipulating routines
<code>load.d</code>	binary loader (contains also <code>open_fasl_data</code>)
<code>macros.d</code>	macros and environment
<code>main.d</code>	ecl boot process
<code>Makefile.in</code>	Makefile for ECL core library
<code>mapfun.d</code>	mapping

<code>multival.d</code>	multiple values
<code>newhash.d</code>	hashing routines
<code>num_arith.d</code>	arithmetic operations
<code>number.d</code>	constructing numbers
<code>numbers/*.d</code>	arithmetic operations (abs, atan, plusp etc)
<code>num_co.d</code>	operations on floating-point numbers (implementation dependent)
<code>num_log.d</code>	logical operations on numbers
<code>num_pred.d</code>	predicates on numbers
<code>num_rand.d</code>	random numbers
<code>package.d</code>	packages (OS dependent)
<code>pathname.d</code>	pathnames
<code>predicate.d</code>	predicates
<code>print.d</code>	print
<code>printer/*.d</code>	printer utilities and object representations
<code>read.d</code>	reader
<code>reader/parse_</code> <code>integer.d</code> <code>reader/parse_</code> <code>number.d</code>	
<code>reference.d</code>	reference in Constants and Variables
<code>sequence.d</code>	sequence routines
<code>serialize.d</code>	serialize a bunch of lisp data
<code>sse2.d</code>	SSE2 vector type support
<code>stacks.d</code>	binding/history/frame stacks

<code>string.d</code>	string routines
<code>structure.d</code>	structure interface
<code>symbol.d</code>	symbols
<code>symbols_list.h</code> <code>symbols_</code> <code>list2.h</code>	The latter is generated from the first. The first has to contain all symbols on the system which aren't local.
<code>tcp.d</code>	stream interface to TCP
<code>time.d</code>	time routines
<code>typespec.d</code>	type specifier routines
<code>unicode/*</code>	unicode definitions
<code>unixfsys.d</code>	Unix file system interface
<code>unixint.d</code>	Unix interrupt interface.
<code>unixsys.d</code>	Unix shell interface
<code>vector_push.d</code>	vector optimizations
<code>threads/atomic.d</code>	atomic operations
<code>threads/barrier.d</code>	wait barriers
<code>threads/condition</code> <code>variable.d</code>	condition variables for native threads
<code>threads/mailbox.d</code>	thread communication queue
<code>threads/mutex.d</code>	mutually exclusive locks.
<code>threads/process.d</code>	native threads
<code>threads/queue.d</code>	waiting queue for threads
<code>threads/rwlock.d</code>	POSIX read-write locks
<code>threads/semaphore.d</code>	POSIX-like semaphores

4.2 Contributing

4.3 Defun preprocessor

The defun preprocessor allows for convenient definition of Lisp functions with optional and keyword arguments and the use of Lisp symbols in ECL's own C source code. It generates the C code necessary to use optional function arguments and to access symbols in ECL's builtin symbol table.

Usage:

```
dpp [in-file [out-file]]
```

The file named in-file is preprocessed and the output will be written to the file whose name is out-file. If in-file is "-" program is read from standard input, while if out-file is "-" C-program is written to standard output.

The function definition:

```
@(defun name ({var}*[
    [&optional {var | (var [initform [svar]])}]*)
    [&rest var]
    [&key {var |
        ({var | (keyword var)} [initform [svar]])}]*
    [&allow_other_keys]
    [&aux {var | (var [initform])}]*)
```

C-declaration

```
@ {
```

C-body

```
} @)
```

name is the name of the lisp function

&optional may be abbreviated as &o.

&rest may be abbreviated as &r.

&key may be abbreviated as &k.

&allow_other_keys may be abbreviated as &aok.

&aux may be abbreviated as &a.

Each variable becomes a C variable.

Each supplied-p parameter becomes a boolean C variable.

Initforms are C expressions. If an expression contains non-alphanumeric characters, it should be surrounded by backquotes (`).

Function return:

```
@(return {form}*);
```

Return function expands into a lexical block {}, so if it's used inside if/else, then it should be enclosed, even if we use sole @(return);, because ";" will be treated as the next instruction.

Symbols:

```
@'name'
```

Expands into a C statement, whole value is the given symbol from `symbols_list.h`

`@[name]`

Expands into a C statement, whole value is a fixnum corresponding to the index in the builtin symbols table of the given symbol from `symbols_list.h`. Used for handling type errors.

4.4 Manipulating Lisp objects

If you want to extend, fix or simply customize ECL for your own needs, you should understand how the implementation works.

`cl_lispunion` *cons big ratio SF DF longfloat gencomplex* [C/C++ identifier]
csfloat cdfloat clfloat symbol pack hash array vector base_string string
stream random readtable pathname bytcodes bclosure cfun cfunfixed
cclosure d instance process queue lock rwlock condition_variable
semaphore barrier mailbox cblock foreign frame weak sse
 Union containing all first-class ECL types.

4.4.1 Objects representation

In ECL a lisp object is represented by a type called `cl_object`. This type is a word which is long enough to host both an integer and a pointer. The least significant bits of this word, also called the tag bits, determine whether it is a pointer to a C structure representing a complex object, or whether it is an immediate data, such as a fixnum or a character.

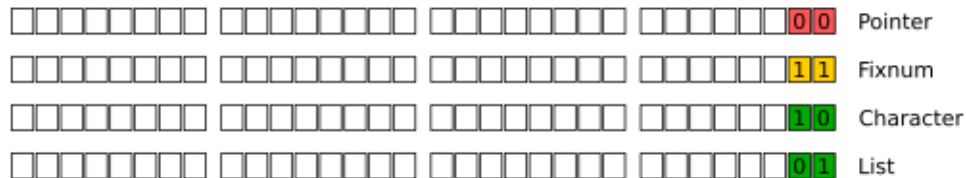


Figure 4.1: Immediate types

The topic of the immediate values and bit fiddling is nicely described in Peter Bex's blog (<http://www.more-magic.net/posts/internals-data-representation.html>) describing Chicken Scheme (<http://www.call-cc.org/>) internal data representation. We could borrow some ideas from it (like improving `fixnum` bitness and providing more immediate values). All changes to code related to immediate values should be carefully **benchmarked**.

The `fixnums` and characters are called immediate data types, because they require no more than the `cl_object` datatype to store all information. All other ECL objects are non-immediate and they are represented by a pointer to a cell that is allocated on the heap. Each cell consists of several words of memory and contains all the information related to that object. By storing data in multiples of a word size, we make sure that the least significant bits of a pointer are zero, which distinguishes pointers from immediate data.

In an immediate datatype, the tag bits determine the type of the object. In non-immediate datatypes, the first byte in the cell contains the secondary type indicator, and distinguishes

between different types of non immediate data. The use of the remaining bytes differs for each type of object. For instance, a cons cell consists of three words:

```
+-----+-----+
| CONS   |         |
+-----+-----+
|   car-pointer   |
+-----+-----+
|   cdr-pointer   |
+-----+-----+
```

Note, that this is one of the possible implementations of `cons`. The second one (currently default) uses the immediate value for the `list` and consumes two words instead of three. Such implementation is more memory and speed efficient (according to the comments in the source code):

```
/*
 * CONSES
 *
 * We implement two variants. The "small cons" type carries the type
 * information in the least significant bits of the pointer. We have
 * to do some pointer arithmetics to find out the CAR / CDR of the
 * cons but the overall result is faster and memory efficient, only
 * using two words per cons.
 *
 * The other scheme stores conses as three-words objects, the first
 * word carrying the type information. This is kept for backward
 * compatibility and also because the oldest garbage collector does
 * not yet support the smaller datatype.
 *
 * To make code portable and independent of the representation, only
 * access the objects using the common macros below (that is all
 * except ECL_CONS_PTR or ECL_PTR_CONS).
 */
```

`cl_object` [C/C++ identifier]

This is the type of a lisp object. For your C/C++ program, a `cl_object` can be either a fixnum, a character, or a pointer to a union of structures (See `cl_lispunion` in the header `object.h`). The actual interpretation of that object can be guessed with the macro `ecl_t_of`.

Example

For example, if `x` is of type `cl_object`, and it is of type `fixnum`, we may retrieve its value:

```
if (ecl_t_of(x) == t_fixnum)
    printf("Integer value: %d\n", ecl_fixnum(x));
```

Example

If `x` is of type `cl_object` and it does not contain an immediate datatype, you may inspect the cell associated to the lisp object using `x` as a pointer. For example:

```
if (ecl_t_of(x) == t_vector)
    printf("Vector's dimension is: %d\n", x->vector.dim);
```

You should see the following sections and the header `object.h` to learn how to use the different fields of a `cl_object` pointer.

`cl_type` [C/C++ identifier]

Enumeration type which distinguishes the different types of lisp objects. The most important values are:

`t_cons`, `t_fixnum`, `t_character`, `t_bignum`, `t_ratio`, `t_singlefloat`, `t_doublefloat`, `t_complex`, `t_symbol`, `t_package`, `t_hashtable`, `t_array`, `t_vector`, `t_string`, `t_bitvector`, `t_stream`, `t_random`, `t_readtable`, `t_pathname`, `t_bytecodes`, `t_cfun`, `t_cclosure`, `t_gfun`, `t_instance`, `t_foreign` and `t_thread`.

`cl_type ecl_t_of (cl_object x)` [Function]

If `x` is a valid lisp object, `ecl_t_of(x)` returns an integer denoting the type that lisp object. That integer is one of the values of the enumeration type `cl_type`.

<code>bool ECL_CHARACTERP (cl_object o)</code>	[Function]
<code>bool ECL_BASE_CHAR_P (cl_object o)</code>	[Function]
<code>bool ECL_BASE_CHAR_CODE_P (ecl_character o)</code>	[Function]
<code>bool ECL_NUMBER_TYPE_P (cl_object o)</code>	[Function]
<code>bool ECL_COMPLEXP (cl_object o)</code>	[Function]
<code>bool ECL_REAL_TYPE_P (cl_object o)</code>	[Function]
<code>bool ECL_FIXNUMP (cl_object o)</code>	[Function]
<code>bool ECL_BIGNUMP (cl_object o)</code>	[Function]
<code>bool ECL_SINGLE_FLOAT_P (cl_object o)</code>	[Function]
<code>bool ECL_DOUBLE_FLOAT_P (cl_object o)</code>	[Function]
<code>bool ECL_LONG_FLOAT_P (cl_object o)</code>	[Function]
<code>bool ECL_CONSP (cl_object o)</code>	[Function]
<code>bool ECL_LISTP (cl_object o)</code>	[Function]
<code>bool ECL_ATOM (cl_object o)</code>	[Function]
<code>bool ECL_SYMBOLP (cl_object o)</code>	[Function]
<code>bool ECL_ARRAYP (cl_object o)</code>	[Function]
<code>bool ECL_VECTORP (cl_object o)</code>	[Function]
<code>bool ECL_BIT_VECTOR_P (cl_object o)</code>	[Function]
<code>bool ECL_STRINGP (cl_object o)</code>	[Function]
<code>bool ECL_HASH_TABLE_P (cl_object o)</code>	[Function]
<code>bool ECL_RANDOM_STATE_P (cl_object o)</code>	[Function]
<code>bool ECL_PACKAGEP (cl_object o)</code>	[Function]
<code>bool ECL_PATHNAMEP (cl_object o)</code>	[Function]
<code>bool ECL_READTABLEP (cl_object o)</code>	[Function]
<code>bool ECL_FOREIGN_DATA_P (cl_object o)</code>	[Function]

bool ECL_SSE_PACK_P (*cl-object o*) [Function]
 Different macros that check whether *o* belongs to the specified type. These checks have been optimized, and are preferred over several calls to `ecl_t_of`.

bool ECL_IMMEDIATE (*cl-object o*) [Function]
 Tells whether *x* is an immediate datatype.

4.4.2 Constructing objects

On each of the following sections we will document the standard interface for building objects of different types. For some objects, though, it is too difficult to make a C interface that resembles all of the functionality in the lisp environment. In those cases you need to

1. build the objects from their textual representation, or
2. use the evaluator to build these objects.

The first way makes use of a C or Lisp string to construct an object. The two functions you need to know are the following ones.

si::string-to-object *string* &optional (*err-value nil*) [Function]

cl_object si_string_to_object (*cl_narg nargs*, *cl-object str*, ...) [Function]

cl_object ecl_read_from_cstring (*const char *s*) [Function]

`ecl_read_from_cstring` builds a lisp object from a C string which contains a suitable representation of a lisp object. `si_string_to_object` performs the same task, but uses a lisp string, and therefore it is less useful.

- **DEPRECATED** `c_string_to_object` – equivalent to `ecl_read_from_cstring`

Example

Using a C string

```
cl_object array1 = ecl_read_from_cstring("#(1 2 3 4)");
```

Using a Lisp string

```
cl_object string = make_simple_base_string("#(1 2 3 4)");
cl_object array2 = si_string_to_object(string);
```

Integers

Common-Lisp distinguishes two types of integer types: **bignums** and **fixnums**. A **fixnum** is a small integer, which ideally occupies only a word of memory and which is between the values `MOST-NEGATIVE-FIXNUM` and `MOST-POSITIVE-FIXNUM`. A **bignum** is any integer which is not a **fixnum** and it is only constrained by the amount of memory available to represent it.

In ECL a **fixnum** is an integer that, together with the tag bits, fits in a word of memory. The size of a word, and thus the size of a **fixnum**, varies from one architecture to another, and you should refer to the types and constants in the `ecl.h` header to make sure that your C extensions are portable. All other integers are stored as **bignums**, they are not immediate objects, they take up a variable amount of memory and the GNU Multiprecision Library is required to create, manipulate and calculate with them.

cl_fixnum [C/C++ identifier]

This is a C signed integer type capable of holding a whole **fixnum** without any loss of precision. The opposite is not true, and you may create a **cl_fixnum** which exceeds the limits of a **fixnum** and should be stored as a **bignum**.

cl_index [C/C++ identifier]

This is a C unsigned integer type capable of holding a non-negative **fixnum** without loss of precision. Typically, a **cl_index** is used as an index into an array, or into a proper list, etc.

MOST_NEGATIVE_FIXNUM [Constant]

MOST_POSITIVE_FIXNUM [Constant]

These constants mark the limits of a **fixnum**.

bool ecl_fixnum_lower (*cl_fixnum a*, *cl_fixnum b*) [Function]

bool ecl_fixnum_greater (*cl_fixnum a*, *cl_fixnum b*) [Function]

bool ecl_fixnum_leq (*cl_fixnum a*, *cl_fixnum b*) [Function]

bool ecl_fixnum_geq (*cl_fixnum a*, *cl_fixnum b*) [Function]

bool ecl_fixnum_plus (*cl_fixnum a*) [Function]

bool ecl_fixnum_minus (*cl_fixnum a*) [Function]

Operations on **fixnums** (comparison and predicates).

cl_object ecl_make_fixnum (*cl_fixnum n*) [Function]

cl_fixnum ecl_fixnum (*cl_object o*) [Function]

ecl_make_fixnum converts from an integer to a lisp object, while the **ecl_fixnum** does the opposite (converts lisp object **fixnum** to integer). These functions do **not** check their arguments.

- **DEPRECATED MAKE_FIXNUM** – equivalent to **ecl_make_fixnum**
- **DEPRECATED fix** – equivalent to **ecl_fixnum**

cl_fixnum fixint (*cl_object o*) [Function]

cl_index fixnint (*cl_object o*) [Function]

Safe conversion of a lisp **fixnum** to a C integer of the appropriate size. Signals an error if *o* is not of **fixnum** type.

fixnint additionally ensure that *o* is not negative.

Characters

ECL has two types of characters – one fits in the C type **char**, while the other is used when ECL is built with a configure option **--enable-unicode** which defaults to 32 (characters are stored in 32bit variable and codepoints have 21-bits).

ecl_character [C/C++ identifier]

Immediate type **t_character**. If ECL built with Unicode support, then may be either base or extended character, which may be distinguished with the predicate **ECL_BASE_CHAR_P**.

Additionally we have **ecl_base_char** for base strings, which is an equivalent to the ordinary **char**.

Example

```
if (ECL_CHARACTERP(o) && ECL_BASE_CHAR_P(o))
    printf("Base character: %c\n", ECL_CHAR_CODE(o));
```

ECL_CHAR_CODE_LIMIT [Constant]

Each character is assigned an integer code which ranges from 0 to (ECL_CHAR_CODE_LIMIT-1).

- **DEPRECATED** CHAR_CODE_LIMIT – equivalent to ECL_CHAR_CODE_LIMIT

cl_object ECL_CODE_CHAR (*cl_character o*) [Function]

ecl_character ECL_CHAR_CODE (*cl_object o*) [Function]

ecl_character ecl_char_code (*cl_object o*) [Function]

ecl_base_char ecl_base_char_code (*cl_object o*) [Function]

ECL_CHAR_CODE, ecl_char_code and ecl_base_char_code return the integer code associated to a lisp character. ecl_char_code and ecl_base_char_code perform a safe conversion, while ECL_CHAR_CODE doesn't check its argument.

ECL_CODE_CHAR returns the lisp character associated to an integer code. It does not check its arguments.

- **DEPRECATED** CHAR_CODE – equivalent to ECL_CHAR_CODE
- **DEPRECATED** CODE_CHAR – equivalent to ECL_CODE_CHAR

bool ecl_char_eq (*cl_object x, cl_object y*) [Function]

bool ecl_char_equal (*cl_object x, cl_object y*) [Function]

Compare two characters for equality. char_eq take case into account and char_equal ignores it.

int ecl_char_cmp (*cl_object x, cl_object y*) [Function]

int ecl_char_compare (*cl_object x, cl_object y*) [Function]

Compare the relative order of two characters. ecl_char_cmp takes care of case and ecl_char_compare converts all characters to uppercase before comparing them.

Arrays

An array is an aggregate of data of a common type, which can be accessed with one or more non-negative indices. ECL stores arrays as a C structure with a pointer to the region of memory which contains the actual data. The cell of an array datatype varies depending on whether it is a vector, a bit-vector, a multidimensional array or a string.

bool ECL_ADJUSTABLE_ARRAY_P (*cl_object x*) [Function]

bool ECL_ARRAY_HAS_FILL_POINTER_P (*cl_object x*) [Function]

All arrays (arrays, strings and bit-vectors) may be tested for being adjustable and whenever they have a fill pointer with this two macros. They don't check the type of their arguments.

ecl_vector [C/C++ identifier]

If *x* contains a vector, you can access the following fields:

x->vector.eltype

The type of the elements of the vector.

`x->vector.displaced`
List storing the vectors that `x` is displaced from and that `x` displaces to.

`x->vector.dim`
The maximum number of elements.

`x->vector.fillp`
Actual number of elements in the vector or `fill pointer`.

`x->vector.self`
Union of pointers of different types. You should choose the right pointer depending on `x->vector.eltype`.

`ecl_array` [C/C++ identifier]
If `x` contains a multidimensional array, you can access the following fields:

`x->array.eltype`
The type of the elements of the array.

`x->array.rank`
The number of array dimensions.

`x->array.displaced`
List storing the arrays that `x` is displaced from and that `x` displaces to.

`x->array.dim`
The maximum number of elements.

`x->array.dims[]`
Array with the dimensions of the array. The elements range from `x->array.dim[0]` to `x->array.dim[x->array.rank-1]`.

`x->array.fillp`
Actual number of elements in the array or `fill pointer`.

`x->array.self`
Union of pointers of different types. You should choose the right pointer depending on `x->array.eltype`.

`cl_eltype` `ecl_aet_object` `ecl_aet_sf` `ecl_aet_df` `ecl_aet_lf` [C/C++ identifier]
`ecl_aet_csf` `ecl_aet_cdf` `ecl_aet_clf` `ecl_aet_bit` `ecl_aet_fix` `ecl_aet_index`
`ecl_aet_b8` `ecl_aet_i8` `ecl_aet_b16` `ecl_aet_i16` `ecl_aet_b32` `ecl_aet_i32`
`ecl_aet_b64` `ecl_aet_i64` `ecl_aet_ch` `ecl_aet_bc`

Each array is of an specialized type which is the type of the elements of the array. ECL has arrays only a few following specialized types, and for each of these types there is a C integer which is the corresponding value of `x->array.eltype` or `x->vector.eltype`. We list some of those types together with the C constant that denotes that type:

`t` `ecl_aet_object`

`single-float`
`ecl_aet_sf`


```

double-float      ecl_aet_df
long-float        ecl_aet_lf
(COMPLEX SINGLE-FLOAT)
                  ecl_aet_csf
(COMPLEX DOUBLE-FLOAT)
                  ecl_aet_cdf
(COMPLEX LONG-FLOAT)
                  ecl_aet_clf
BIT               ecl_aet_bit
FIXNUM            ecl_aet_fix
INDEX             ecl_aet_index
CHARACTER         ecl_aet_ch
BASE-CHAR         ecl_aet_bc

```

`cl_elttype ecl_array_elttype (cl-object array)` [Function]
 Returns the element type of the array `o`, which can be a string, a bit-vector, vector, or a multidimensional array.

Example

For example, the code

```

ecl_array_elttype(ecl_read_from_cstring("\\"AAA\\"")); /* returns ecl_aet_ch */
ecl_array_elttype(ecl_read_from_cstring("#(A B C)")); /* returns ecl_aet_object */

```

`cl_object ecl_aref (cl-object x, cl-index index)` [Function]
`cl_object ecl_aset (cl-object x, cl-index index, cl-object value)` [Function]

These functions are used to retrieve and set the elements of an array. The elements are accessed with one index, `index`, as in the lisp function ROW-MAJOR-AREF.

Example

```

cl_object array = ecl_read_from_cstring("#2A((1 2) (3 4))");
cl_object x = ecl_aref(array, 3);
cl_print(1, x); /* Outputs 4 */
ecl_aset(array, 3, ecl_make_fixnum(5));
cl_print(1, array); /* Outputs #2A((1 2) (3 5)) */

```

`cl_object ecl_aref1 (cl-object x, cl-index index)` [Function]
`cl_object ecl_aset1 (cl-object x, cl-index index, cl-object value)` [Function]

These functions are similar to `aref` and `aset`, but they operate on vectors.

Example

```
cl_object array = ecl_read_from_cstring("#(1 2 3 4)");
cl_object x = ecl_aref1(array, 3);
cl_print(1, x);      /* Outputs 4 */
ecl_aset1(array, 3, ecl_make_fixnum(5));
cl_print(1, array); /* Outputs #(1 2 3 5) */
```

Strings

A string, both in Common-Lisp and in ECL is nothing but a vector of characters. Therefore, almost everything mentioned in the section of arrays remains valid here.

The only important difference is that ECL stores the base-strings (non-Unicode version of a string) as a lisp object with a pointer to a zero terminated C string. Thus, if a string has *n* characters, ECL will reserve *n*+1 bytes for the base-string. This allows us to pass the base-string self pointer to any C routine.

`ecl_string` [C/C++ identifier]
`ecl_base_string` [C/C++ identifier]

If *x* is a lisp object of type string or a base-string, we can access the following fields:

```
x->string.dim x->base_string.dim
    Actual number of characters in the string.

x->string.fillp x->base_string.fillp
    Actual number of characters in the string.

x->string.self x->base_string.self
    Pointer to the characters (appropriately ecl_character's and ecl_base_char's).
```

`bool ECL_EXTENDED_STRING_P (cl_object object)` [Function]
`bool ECL_BASE_STRING_P (cl_object object)` [Function]
 Verifies if an objects is an extended or base string. If Unicode isn't supported, then `ECL_EXTENDED_STRING_P` always returns 0.

Bit-vectors

Bit-vector operations are implemented in file `src/c/array.d`. Bit-vector shares the structure with a vector, therefore, almost everything mentioned in the section of arrays remains valid here.

Streams

Streams implementation is a broad topic. Most of the implementation is done in the file `src/c/file.d`. Stream handling may have different implementations referred by a member pointer `ops`.

Additionally on top of that we have implemented *Gray Streams* (in portable Common Lisp) in file `src/clos/streams.lsp`, which may be somewhat slower (we need to benchmark it!). This implementation is in a separate package *GRAY*. We may redefine functions in the *COMMON-LISP* package with a function `redefine-cl-functions` at run-time.

`ecl_file_ops` *write_* read_* unread_* peek_* listen* [C/C++ identifier]
clear_input clear_output finish_output force_output input_p output_p
interactive_p element_type length get_position set_position column close

`ecl_stream` [C/C++ identifier]

`ecl_smmode` `mode`
Stream mode (in example `ecl_smm_string_input`).

`int` `closed`
Whenever stream is closed or not.

`ecl_file_ops` `*ops`
Pointer to the structure containing operation implementations (dispatch table).

`union` `file`
Union of ANSI C streams (`FILE *stream`) and POSIX files interface (`cl_fixnum` descriptor).

`cl_object` `object0, object1`
Some objects (may be used for a specific implementation purposes).

`cl_object` `byte_stack`
Buffer for unread bytes.

`cl_index` `column`
File column.

`cl_fixnum` `last_char`
Last character read.

`cl_fixnum` `last_code[2]`
Actual composition of the last character.

`cl_fixnum` `int0 int1`
Some integers (may be used for a specific implementation purposes).

`cl_index` `byte_size`
Size of byte in binary streams.

`cl_fixnum` `last_op`
0: unknown, 1: reading, -1: writing

`char *``buffer`
Buffer for FILE

`cl_object` `format`
external format

`cl_eformat_encoder` `encoder`
`cl_eformat_encoder` `decoder`
`cl_object` `format_table`
in flags Character table, flags, etc

`ecl_character` `eof_character`

`bool ECL_ANSI_STREAM_P (cl_object o)` [Function]
 Predicate determining if `o` is a first-class stream object.

`bool ECL_ANSI_STREAM_TYPE_P (cl_object o, ecl_smmode m)` [Function]
 Predicate determining if `o` is a first-class stream object of type `m`.

Structures

Structures and instances share the same datatype `t_instance` (with a few exceptions. Structure implementation details are the file `src/c/structure.d`.

`cl_object ECL_STRUCT_TYPE (cl_object x)` [Function]
`cl_object ECL_STRUCT_SLOTS (cl_object x)` [Function]
`cl_object ECL_STRUCT_LENGTH (cl_object x)` [Function]
`cl_object ECL_STRUCT_SLOT (cl_object x, cl_index i)` [Function]
`cl_object ECL_STRUCT_NAME (cl_object x)` [Function]
 Convenience functions for the structures.

Instances

`cl_object ECL_CLASS_OF (cl_object x)` [Function]
`cl_object ECL_SPEC_FLAG (cl_object x)` [Function]
`cl_object ECL_SPEC_OBJECT (cl_object x)` [Function]
`cl_object ECL_CLASS_NAME (cl_object x)` [Function]
`cl_object ECL_CLASS_SUPERIORS (cl_object x)` [Function]
`cl_object ECL_CLASS_INFERIORS (cl_object x)` [Function]
`cl_object ECL_CLASS_SLOTS (cl_object x)` [Function]
`cl_object ECL_CLASS_CPL (cl_object x)` [Function]
`bool ECL_INSTANCEP (cl_object x)` [Function]
 Convenience functions for the structures.

Bytecodes

A bytecodes object is a lisp object with a piece of code that can be interpreted. The objects of type `t_bytecodes` are implicitly constructed by a call to `eval`, but can also be explicitly constructed with the `si_make_lambda` function.

`si:safe-eval form env &optional err-value` [Function]

`cl_object si_safe_eval (cl_narg nargs, cl_object form, cl_object env, ...)` [Function]

`si_safe_eval` evaluates `form` in the lexical environment¹ `env`, which can be `ECL_NIL`. Before evaluating it, the expression form is bytecompiled. If the form signals an error, or tries to jump to an outer point, the function has two choices: by default, it will invoke a debugger, but if a third value is supplied, then `si_safe_eval` will not use a debugger but rather return that value.

- **DEPRECATED** `cl_object cl_eval (cl_object form)` - `cl_eval` is the equivalent of `si_safe_eval` but without environment and with no `err-value` supplied. It exists only for compatibility with previous versions.

¹ Note that `env` must be a lexical environment as used in the interpreter, See Section 4.6.3 [The lexical environment], page 167

- **DEPRECATED** `cl_object cl_safe_eval (cl_object form, cl_object env, cl_object err_value)` - Equivalent of `si_safe_eval`.

Example

```
cl_object form = ecl_read_from_cstring("(print 1)");
si_safe_eval(2, form, ECL_NIL);
si_safe_eval(3, form, ECL_NIL, ecl_make_fixnum(3)); /* on error function will ret
```

`cl_object si_make_lambda (cl_object name, cl_object def)` [Function]
Builds an interpreted lisp function with name given by the symbol name and body given by `def`.

Example

For instance, we would achieve the equivalent of

```
(funcall #'(lambda (x y)
             (block foo (+ x y)))
 1 2)
```

with the following code

```
cl_object def = ecl_read_from_cstring("((x y) (+ x y))");
cl_object name = ecl_make_symbol("FOO", "COMMON-LISP-USER");
cl_object fun = si_make_lambda(name, def);
return cl_funcall(3, fun, ecl_make_fixnum(1), ecl_make_fixnum(2));
```

Notice that `si_make_lambda` performs a bytecodes compilation of the definition and thus it may signal some errors. Such errors are not handled by the routine itself so you might consider using `si_safe_eval` instead.

4.5 Environment implementation

4.6 The interpreter

4.6.1 ECL stacks

ECL uses the following stacks:

Frame Stack	consisting of catch, block, tagbody frames
Bind Stack	for shallow binding of dynamic variables
Interpreter Stack	acts as a Forth data stack, keeping intermediate arguments to interpreted functions, plus a history of called functions.
C Control Stack	used for arguments/values passing, typed lexical variables, temporary values, and function invocation.

4.6.2 Procedure Call Conventions

ECL employs standard C calling conventions to achieve efficiency and interoperability with other languages. Each Lisp function is implemented as a C function which takes as many arguments as the Lisp original. If the function takes optional or keyword arguments, the corresponding C function takes one additional integer argument which holds the number of actual arguments. The function sets `nvalues` in the thread local environment to the

number of Lisp values produced, it returns the first one and the remaining ones are kept in a global (per thread) array (`values`).

To show the argument/value passing mechanism, here we list the actual code for the Common-Lisp function `last`.

```
cl_object
cl_last(cl_narg nargs, cl_object l, ...)
{
    const cl_env_ptr the_env = ecl_process_env();
    cl_object k;
    va_list ARGS;
    va_start(ARGS, l);
    if (ecl_unlikely(narg < 1 || nargs > 2)) FEwrong_num_arguments(/* ... */);
    if (narg > 1) {
        k = va_arg(ARGS, cl_object);
    } else {
        k = ecl_make_fixnum(1);
    }
    cl_object __value0 = ecl_last(l, ecl_to_size(k));
    the_env->nvalues = 1;
    the_env->values[0] = __value0;
    va_end(ARGS);
    return __value0;
}
```

ECL adopts the convention that the name of a function that implements a Common-Lisp function begins with a short package name (`cl` for **COMMON-LISP**, `si` for **SYSTEM**, etc), followed by `L`, and followed by the name of the Common-Lisp function. (Strictly speaking, ‘-’ and ‘*’ in the Common-Lisp function name are replaced by ‘_’ and ‘A’, respectively, to obey the syntax of C.)

The code for the function `last` first checks that the right number of arguments are supplied to `cl_last`. That is, it checks that `narg` is 1 or 2, and otherwise, it causes an error. Following that, the optional variable `k` is initialized and the return value `__value0` is computed. The number assigned to `nvalues` set by the function (1 in this case) represents the number of values of the function. The return value of the function is copied in the `values` array as well as returned directly.

In general, if one is to play with the C kernel of ECL there is no need to know about all these conventions. There is a preprocessor (see Section 4.3 [Defun preprocessor], page 153) that takes care of the details, by using a lisp representation of the statements that output values, and of the function definitions. For instance, the actual source code for `cl_last` in `src/c/list.d` is

```
@(defun last (l &optional (k ecl_make_fixnum(1)))
@
    @(return ecl_last(l, ecl_to_size(k)));
@)
```

4.6.3 The lexical environment

The ECL interpreter uses a list containing local functions and macros, variables, tags and blocks to represent the lexical environment. When a function closure is created, the current lexical environment is saved in the closure along with the lambda expression. Later, when the closure is invoked, this list is used to recover the lexical environment.

Note that this list is different from what the Common Lisp standard calls a lexical environment, which is the content of a `&environment` parameter to `defmacro`. For the differences between this two environments see the comments in `src/c/compiler.d` and `src/c/interpreter.d`.

4.6.4 The interpreter stack

The bytecodes interpreter uses a stack of its own to save and restore values from intermediate calculations. This Forth-like data stack is also used in other parts of the C kernel for various purposes, such as saving compiled code, keeping arguments to `format`, etc.

However, one of the most important roles of the Interpreter Stack is to keep a log of the functions which are called during the execution of bytecodes. For each function invoked, the interpreter keeps three lisp objects on the stack:

```
+-----+-----+-----+
| function | lexical environment | index to previous record |
+-----+-----+-----+
```

The first item is the object which is funcalled. It can be a bytecodes object, a compiled function or a generic function. In the last two cases the lexical environment is just `nil`. In the first case, the second item on the stack is the lexical environment on which the code is executed. Each of these records are popped out of the stack after function invocation.

Let us see how these invocation records are used for debugging.

```
> (defun fact (x)                               ;;; Wrong definition of the
  (if (= x 0)                                   ;;; factorial function.
      one                                       ;;; one should be 1.
      (* x (fact (1- x)))))
FACT

> (fact 3)                                       ;;; Tries 3!
Error: The variable ONE is unbound.
Error signalled by IF.
Broken at IF.
>> :b                                          ;;; Backtrace.
Backtrace: eval > fact > if > fact > if > fact > if > fact > IF
;;; Currently at the last IF.
>> :h                                          ;;; Help.

Break commands:
:q(uit)      Return to some previous break level.
:pop         Pop to previous break level.
:c(ontinue)  Continue execution.
:b(acktrace) Print backtrace.
```



```
;;; Again at the top-level loop.
```

4.7 The compiler

4.7.1 The compiler translates to C

The ECL compiler is essentially a translator from Common-Lisp to C. Given a Lisp source file, the compiler first generates three intermediate files:

- a C-file which consists of the C version of the Lisp program
- an H-file which consists of declarations referenced in the C-file
- a Data-file which consists of Lisp data to be used at load time

The ECL compiler then invokes the C compiler to compile the C-file into an object file. Finally, the contents of the Data-file is appended to the object file to make a *Fasl-file*. The generated Fasl-file can be loaded into the ECL system by the Common-Lisp function `load`. By default, the three intermediate files are deleted after the compilation, but, if asked, the compiler leaves them.

The merits of the use of C as the intermediate language are:

- The ECL compiler is highly portable.
- Cross compilation is possible, because the contents of the intermediate files are common to all versions of ECL. For example, one can compile his or her Lisp program by the ECL compiler on a Sun, bring the intermediate files to DOS, compile the C-file with the gcc compiler under DOS, and then append the Data-file to the object file. This procedure generates the Fasl-file for the ECL system on DOS. This kind of cross compilation makes it easier to port ECL.
- Hardware-dependent optimizations such as register allocations are done by the C compiler.

The demerits are:

- At those sites where no C compiler is available, the users cannot compile their Lisp programs.
- The compilation time is long. 70% to 80% of the compilation time is used by the C compiler. The ECL compiler is therefore slower than compiler generating machine code directly.

4.7.2 The compiler mimics human C programmer

The format of the intermediate C code generated by the ECL compiler is the same as the hand-coded C code of the ECL source programs. For example, supposing that the Lisp source file contains the following function definition:

```
(defvar *delta* 2)
(defun add1 (x) (+ *delta* x))
```

The compiler generates the following intermediate C code.

```
/*      function definition for ADD1                                */
/*      optimize speed 3, debug 0, space 0, safety 2                */
static cl_object Lladd1(cl_object v1x)
{
```

```

    cl_object env0 = ECL_NIL;
    const cl_env_ptr cl_env_copy = ecl_process_env();
    cl_object value0;
    ecl_cs_check(cl_env_copy,value0);
    {
TTL:
        value0 = ecl_plus(ecl_symbol_value(VV[0]),v1x);
        cl_env_copy->nvalues = 1;
        return value0;
    }
}

/*      initialization of this module                                */
ECL_DLLEXPORT void init_fas_CODE(cl_object flag)
{
    const cl_env_ptr cl_env_copy = ecl_process_env();
    cl_object value0;
    cl_object *VVtemp;
    if (flag != OBJNULL){
        Cblock = flag;
#ifdef ECL_DYNAMIC_VV
        flag->cblock.data = VV;
#endif
        flag->cblock.data_size = VM;
        flag->cblock.temp_data_size = VMtemp;
        flag->cblock.data_text = compiler_data_text;
        flag->cblock.cfuns_size = compiler_cfuns_size;
        flag->cblock.cfuns = compiler_cfuns;
        flag->cblock.source = make_constant_base_string("test.lisp");
        return;}
#ifdef ECL_DYNAMIC_VV
    VV = Cblock->cblock.data;
#endif
    Cblock->cblock.data_text = (const cl_object *)"@EcLtAg:init_fas_CODE@";
    VVtemp = Cblock->cblock.temp_data;
    ECL_DEFINE_SETF_FUNCTIONS
        si_Xmake_special(VV[0]);
        if (ecl_boundp(cl_env_copy,VV[0])) { goto L2; }
        cl_set(VV[0],ecl_make_fixnum(2));
L2;;
        ecl_cmp_defun(VV[2]);
    }
}
/*      ADD1                                                        */

```

The C function `L1add1` implements the Lisp function `add1`. This relation is established by `ecl_cmp_defun` in the initialization function `init_fas_CODE`, which is invoked at load time. There, the vector `VV` consists of Lisp objects; `VV[0]` and `VV[1]` in this example hold the Lisp symbols `*delta*` and `add1`, while `VV[2]` holds the function object for `add1`, which

is created during initialization of the module. `VM` in the definition of `L1add1` is a C macro declared in the corresponding H-file. The actual value of `VM` is the number of value stack locations used by this module, i.e., 3 in this example. Thus the following macro definition is found in the H-file.

```
#define VM 3
```

4.7.3 Implementation of Compiled Closures

The ECL compiler takes two passes before it invokes the C compiler. The major role of the first pass is to detect function closures and to detect, for each function closure, those lexical objects (i.e., lexical variable, local function definitions, tags, and block-names) to be enclosed within the closure. This check must be done before the C code generation in the second pass, because lexical objects to be enclosed in function closures are treated in a different way from those not enclosed.

Ordinarily, lexical variables in a compiled function `f` are allocated on the C stack. However, if a lexical variable is to be enclosed in function closures, it is allocated on a list, called the "environment list", which is local to `f`. In addition, a local variable is created which points to the lexical variable's location (within the environment list), so that the variable may be accessed through an indirection rather than by list traversal.

The environment list is a pushdown list: It is empty when `f` is called. An element is pushed on the environment list when a variable to be enclosed in closures is bound, and is popped when the binding is no more in effect. That is, at any moment during execution of `f`, the environment list contains those lexical variables whose binding is still in effect and which should be enclosed in closures. When a compiled closure is created during execution of `f`, the compiled code for the closure is coupled with the environment list at that moment to form the compiled closure.

Later, when the compiled closure is invoked, a pointer is set up to each lexical variable in the environment list, so that each object may be referenced through a memory indirection.

Let us see an example. Suppose the following function has been compiled.

```
(defun foo (x)
  (let ((a #'(lambda () (incf x)))
        (y x))
    (values a #'(lambda () (incf x y))))))
```

`foo` returns two compiled closures. The first closure increments `x` by one, whereas the second closure increments `x` by the initial value of `x`. Both closures return the incremented value of `x`.

```
>(multiple-value-setq (f g) (foo 10))
#<compiled-closure nil>
```

```
>(funcall f)
11
```

```
>(funcall g)
21
```

```
>
```

After this, the two compiled closures look like:

```

second closure      y:      x:
|-----|-----|      |-----|-----|      |-----|-----|
|  **   |      --|----->|  10   |      --|----->|  21   |  nil   |
|-----|-----|      |-----|-----|      |-----|-----|
^

first closure
|-----|-----|
|  *    |      --|-----|
|-----|-----|

```

```

* : address of the compiled code for #'(lambda () (incf x))
** : address of the compiled code for #'(lambda () (incf x y))

```

4.7.4 Use of Declarations to Improve Efficiency

Declarations, especially type and function declarations, increase the efficiency of the compiled code. For example, for the following Lisp source file, with two Common-Lisp declarations added,

```

(eval-when (:compile-toplevel)
  (proclaim '(ftype (function (fixnum fixnum) fixnum) tak))
  (proclaim '(optimize (speed 3) (debug 0) (safety 0))))

(defun tak (x y)
  (declare (fixnum x y))
  (if (not (< y x))
      y
      (tak (tak (1- x) y)
            (tak (1- y) x))))

```

The compiler generates the following C code (Note that the tail-recursive call of `tak` was replaced by iteration):

```

/*      function definition for TAK                                     */
/*      optimize speed 3, debug 0, space 0, safety 0                   */
static cl_object L1tak(cl_object v1x, cl_object v2y)                  */
{
  cl_object env0 = ECL_NIL;
  const cl_env_ptr cl_env_copy = ecl_process_env();
  cl_object value0;
  cl_fixnum v3x;
  cl_fixnum v4y;
  v3x = ecl_fixnum(v1x);
  v4y = ecl_fixnum(v2y);
TTL:
  if ((v4y)<(v3x)) { goto L1; }
  value0 = ecl_make_fixnum(v4y);
  cl_env_copy->nvalues = 1;
  return value0;
}

```

```

L1;;
{
  cl_fixnum v5;
  {
    cl_fixnum v6;
    v6 = (v3x)-1;
    v5 = ecl_fixnum(L1tak(ecl_make_fixnum(v6), ecl_make_fixnum(v4y)));
  }
  {
    cl_fixnum v6;
    v6 = (v4y)-1;
    v4y = ecl_fixnum(L1tak(ecl_make_fixnum(v6), ecl_make_fixnum(v3x)));
  }
  v3x = v5;
}
goto TTL;
}

```

4.7.5 Inspecting generated C code

Common-Lisp defines a function `disassemble`, which is supposed to disassemble a compiled function and to display the assembler code. According to *Common-Lisp: The Language*,

This is primary useful for debugging the compiler, ..\\

This is, however, *useless* in our case, because we are not concerned with assembly language. Rather, we are interested in the C code generated by the ECL compiler. Thus the `disassemble` function in ECL accepts not-yet-compiled functions only and displays the translated C code.

```
> (defun add1 (x) (1+ x))
```

```
ADD1
```

```
> (disassemble *)
```

```

/*      function definition for ADD1                                */
/*      optimize speed 3, debug 0, space 0, safety 2                */
static cl_object L1add1(cl_object v1x)
{
  cl_object env0 = ECL_NIL;
  const cl_env_ptr cl_env_copy = ecl_process_env();
  cl_object value0;
  ecl_cs_check(cl_env_copy,value0);
  {
TTL:
    value0 = ecl_one_plus(v1x);
    cl_env_copy->nvalues = 1;
    return value0;
  }
}

```

4.8 Porting ECL

To port ECL to a new architecture, the following steps are required:

1. Ensure that the GNU Multiprecision library supports this machine.
2. Ensure that the Boehm-Weiser garbage collector is supported by that architecture. Alternatively, port ECL's own garbage collector `src/c/alloc.d` and `src/c/gbc.d` to that platform.
3. Fix `src/aclocal.in`, `src/h/config.h.in` and `src/h/ecl.h` so that they supply flags for the new host machine.
4. Fix the machine dependent code in `src/c/`. The most critical parts are in the `unix*.d` and `thread*.d` files.
5. Compile as in any other platform.
6. Run the tests and compare to the results of other platforms.

4.9 Removed features

In-house DFFI

Commit 10bd3b613fd389da7640902c2b88a6e36088c920. Native DFFI was replaced by a libffi (<https://sourceware.org/libffi/>) long time ago, but we have maintained the code as a fallback. Due to small number of supported platforms and no real use it has been removed in 2016.

In-house GC

Commit 61500316b7ea17d0e42f5ca127f2f9fa3e6596a8. Broken GC is replaced by BoehmGC library. This may be added back as a fallback in the near future.

3bd9799a2fef21cc309472e604a46be236b155c7 removes a leftover (apparently `gbc.d` wasn't bdwgc glue).

Green threads

Commit 41923d5927f31f4dd702f546b9caee74e98a2080. Green threads (aka light weight processes) has been replaced with native threads implementation. There is an ongoing effort to bring them back as an alternative interface.

Compiler newcmp

Commit 9b8258388487df8243e2ced9c784e569c0b34c4f This was abandoned effort of changing the compiler architecture. Some clever ideas and a compiler package hierarchy. Some of these things should be incorporated during the evolution of the primary compiler.

Old MIT loop

Commit 5042589043a7be853b7f85fd7a996747412de6b4. This old loop implementation has got superseded by the one incorporated from Symbolics LOOP in 2001.

Support for bignum arithmetic (earith.d)

Commit edfc2ba785d6a64667e89c869ef0a872d7b9704b. Removes pre-gmp bignum code. Name comes probably from “extended arithmetic”, contains multiplication and division routines (assembler and a portable implementation).

Unification module

Commit [6ff5d20417a21a76846c4b28e532aac097f03109](#). Old unification module (logic programming) from EcoLisp times.

Hierarchical packages

Commit [72e422f1b3c4b3c52fa273b961517db943749a8f](#). Partially broken. Tests left in `package-extensions.lsp`.

8-bit opcodes in bytecodes interpreter

Commit [c3244b0148ed352808779b07b25c3edddf9d7349](#). Works fine but provides no real gain and is limited to intel.

thread local variables

Commit [618f6e92e8144f7b95bc36b42a337c212bacc8e7](#). Disabled by default, practically not tested, works on limited number of platforms.

Indexes

Concept index

- (
(complex float) internal representation 40
- A**
ANSI Dictionary 23
Arrays 54
- B**
Bytecodes eager compilation 26
- C**
C/C++ code inlining 103
Command line processing 94
Common Lisp functions limits 27
Compiler declarations 23
Creating executables and libraries 85
cstring and foreign string differences 116
- D**
Defun preprocessor 153
disassemble and compile on
 defined functions 27
Dynamic foreign function interface 107
- E**
Eager compilation implications 26
Environment implementation 165
External processes 95
- F**
FIFO files (named pipes) 97
Foreign aggregate types 109
Foreign function interface 98
Foreign functions and libraries 120
Foreign objects 112
Foreign primitive types 107
Foreign strings 116
- H**
Hash table generic test 65
Hash table serialization 65
- L**
long-float internal representation 40
- M**
Memory management 138
- N**
Native FASL 87
Native threads 122
- O**
Object file internal layout 88
One type for everything: `cl_object` 21
Only in Common Lisp 23
- P**
Package local nicknames 143
Package locks 145
Parsing arguments in standalone executable 95
Portable FASL 86
- R**
Readers-writer locks 127
- S**
Shadowed bindings in let, flet, labels
 and lambda-list 26
Shared-exclusive locks 127
Static foreign function interface 103
Synchronized hash tables 65
System building 85
- T**
Thread-safe hash tables 65
Two kinds of FFI 99
- U**
Universal foreign function interface 107
- W**
Weak hash tables 64

Configure option index

--enable-c99complex [auto]	42	--with-dffi [system included AUTO no]	99
--enable-shared [YES no]	87	--with-fpe [YES no]	40
--enable-small-cons [YES no]	155	--with-ieee-fp [YES no]	40
--enable-threads [yes no AUTO]	122	--with-libffi-prefix=path	99
--enable-unicode [32 16 no]	47, 158		

Feature index

C

COMPLEX-FLOAT	40, 107
---------------------	---------

D

DFFI	100
DLOPEN	87

E

ecl-read-write-lock	127
ECL-WEAK-HASH	64

F

FFI	98
-----------	----

Example index

A

Accessing underlying <code>cl_object</code> structure	156
Atomic update of a structure slot	131

B

Building executable	89
Building native FASL	87
Building Portable FASL file	86
Building shared library	89
Building static library	88

C

CFFI usage	102
<code>cl_object</code> checking the type with <code>ecl_t_of</code> ...	155
Conversion between foreign	
string and cstring	117
cstring used to send and return a value	116

L

LONG-FLOAT	40, 107
LONG-LONG	107

P

PACKAGE-LOCAL-NICKNAMES	143
PACKAGE-LOCKS	145

T

THREADS	122
---------------	-----

U

UINT16-T	107
UINT32-T	107
UINT64-T	107

D

Define a compare-and-swap expansion	132
Defpackage :lock option	146
defpackage and package local nicknames	144
distinguishing between base and	
Unicode character	159
dpp: function definition	153

E

Eager compilation impact on macros	26
<code>ecl_aret</code> and <code>ecl_aset</code> accessing arrays	161
<code>ecl_aret1</code> and <code>ecl_aset1</code> accessing vectors ...	162
<code>ecl_array_eltype</code> different	
types of objects	161
<code>ecl_read_from_cstring</code> constructing	
Lisp objects in C	157
<code>ext:with-backend</code> use different code for c and	
bytecodes compiler	107

F

<code>ffi:allocate-foreign-object</code>	
allocating structure object	112
<code>ffi:c-inline</code> inlining c code	105
<code>ffi:c-inline</code> returning multiple values	105
<code>ffi:c-progn</code> interleaving c and lisp code	105
<code>ffi:clines</code> adding c toplevel declarations	104
<code>ffi:def-array-pointer</code> usage	111
<code>ffi:def-constant</code> defining constants	108
<code>ffi:def-enum</code> sample enumerations	110
<code>ffi:def-foreign-type</code> examples	109
<code>ffi:def-foreign-var</code> places in	
foreign world	115
<code>ffi:def-function</code>	120
<code>ffi:def-struct</code> defining C structure	110
<code>ffi:def-union</code> union definition and usage	112
<code>ffi:deref-array</code> retrieving array element	111
<code>ffi:deref-pointer</code>	113
<code>ffi:ensure-char-character</code>	114
<code>ffi:ensure-char-integer</code>	114
<code>ffi:find-foreign-library</code>	121
<code>ffi:get-slot-value</code>	
manipulating a struct field	110
<code>ffi:get-slot-value</code> usage	111
<code>ffi:load-foreign-library</code>	121
<code>ffi:null-char-p</code> example	109
<code>ffi:size-of-foreign-type</code>	113
<code>ffi:with-cast-pointer</code>	115
<code>ffi:with-foreign-object</code> macro usage	113
foreign string used to send and	
return a value	117

H

Hash table extensions example	65
-------------------------------------	----

I

Initializing static/shared library in C/C++	88
---	----

Function index**K**

Keeping lambda definitions with	
<code>si:*keep-definitions*</code>	27
Killing process	124

L

LS implementation	95
-------------------------	----

M

<code>mp:process-run-function</code> usage	125
--	-----

P

Possible implementation of	
<code>mp:process-run-function:</code>	123
Process interruption	123

S

Safely executing Lisp code with floating point	
exceptions in embedding program	18
Setting a signal handler	138
SFFI usage	103
<code>si::make-lambda</code> usage	
(bytecodes compilation)	26
<code>si_make_lambda</code> building functions	165
<code>si_safe_eval</code>	165
Suspend and resume process	124

T

trace usage	82
-------------------	----

U

UFFI usage	101
Using sequence streams	73

W

<code>with-cstring</code>	118
---------------------------------	-----

-
 _ecl_caar 51
 _ecl_cadr 51
 _ecl_car 51
 _ecl_cdr 51

C

cl:deffpackage 144, 146
 cl_boot 13
 cl_shutdown 14

D

directory 69
 disassemble 81

E

ecl_aet_to_symbol 56
 ecl_alloc_adjustable_base_string 60
 ecl_alloc_simple_base_string 60
 ecl_alloc_simple_vector 56
 ecl_alpha_char_p 49
 ecl_alphanumericp 49
 ecl_aref 57, 161
 ecl_aref1 57, 161
 ecl_array_dimension 58
 ecl_array_elttype 58, 161
 ecl_array_rank 58
 ecl_aset 57, 161
 ecl_aset1 57, 161
 ecl_atomic_get 131
 ecl_atomic_incf 130
 ecl_atomic_incf_by_fixnum 130
 ecl_atomic_index_incf 130
 ecl_atomic_pop 131
 ecl_atomic_push 131
 ecl_base_char_code 49, 159
 ecl_base_char_p 49
 ecl_bds_bind 28
 ecl_bds_push 28
 ecl_bds_unwind_n 29
 ecl_bds_unwind1 29
 ecl_both_case_p 50
 ecl_cdfloat 44
 ecl_char 61
 ecl_char_cmp 159
 ecl_char_code 49, 159
 ecl_char_compare 159
 ecl_char_downcase 50
 ecl_char_eq 159
 ecl_char_equal 159
 ecl_char_set 61
 ecl_char_upcase 50
 ecl_clear_interrupts 18
 ecl_clfloat 44
 ecl_compare_and_swap 130

ecl_csfloat 44
 ecl_digitp 49
 ecl_disable_interrupts 18
 ecl_double_float 44
 ecl_enable_interrupts 18
 ecl_fixnum 44, 158
 ecl_fixnum_geq 158
 ecl_fixnum_greater 158
 ecl_fixnum_leq 158
 ecl_fixnum_lower 158
 ecl_fixnum_minusp 158
 ecl_fixnum_plus 158
 ecl_get_option 16
 ecl_graphic_char_p 49
 ecl_import_current_thread 16
 ecl_long_float 44
 ecl_lower_case_p 50
 ecl_make_cdfloat 43
 ecl_make_clfloat 43
 ecl_make_complex 43
 ecl_make_constant_base_string 61
 ecl_make_csfloat 43
 ecl_make_double_float 43
 ecl_make_fixnum 43, 158
 ecl_make_int 43
 ecl_make_int16_t 43
 ecl_make_int32_t 43
 ecl_make_int64_t 43
 ecl_make_int8_t 43
 ecl_make_integer 43
 ecl_make_keyword 36
 ecl_make_lock 126
 ecl_make_long 43
 ecl_make_long_float 43
 ecl_make_long_long 43
 ecl_make_ratio 43
 ecl_make_rwlock 128
 ecl_make_semaphore 129
 ecl_make_short_t 43
 ecl_make_simple_base_string 61
 ecl_make_single_float 43
 ecl_make_symbol 37
 ecl_make_uint 43
 ecl_make_uint16_t 43
 ecl_make_uint32_t 43
 ecl_make_uint64_t 43
 ecl_make_uint8_t 43
 ecl_make_ulong 43
 ecl_make_ulong_long 43
 ecl_make_unsigned_integer 43
 ecl_make_ushort_t 43
 ecl_nth_value 30
 ecl_nvalues 30
 ecl_process_env 25
 ecl_read_from_cstring 157
 ecl_release_current_thread 16
 ecl_return0 31
 ecl_return1 31

ecl_return2.....	31	ECL_CLASS_SUPERIORS.....	164
ecl_return3.....	31	ECL_CODE_CHAR.....	49, 159
ecl_set_option.....	14	ECL_COMPLEXP.....	156
ecl_setq.....	29	ECL_CONS_CAR.....	51
ecl_single_float.....	44	ECL_CONS_CDR.....	51
ecl_standard_char_p.....	49	ECL_CONSP.....	156
ecl_symbol_to_aet.....	56	ECL_DOUBLE_FLOAT_P.....	156
ecl_symbol_value.....	29	ECL_EXTENDED_STRING_P.....	162
ecl_t_of.....	156	ECL_FIXNUMP.....	156
ecl_to_cdfloat.....	44	ECL_FOREIGN_DATA_P.....	156
ecl_to_clfloat.....	44	ECL_HANDLER_CASE.....	34
ecl_to_csfloat.....	44	ECL_HASH_TABLE_P.....	156
ecl_to_double.....	44	ECL_IMMEDIATE.....	157
ecl_to_fixnum.....	44	ECL_INSTANCEP.....	164
ecl_to_float.....	44	ECL_LISTP.....	156
ecl_to_int.....	45	ECL_LONG_FLOAT_P.....	156
ecl_to_int16_t.....	44	ECL_NUMBER_TYPE_P.....	156
ecl_to_int32_t.....	44	ECL_PACKAGEP.....	156
ecl_to_int64_t.....	44	ECL_PATHNAMEP.....	156
ecl_to_int8_t.....	44	ECL_RANDOM_STATE_P.....	156
ecl_to_long.....	45	ECL_READTABLEP.....	156
ecl_to_long_double.....	44	ECL_REAL_TYPE_P.....	156
ecl_to_long_long.....	45	ECL_RESTART_CASE.....	35
ecl_to_short.....	44	ECL_RPLACA.....	51
ecl_to_uint.....	45	ECL_RPLACD.....	51
ecl_to_uint16_t.....	44	ECL_SINGLE_FLOAT_P.....	156
ecl_to_uint32_t.....	44	ECL_SPEC_FLAG.....	164
ecl_to_uint64_t.....	44	ECL_SPEC_OBJECT.....	164
ecl_to_uint8_t.....	44	ECL_SSE_PACK_P.....	156
ecl_to_ulong.....	45	ECL_STRINGP.....	156
ecl_to_ulong_long.....	45	ECL_STRUCT_LENGTH.....	164
ecl_to_unsigned_integer.....	44	ECL_STRUCT_NAME.....	164
ecl_to_ushort.....	44	ECL_STRUCT_SLOT.....	164
ecl_upper_case_p.....	50	ECL_STRUCT_SLOTS.....	164
ecl_va_arg.....	29	ECL_STRUCT_TYPE.....	164
ecl_va_end.....	29	ECL_SYMBOLP.....	156
ecl_va_start.....	29	ECL_UNWIND_PROTECT.....	17
ECL_ADJUSTABLE_ARRAY_P.....	159	ECL_UNWIND_PROTECT_BEGIN.....	31
ECL_ANSI_STREAM_P.....	164	ECL_VECTORP.....	156
ECL_ANSI_STREAM_TYPE_P.....	164	ECL_WITH_LISP_FPE.....	18
ECL_ARRAY_HAS_FILL_POINTER_P.....	159	ext:add-package-local-nickname.....	144
ECL_ARRAYP.....	156	ext:all-encodings.....	74
ECL_ATOM.....	156	ext:catch-signal.....	138
ECL_BASE_CHAR_CODE_P.....	156	ext:character-coding-error- external-format.....	74
ECL_BASE_CHAR_P.....	156	ext:character-decoding-error-octects.....	74
ECL_BASE_STRING_P.....	162	ext:character-encoding-error-code.....	74
ECL_BIGNUMP.....	156	ext:chdir.....	98
ECL_BIT_VECTOR_P.....	156	ext:chmod.....	98
ECL_BLOCK_BEGIN.....	31	ext:command-args.....	94
ECL_CATCH_ALL.....	16	ext:copy-file.....	98
ECL_CATCH_BEGIN.....	31	ext:decoding-error.....	75
ECL_CHAR_CODE.....	49, 159	ext:encoding-error.....	75
ECL_CHARACTERP.....	156	ext:environ.....	98
ECL_CLASS_CPL.....	164	ext:external-process-error-stream.....	96
ECL_CLASS_INFERIORS.....	164	ext:external-process-input.....	96
ECL_CLASS_NAME.....	164	ext:external-process-output.....	96
ECL_CLASS_OF.....	164	ext:external-process-pid.....	96
ECL_CLASS_SLOTS.....	164		

ext:external-process-status	96
ext:external-process-wait	96
ext:file-kind	98
ext:file-stream-fd	74
ext:float-infinity-p	41
ext:float-nan-p	41
ext:get-finalizer	141
ext:get-limit	141
ext:get-signal-handler	137
ext:getcwd	98
ext:getenv	98
ext:getpid	98
ext:getuid	98
ext:hash-table-content	65
ext:hash-table-fill	65
ext:hash-table-synchronized-p	65
ext:hash-table-weakness	65
ext:lock-package	146
ext:make-pipe	98
ext:make-sequence-input-stream	73
ext:make-sequence-output-stream	73
ext:nan	41
ext:package-local-nicknames	144
ext:package-locally-nicknamed-by-list	144
ext:package-locked-p	146
ext:process-command-args	94
ext:quit	98
ext:remove-package-local-nickname	145
ext:run-program	96
ext:set-buffering-mode	73
ext:set-finalizer	142
ext:set-limit	142
ext:set-signal-handler	137
ext:setenv	98
ext:stack-overflow-size	141
ext:stack-overflow-type	141
ext:system	98
ext:terminate-process	96
ext:trap-fpe	40
ext:unix-signal-received-code	137
ext:unlock-package	146
ext:with-backend	107
ext:with-unlocked-packages	146
ext:without-package-locks	146

F

ffi:allocate-foreign-object	112
ffi:allocate-foreign-string	119
ffi:c-inline	104
ffi:c-progn	105
ffi:clines	103
ffi:convert-from-cstring	117, 118
ffi:convert-from-foreign-string	119
ffi:convert-to-cstring	117
ffi:convert-to-foreign-string	119
ffi:def-array-pointer	111
ffi:def-constant	108

ffi:def-enum	109
ffi:def-foreign-type	109
ffi:def-foreign-var	115
ffi:def-function	120
ffi:def-struct	110
ffi:def-union	111
ffi:defcallback	106
ffi:defcbody	106
ffi:defentry	106
ffi:defla	107
ffi:deref-array	111
ffi:deref-pointer	113
ffi:ensure-char-character	114
ffi:ensure-char-integer	114
ffi:find-foreign-library	121
ffi:free-foreign-object	112
ffi:get-slot-pointer	111
ffi:get-slot-value	110
ffi:load-foreign-library	120
ffi:make-null-pointer	114
ffi:null-char-p	109
ffi:null-pointer-p	115
ffi:pointer-address	113
ffi:size-of-foreign-type	113
ffi:with-cast-pointer	115
ffi:with-cstring	118
ffi:with-cstrings	118
ffi:with-foreign-object	112
ffi:with-foreign-string	119
ffi:with-foreign-strings	120
fixint	158
fixnint	158

M

mp:all-processes	122
mp:atomic-decf	131
mp:atomic-incf	131
mp:atomic-pop	132
mp:atomic-push	132
mp:atomic-update	131
mp:block-signals	125
mp:compare-and-swap	131
mp:condition-variable-broadcast	129
mp:condition-variable-signal	129
mp:condition-variable-timedwait	129
mp:condition-variable-wait	129
mp:defcas	133
mp:define-cas-expander	132
mp:exit-process	122
mp:get-cas-expansion	133
mp:get-lock	127
mp:get-rwlock-read	128
mp:get-rwlock-write	128
mp:giveup-lock	127
mp:giveup-rwlock-read	128
mp:giveup-rwlock-write	128
mp:holding-lock-p	127

mp:interrupt-process	122
mp:lock-count	127
mp:lock-owner	127
mp:lock_name	127
mp:make-condition-variable	129
mp:make-lock	126
mp:make-process	123
mp:make-rwlock	128
mp:make-semaphore	129
mp:process-active-p	123
mp:process-enable	123
mp:process-join	124
mp:process-kill	124
mp:process-name	125
mp:process-preset	125
mp:process-resume	124
mp:process-run-function	125
mp:process-suspend	124
mp:process-yield	124
mp:recursive-lock-p	127
mp:remcas	133
mp:restore-signals	125
mp:rwlock-name	128
mp:semaphore-count	130
mp:semaphore-name	129
mp:semaphore-wait-count	130
mp:signal-semaphore	130
mp:try-get-semaphore	130
mp:wait-on-semaphore	130
mp:with-interrupts	126, 137
mp:with-lock	127
mp:with-rwlock	128
mp:without-interrupts	125, 137
mp_all_processes	122
mp_block_signals	125
mp_condition_variable_broadcast	129
mp_condition_variable_signal	129
mp_condition_variable_timedwait	129
mp_condition_variable_wait	129
mp_current_process	125
mp_exit_process	122
mp_get_lock_nowait	127
mp_get_lock_wait	127
mp_get_rwlock_read_nowait	128
mp_get_rwlock_read_wait	128
mp_get_rwlock_write_nowait	128
mp_get_rwlock_write_wait	128
mp_giveup_lock	127
mp_giveup_rwlock_read	128
mp_giveup_rwlock_write	128

Variable index

mp_holding_lock_p	127
mp_interrupt_process	122
mp_lock_count	127
mp_lock_name	127
mp_lock_owner	127
mp_make_condition_variable	129
mp_make_process	123
mp_process_active_p	123
mp_process_enable	123
mp_process_join	124
mp_process_kill	124
mp_process_name	125
mp_process_preset	125
mp_process_resume	124
mp_process_run_function	125
mp_process_suspend	124
mp_process_yield	124
mp_recursive_lock_p	127
mp_restore_signals	125
mp_rwlock_name	128
mp_semaphore_count	130
mp_semaphore_name	129
mp_semaphore_wait_count	130
mp_signal_semaphore	130
mp_try_get_semaphore	130
mp_wait_on_semaphore	130

R

rename-file	69
-------------------	----

S

si::string-to-object	157
si:safe-eval	164
si_add_package_local_nickname	144
si_make_array	56
si_make_lambda	165
si_make_vector	56
si_package_local_nicknames	144
si_package_locally_nicknamed_by_list	144
si_remove_package_local_nickname	145
si_safe_eval	164
si_string_to_object	157

T

trace	81
typedef struct { ... } ecl_va_list[1];	29

C

c:*cc-optimize*	93
c:*user-cc-flags*	93
c:*user-ld-flags*	93
c::*ar*	93
c::*cc*	93
c::*ecl-include-directory*	94
c::*ecl-library-directory*	94
c::*ld*	93
c::*ranlib*	93

E

ECL_ARRAY_DIMENSION_LIMIT	55
ECL_ARRAY_RANK_LIMIT	55
ECL_ARRAY_TOTAL_LIMIT	55
ECL_CHAR_CODE_LIMIT	159
ECLDIR	16
ext:*help-message*	94

Type index**C**

cl_elttype	55, 160
cl_fixnum	158
cl_index	158
cl_lispunion	154
cl_object	155
cl_type	156

E

ecl_array	160
ecl_base_string	162
ecl_character	158
ecl_file_ops	163

Common Lisp symbols**:**

:*	107
:byte	107
:cdfloat	107
:char	107
:clfloat	107
:csfloat	107
:cstring	107
:double	107
:float	107
:int	107
:int16_t	107
:int32_t	107

ext:*lisp-init-file-list*	94
ext:*default-command-arg-rules*	94
ext:{short,single,double,long}-float- {positive,negative}-infinity	41

F

ffi:*use-dffi*	107
ffi:*null-cstring-pointer*	115

M

MOST_NEGATIVE_FIXNUM	158
MOST_POSITIVE_FIXNUM	158
mp:*current-process*	125

S

si:*keep-definitions*	27
-----------------------	----

ecl_stream	163
ecl_string	162
ecl_vector	159
ext:character-coding-error	74
ext:character-decoding-error	74
ext:character-encoding-error	74
ext:sequence-stream	73
ext:stack-overflow	141
ext:storage-exhausted	141
ext:stream-decoding-error	75
ext:stream-encoding-error	75
ext:unix-signal-received	137

:int64_t	107
:long	107
:long-double	107
:object	107
:pointer-void	107
:short	107
:uint16_t	107
:uint32_t	107
:uint64_t	107
:unsigned-byte	107
:unsigned-char	107
:unsigned-int	107
:unsigned-long	107

:unsigned-short 107
:void 107

C

call-arguments-limit 27

D

debug 23
directory 69
disassemble 81
double-float 25

E

ext:*help-message* 94
ext:*lisp-init-file-list* 94
ext:+default-command-arg-rules+ 94
ext:add-package-local-nickname 144
ext:all-encodings 74
ext:binding-stack 139
ext:c-stack 139
ext:catch-signal 138
ext:character-coding-error 74
ext:character-coding-error-
 external-format 74
ext:character-decoding-error 74
ext:character-decoding-error-octects 74
ext:character-encoding-error 74
ext:character-encoding-error-code 74
ext:chdir 98
ext:chmod 98
ext:command-args 94
ext:copy-file 98
ext:decoding-error 75
ext:double-float-negative-infinity 41
ext:double-float-positive-infinity 41
ext:encoding-error 75
ext:environ 98
ext:external-process-error-stream 96
ext:external-process-input 96
ext:external-process-output 96
ext:external-process-pid 96
ext:external-process-status 96
ext:external-process-wait 96
ext:file-kind 98
ext:file-stream-fd 74
ext:float-infinity-p 41
ext:float-nan-p 41
ext:frame-stack 139
ext:get-finalizer 141
ext:get-limit 141
ext:get-signal-handler 137
ext:getcwd 98
ext:getenv 98
ext:getpid 98
ext:getuid 98

ext:hash-table-fill 65
ext:hash-table-synchronized-p 65
ext:hash-table-weakness 65
ext:heap-size 139
ext:lisp-stack 139
ext:lock-package 146
ext:long-float-negative-infinity 41
ext:long-float-positive-infinity 41
ext:make-pipe 98
ext:make-sequence-input-stream 73
ext:make-sequence-output-stream 73
ext:nan 41
ext:package-local-nicknames 144
ext:package-locally-nicknamed-by-list 144
ext:package-locked-p 146
ext:process-command-args 94
ext:quit 98
ext:remove-package-local-nickname 145
ext:run-program 96
ext:sequence-stream 73
ext:set-buffering-mode 73
ext:set-finalizer 142
ext:set-limit 142
ext:set-signal-handler 137
ext:setenv 98
ext:short-float-negative-infinity 41
ext:short-float-positive-infinity 41
ext:single-float-negative-infinity 41
ext:single-float-positive-infinity 41
ext:stack-overflow 141
ext:stack-overflow-size 141
ext:stack-overflow-type 141
ext:storage-exhausted 141
ext:stream-decoding-error 75
ext:stream-encoding-error 75
ext:system 98
ext:terminate-process 96
ext:trap-fpe 40
ext:unix-signal-received 137
ext:unix-signal-received-code 137
ext:unlock-package 146
ext:with-backend 107
ext:with-unlocked-packages 146
ext:without-package-locks 146
extended-char 48

F

ffi:*use-dffi*	107
ffi:+null-cstring-pointer+	115
ffi:allocate-foreign-object	112
ffi:allocate-foreign-string	119
ffi:c-inline	104
ffi:c-progn	105
ffi:clines	103
ffi:convert-from-cstring	117
ffi:convert-from-foreign-string	119
ffi:convert-to-cstring	117
ffi:convert-to-foreign-string	119
ffi:def-array-pointer	111
ffi:def-constant	108
ffi:def-enum	109
ffi:def-foreign-type	109
ffi:def-foreign-var	115
ffi:def-function	120
ffi:def-struct	110
ffi:def-union	111
ffi:decallback	106
ffi:decbbody	106
ffi:defentry	106
ffi:defla	107
ffi:deref-array	111
ffi:deref-pointer	113
ffi:ensure-char-character	114
ffi:ensure-char-integer	114
ffi:find-foreign-library	121
ffi:free-cstring	118
ffi:free-foreign-object	112
ffi:get-slot-pointer	111
ffi:get-slot-value	110
ffi:load-foreign-library	120
ffi:make-null-pointer	114
ffi:null-char-p	109
ffi:null-pointer-p	115
ffi:pointer-address	113
ffi:size-of-foreign-type	113
ffi:with-cast-pointer	115
ffi:with-cstring	118
ffi:with-cstrings	118
ffi:with-foreign-object	112
ffi:with-foreign-string	119
ffi:with-foreign-strings	120

H

hash-table-content	65
--------------------	----

L

lambda-list-keywords	27
lambda-parameters-limit	27

M

MOST-NEGATIVE-FIXNUM	158
MOST-POSITIVE-FIXNUM	158
mp:*current-process*	125
mp:all-processes	122
mp:allow-with-interrupts	125
mp:atomic-decf	131
mp:atomic-incf	131
mp:atomic-pop	132
mp:atomic-push	132
mp:atomic-update	131
mp:block-signals	125
mp:compare-and-swap	131
mp:condition-variable-broadcast	129
mp:condition-variable-signal	129
mp:condition-variable-timedwait	129
mp:condition-variable-wait	129
mp:defcas	133
mp:define-cas-expander	132
mp:exit-process	122
mp:get-cas-expansion	133
mp:get-lock	127
mp:get-rwlock-read	128
mp:get-rwlock-write	128
mp:giveup-lock	127
mp:giveup-rwlock-read	128
mp:giveup-rwlock-write	128
mp:holding-lock-p	127
mp:interrupt-process	122
mp:lock-count	127
mp:lock-name	127
mp:lock-owner	127
mp:make-condition-variable	129
mp:make-lock	126
mp:make-process	123
mp:make-rwlock	128
mp:make-semaphore	129
mp:process-active-p	123
mp:process-enable	123
mp:process-join	124
mp:process-kill	124
mp:process-name	125
mp:process-preset	125
mp:process-resume	124
mp:process-run-function	125
mp:process-suspend	124
mp:process-yield	124
mp:recursive-lock-p	127
mp:remcas	133
mp:restore-signals	125
mp:rwlock-name	128
mp:semaphore-count	130
mp:semaphore-name	129
mp:semaphore-wait-count	130
mp:signal-semaphore	130
mp:try-get-semaphore	130
mp:wait-on-semaphore	130
mp:with-interrupts	126

mp:with-local-interrupts.....	125
mp:with-lock.....	127
mp:with-restored-interrupts.....	125
mp:with-rwlock.....	128
mp:without-interrupts.....	125
mp_lock_owner.....	127
multiple-values-limit.....	27

O

optimize.....	23
---------------	----

R

read-char.....	70
----------------	----

C/C++ index

-	
_ecl_caar.....	51
_ecl_cadr.....	51
_ecl_car.....	51
_ecl_cdr.....	51

C

c_string_to_object.....	157
CHAR_CODE.....	159
CHAR_CODE_LIMIT.....	159
cl_boot.....	13
cl_elttype.....	55, 160
cl_env_ptr.....	165
cl_env_struct.....	165
cl_eval.....	164
cl_fixnum.....	157, 158
cl_index.....	158
cl_lispunion.....	154
cl_object.....	155
cl_safe_eval.....	164
cl_shutdown.....	14
CODE_CHAR.....	159

E

ecl_aet_to_symbol.....	56
ecl_alloc_adjustable_base_string.....	60
ecl_alloc_simple_base_string.....	60
ecl_alloc_simple_vector.....	56
ecl_alpha_char_p.....	49
ecl_alphanumeric_p.....	49
ecl_aref.....	57, 161
ecl_aref1.....	57, 161
ecl_array.....	160
ecl_array_dimension.....	58

S

safety.....	23
si:*keep-definitions*.....	27
si::make-lambda.....	26
single-float.....	25
space.....	23
speed.....	23
standard-char.....	48

T

trace.....	81
------------	----

W

write-char.....	70
write-sequence.....	70

ecl_array_elttype.....	58, 161
ecl_array_rank.....	58
ecl_aset.....	57, 161
ecl_aset1.....	57, 161
ecl_atomic_get.....	131
ecl_atomic_incf.....	130
ecl_atomic_incf_by_fixnum.....	130
ecl_atomic_index_incf.....	130
ecl_atomic_pop.....	131
ecl_atomic_push.....	131
ecl_base_char.....	49, 158
ecl_base_char_code.....	49, 159
ecl_base_char_p.....	49
ecl_base_string.....	162
ecl_bds_bind.....	28
ecl_bds_push.....	28
ecl_bds_unwind_n.....	29
ecl_bds_unwind1.....	29
ecl_both_case_p.....	50
ecl_cdfloat.....	44
ecl_char.....	61
ecl_char_cmp.....	159
ecl_char_code.....	49, 159
ecl_char_compare.....	159
ecl_char_downcase.....	50
ecl_char_eq.....	159
ecl_char_equal.....	159
ecl_char_set.....	61
ecl_char_upcase.....	50
ecl_character.....	49, 158
ecl_clear_interrupts.....	18
ecl_clfloat.....	44
ecl_compare_and_swap.....	130
ecl_csfloat.....	44
ecl_digit_p.....	49
ecl_disable_interrupts.....	18

<code>ecl_double_float</code>	44	<code>ecl_return3</code>	31
<code>ecl_enable_interrupts</code>	18	<code>ecl_set_option</code>	14
<code>ecl_file_pos</code>	162	<code>ecl_setq</code>	29
<code>ecl_fixnum</code>	44, 158	<code>ecl_single_float</code>	44
<code>ecl_fixnum_geq</code>	158	<code>ecl_standard_char_p</code>	49
<code>ecl_fixnum_greater</code>	158	<code>ecl_stream</code>	163
<code>ecl_fixnum_leq</code>	158	<code>ecl_string</code>	162
<code>ecl_fixnum_lower</code>	158	<code>ecl_symbol_to_aet</code>	56
<code>ecl_fixnum_minusp</code>	158	<code>ecl_symbol_value</code>	29
<code>ecl_fixnum_plus</code>	158	<code>ecl_t_of</code>	156
<code>ecl_get_option</code>	16	<code>ecl_to_cdfloat</code>	44
<code>ecl_graphic_char_p</code>	49	<code>ecl_to_clfloat</code>	44
<code>ecl_import_current_thread</code>	16	<code>ecl_to_csfloat</code>	44
<code>ecl_init_module</code>	88	<code>ecl_to_double</code>	44
<code>ecl_long_float</code>	44	<code>ecl_to_fixnum</code>	44
<code>ecl_lower_case_p</code>	50	<code>ecl_to_float</code>	44
<code>ecl_make_cdfloat</code>	43	<code>ecl_to_int</code>	44
<code>ecl_make_clfloat</code>	43	<code>ecl_to_int16_t</code>	44
<code>ecl_make_complex</code>	43	<code>ecl_to_int32_t</code>	44
<code>ecl_make_constant_base_string</code>	60	<code>ecl_to_int64_t</code>	44
<code>ecl_make_csfloat</code>	43	<code>ecl_to_int8_t</code>	44
<code>ecl_make_double_float</code>	43	<code>ecl_to_long</code>	44
<code>ecl_make_fixnum</code>	43, 158	<code>ecl_to_long_double</code>	44
<code>ecl_make_int</code>	43	<code>ecl_to_long_long</code>	44
<code>ecl_make_int16_t</code>	43	<code>ecl_to_short</code>	44
<code>ecl_make_int32_t</code>	43	<code>ecl_to_uint</code>	44
<code>ecl_make_int64_t</code>	43	<code>ecl_to_uint16_t</code>	44
<code>ecl_make_int8_t</code>	43	<code>ecl_to_uint32_t</code>	44
<code>ecl_make_integer</code>	43	<code>ecl_to_uint64_t</code>	44
<code>ecl_make_keyword</code>	36	<code>ecl_to_uint8_t</code>	44
<code>ecl_make_lock</code>	126	<code>ecl_to_ulong</code>	44
<code>ecl_make_long</code>	43	<code>ecl_to_ulong_long</code>	44
<code>ecl_make_long_float</code>	43	<code>ecl_to_unsigned_integer</code>	44
<code>ecl_make_long_long</code>	43	<code>ecl_to_ushort</code>	44
<code>ecl_make_ratio</code>	43	<code>ecl_upper_case_p</code>	50
<code>ecl_make_rwlock</code>	128	<code>ecl_va_arg</code>	29
<code>ecl_make_semaphore</code>	129	<code>ecl_va_end</code>	29
<code>ecl_make_short_t</code>	43	<code>ecl_va_list</code>	29
<code>ecl_make_simple_base_string</code>	60	<code>ecl_va_start</code>	29
<code>ecl_make_single_float</code>	43	<code>ecl_vector</code>	159
<code>ecl_make_symbol</code>	37	<code>ECL_ADJUSTABLE_ARRAY_P</code>	159
<code>ecl_make_uint</code>	43	<code>ECL_ANSI_STREAM_P</code>	164
<code>ecl_make_uint16_t</code>	43	<code>ECL_ANSI_STREAM_TYPE_P</code>	164
<code>ecl_make_uint32_t</code>	43	<code>ECL_ARRAY_DIMENSION_LIMIT</code>	55
<code>ecl_make_uint64_t</code>	43	<code>ECL_ARRAY_HAS_FILL_POINTER_P</code>	159
<code>ecl_make_uint8_t</code>	43	<code>ECL_ARRAY_RANK_LIMIT</code>	55
<code>ecl_make_ulong</code>	43	<code>ECL_ARRAY_TOTAL_LIMIT</code>	55
<code>ecl_make_ulong_long</code>	43	<code>ECL_ARRAYP</code>	156
<code>ecl_make_unsigned_integer</code>	43	<code>ECL_ATOM</code>	156
<code>ecl_make_ushort_t</code>	43	<code>ECL_BASE_CHAR_CODE_P</code>	156
<code>ecl_nth_value</code>	30	<code>ECL_BASE_CHAR_P</code>	156
<code>ecl_nvalues</code>	30	<code>ECL_BASE_STRING_P</code>	162
<code>ecl_process_env</code>	25	<code>ECL_BIGNUMP</code>	156
<code>ecl_read_from_cstring</code>	157	<code>ECL_BIT_VECTOR_P</code>	156
<code>ecl_release_current_thread</code>	16	<code>ECL_BLOCK_BEGIN</code>	31
<code>ecl_return0</code>	31	<code>ECL_CATCH_ALL</code>	16
<code>ecl_return1</code>	31	<code>ECL_CATCH_BEGIN</code>	31
<code>ecl_return2</code>	31	<code>ECL_CHAR_CODE</code>	49, 159

ECL_CHAR_CODE_LIMIT	159
ECL_CHARACTERP	156
ECL_CLASS_CPL	164
ECL_CLASS_INFERIORS	164
ECL_CLASS_NAME	164
ECL_CLASS_OF	164
ECL_CLASS_SLOTS	164
ECL_CLASS_SUPERIORS	164
ECL_CODE_CHAR	49, 159
ECL_COMPLEXP	156
ECL_CONS_CAR	51
ECL_CONS_CDR	51
ECL_CONSP	156
ECL_DOUBLE_FLOAT_P	156
ECL_EXTENDED_STRING_P	162
ECL_FIXNUMP	156
ECL_FOREIGN_DATA_P	156
ECL_HANDLER_CASE	34
ECL_HASH_TABLE_P	156
ECL_IMMEDIATE	157
ECL_INSTANCEP	164
ECL_LISTP	156
ECL_LONG_FLOAT_P	156
ECL_NUMBER_TYPE_P	156
ECL_PACKAGEP	156
ECL_PATHNAMEP	156
ECL_RANDOM_STATE_P	156
ECL_READTABLEP	156
ECL_REAL_TYPE_P	156
ECL_RESTART_CASE	35
ECL_RPLACA	51
ECL_RPLACD	51
ECL_SINGLE_FLOAT_P	156
ECL_SPEC_FLAG	164
ECL_SPEC_OBJECT	164
ECL_SSE_PACK_P	156
ECL_STRINGP	156
ECL_STRUCT_LENGTH	164
ECL_STRUCT_NAME	164
ECL_STRUCT_SLOT	164
ECL_STRUCT_SLOTS	164
ECL_STRUCT_TYPE	164
ECL_SYMBOLP	156
ECL_UNWIND_PROTECT	17
ECL_UNWIND_PROTECT_BEGIN	31
ECL_VECTORP	156
ECL_WITH_LISP_FPE	18

F

fix	158
fixint	158
fixnint	158

M

MAKE_FIXNUM	158
MOST_NEGATIVE_FIXNUM	158
MOST_POSITIVE_FIXNUM	158
mp_all_processes	122
mp_block_signals	125
mp_condition_variable-broadcast	129
mp_condition_variable_signal	129
mp_condition_variable_timedwait	129
mp_condition_variable_wait	129
mp_current_process	125
mp_exit_process	122
mp_get_lock_nowait	127
mp_get_lock_wait	127
mp_get_rwlock_read_nowait	128
mp_get_rwlock_read_wait	128
mp_get_rwlock_write_nowait	128
mp_get_rwlock_write_wait	128
mp_giveup_lock	127
mp_giveup_rwlock_read	128
mp_giveup_rwlock_write	128
mp_holding_lock_p	127
mp_interrupt_process	122
mp_lock_count	127
mp_lock_name	127
mp_make_condition_variable	129
mp_make_process	123
mp_make_semaphore	129
mp_process-join	124
mp_process_active_p	123
mp_process_enable	123
mp_process_kill	124
mp_process_name	125
mp_process_preset	125
mp_process_resume	124
mp_process_run_function	125
mp_process_suspend	124
mp_process_yield	124
mp_recursive_lock_p	127
mp_restore_signals	125
mp_rwlock_name	128
mp_semaphore_count	130
mp_semaphore_name	129
mp_semaphore_wait_count	130
mp_signal_semaphore	130
mp_try_get_semaphore	130
mp_wait_on_semaphore	130

S

si_add_package_local_nickname	144
si_make_array	56
si_make_lambda	165
si_make_vector	56
si_package_local_nicknames	144
si_package_locally_nicknamed_by_list	144
si_remove_package_local_nickname	145

si_safe_eval 164
 si_string_to_object 157

T

t_array 156
 t_barrier 156
 t_base_string 156
 t_bclosure 156
 t_bignum 156
 t_bitvector 156
 t_bytecodes 156
 t_cclosure 156
 t_cdfloat 156
 t_cfun 156
 t_cfunfixed 156
 t_character 156
 t_clfloat 156
 t_codeblock 156
 t_complex 156
 t_condition_variable 156
 t_contiguous -- contiguous block 156
 t_csfloat 156
 t_end 156
 t_fixnum 156

t_foreign 156
 t_frame 156
 t_hashtable 156
 t_instance 156
 t_list 156
 t_lock 156
 t_longfloat 156
 t_mailbox 156
 t_other 156
 t_package 156
 t_pathname 156
 t_process 156
 t_random 156
 t_ratio 156
 t_readtable 156
 t_rwlock 156
 t_semaphore 156
 t_singlefloat 156
 t_sse_pack 156
 t_start 156
 t_stream 156
 t_string 156
 t_structure = t_instance 156
 t_symbol 156
 t_vector 156
 t_weak_pointer 156

Bibliography

- ANSI** ANSI Common-Lisp Specification, 1986.
- AMOP** Gregor Kiczales et al. “The Art of the Metaobject Protocol” The M.I.T. Press, Massachussets Institute of Technology, 1999
- LISP1.5** John McCarthy et al. “Lisp 1.5 Programmer’s Manual 2nd ed” The M.I.T. Press, Massachussets Institute of Technology, 1985
- Steele:84** Guy L. Steele Jr. et al. “Common Lisp: the Language”, Digital Press, 1984.
- Steele:90** Guy L. Steele Jr. at al. “Common Lisp: the Language II”, second edition, Digital Press, 1990.
- Yasa:85** Taiichi Yuasa and Masami Hagiya “Kyoto Common-Lisp Report”, Research Institute for Mathematical Sciences, Kyoto University, 1988.
- Attardi:95** Giuseppe Attardi “Embeddable Common-Lisp”, ACM Lisp Pointers, 8(1), 30-41, 1995
- Smith:84** B.C. Smith and J. des Rivieres “The Implementation of Procedurally Reflective Languages”, *Proc. of the 1984 ACM Symposium on LISP and Functional Programming*, 1984.

