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

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
1.2 Entering and leaving Embeddable Common Lisp 10
1.3 The break loop 12
1.4 Embedding ECL 12

2 Developer's guide **13**
2.1 Sources structure 13
2.1.1 src/c 13
2.2 Contributing 16
2.3 Manipulating Lisp objects 16
2.3.1 Objects representation 17
2.3.2 Constructing objects 19
2.4 Environment implementation 27
2.5 Removed features 27

3 Standards **29**
3.1 Overview 29
3.1.1 Reading this manual 29
3.1.2 C Reference 29
3.2 Evaluation and compilation 31
3.2.1 Compiler declaration `OPTIMIZE` 31
3.2.2 C Reference 32
3.3 Types and classes 32
3.4 Data and control flow 33
3.4.1 Shadowed bindings 33

3.4.2 Minimal compilation ... 33
3.4.3 Function types ... 34
3.5 Hash tables ... 34

4 Extensions ... 37

4.1 System building ... 37
4.1.1 Compiling with ECL ... 37
4.1.1.1 Portable FASL ... 38
4.1.1.2 Native FASL ... 39
4.1.1.3 Object file ... 40
4.1.1.4 Static library ... 40
4.1.1.5 Shared library ... 41
4.1.1.6 Executable ... 41
4.1.1.7 Summary ... 42
4.1.2 Compiling with ASDF ... 42
4.1.2.1 Example code to build ... 42
4.1.2.2 Build it as an single executable ... 43
4.1.2.3 Build it as shared library and use in C ... 43
4.1.2.4 Build it as static library and use in C ... 44
4.2 Operating System Interface ... 45
4.2.1 Command line arguments ... 45
4.2.2 External processes ... 47
4.2.3 Operating System Interface Reference ... 49
4.3 Foreign Function Interface ... 49
4.3.1 What is a FFI? ... 49
4.3.2 Two kinds of FFI ... 50
4.3.3 Foreign objects ... 51
4.3.4 Higher level interfaces ... 52
4.3.5 SFFI Reference ... 54
4.3.6 UFFI Reference ... 58
4.3.6.1 Primitive Types ... 58
4.3.6.2 Aggregate Types ... 60
4.3.6.3 Foreign Objects ... 63
4.3.6.4 Foreign Strings ... 68
4.3.6.5 Functions and Libraries ... 72
4.4 Native threads ... 74
4.4.1 Tasks, threads or processes ... 74
4.4.2 Processes (native threads) ... 74
4.4.3 Processes dictionary ... 75
4.4.4 Locks (mutexes) ... 78
4.4.5 Locks dictionary ... 79
4.4.6 Readers-writer locks ... 80
4.4.7 Read-Write locks dictionary ... 80
4.4.8 Condition variables ... 81
4.4.9 Condition variables dictionary ... 81
4.4.10 Semaphores ... 81
4.4.11 Semaphores dictionary ... 81
4.5 Signals and Interrupts ... 82

4.6 Memory Management ... 82
4.7 Meta-Object Protocol (MOP) ... 82
4.8 Gray Streams ... 82
4.9 Tree walker ... 82
4.10 Package locks ... 82
4.10.1 Package Locking Overview ... 82
4.10.2 Operations Violating Package Locks ... 83
4.10.3 Package Lock Dictionary ... 83
4.11 CDR Extensions ... 84

Indexes ... 85

Concept index ... 85
Configure option index ... 85
Feature index ... 87
Example index ... 87
Function index ... 88
Variable index ... 90
Type index ... 90
Common Lisp symbols ... 91
C/C++ index ... 92

Bibliography ... 95

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 95] and [Steele:84, see [Bibliography], page 95]. Chapter 2 [Developer's guide], page 13, and Chapter 1 [User's guide], page 9, for the details about the implementation and how to interface with other languages.

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 95], [AMOP, see [Bibliography], page 95],...), 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 2 [Developer's guide], page 13] documents Embeddable Common Lisp implementation details. This part isn 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 3 [Standards], page 29] 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 4 [Extensions], page 37] 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 95] developed by John McCarthy in

the 80s. Common-Lisp as we know it ANSI Common-Lisp is the result of an standarization 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 95]. [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 95], the ECL is now compliant with X3J13 ANSI Common Lisp [ANSI, see [Bibliography], page 95].

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 95]. This implementation was developed partially in C and partially in Common Lisp itself and featured a lisp to C translator.

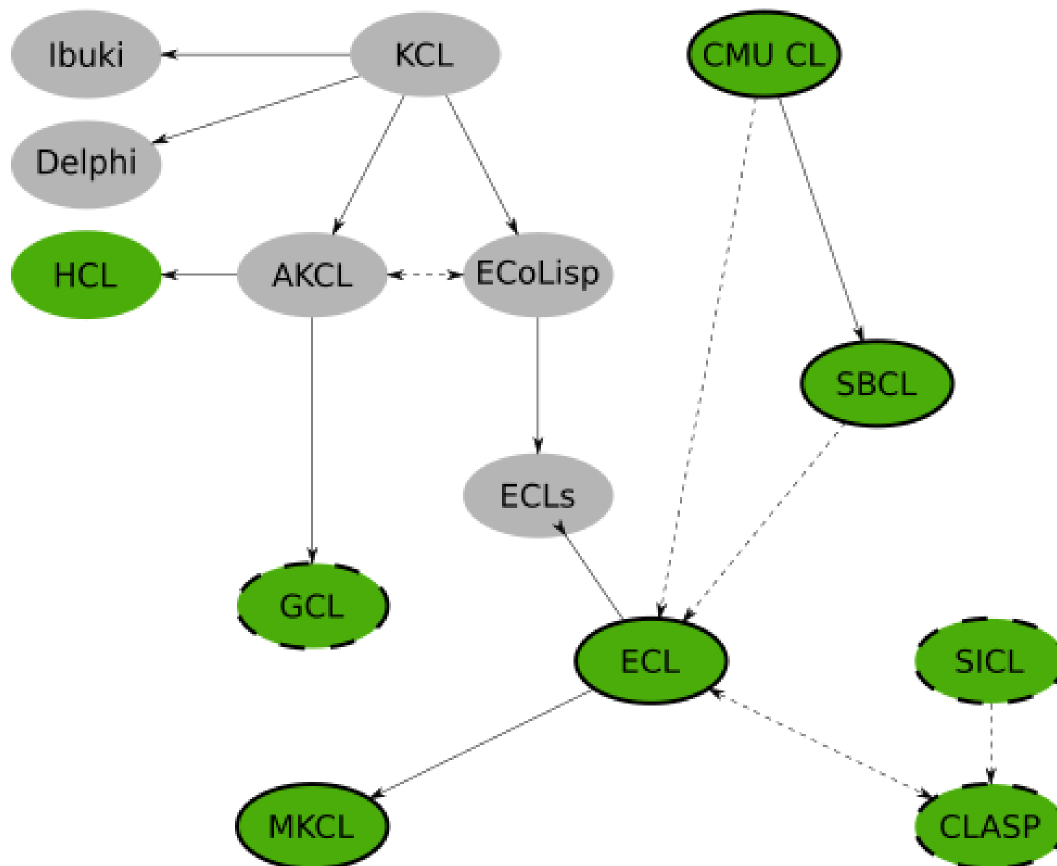

Figure 1: ECL's family tree

The KCL implementation remained a propietary 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 propietary 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, Giusseppe 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 95]. 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 95], 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 conformat 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 95]

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 (Toplevel, 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 Spilsbury.

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) 2015, Daniel Kochmański
Copyright (c) 2000, 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, 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 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

If you have a commercial version of Microsoft Visual Studio, the steps are simple:

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. Optionally, if you want to build a self-installing executable, you can install NSIS and run nmake windows-nsi.

If you want to build ECL using the free Microsoft Visual Studio Express 2013 or better, you should follow these before building ECL as sketched before:

1. Download and install Microsoft Visual Studio C++ Compiler.
2. Download and install the Windows SDK
3. Open the Windows SDK terminal, which will set up the appropriate paths and environment variables.

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 `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 95]. 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 signalled, 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 BAR 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 `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 12] 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

2 Developer's guide

2.1 Sources structure

2.1.1 src/c

alloc_2.d	memory allocation based on the Boehm GC
all_symbols.d	name mangler and symbol initialization
apply.d	interface to C call mechanism
arch/*	architecture dependant code
array.d	array routines
assignment.c	assignment
backq.d	backquote mechanism
big.d	bignum routines based on the GMP
big_ll.d	bignum emulation with long long
cfun.d	compiled functions
cfun_dispatch.d	trampolines for functions
character.d	character routines
char_ctype.d	character properties.
cinit.d	lisp initialization
clos/accessor.d	dispatch for slots
clos/cache.d	thread-local cache for a variety of operations
cmpaux.d	auxiliaries used in compiled Lisp code
compiler.d	bytecode compiler
disassembler.d	bytecodes disassembler utilities
dpp.c	defun preprocessor

ecl_constants.h	contstant values for all_symbols.d
features.h	names of features compiled into ECL
error.d	error handling
eval.d	evaluation
ffi/backtrace.d	C backtraces
ffi/cdata.d	data for compiled files
ffi/libraries.d	shared library and bundle opening / copying / closing
ffi/mmap.d	mapping of binary files
ffi.d	user defined data types and foreign functions interface
file.d	file interface (implementation dependent)
format.d	format (this isn't ANSI compliant, we need it for bootstrapping though)
gfun.d	dispatch for generic functions
hash.d	hash tables
instance.d	CLOS interface
interpreter.d	bytecode interpreter
iso_latin_names.h	character names in ISO-LATIN-1
list.d	list manipulating routines
load.d	binary loader (contains also open_fasl_data)
macros.d	macros and environment
main.d	ecl boot proccess
Makefile.in	Makefile for ECL core library
mapfun.d	mapping
newhash.d	hashing routines

num_arith.d arithmetic operations

number.d constructing numbers

numbers/*.d arithmetic operations (abs, atan, plusp etc)

num_co.d operations on floating-point numbers (implementation dependent)

num_log.d logical operations on numbers

num_pred.d predicates on numbers

num_rand.d random numbers

package.d packages (OS dependent)

pathname.d pathnames

predicate.d predicates

print.d print

printer/*.d printer utilities and object representations

read.d read.d - reader

reader/parse_integer.d
reader/parse_number.d

reference.d reference in Constants and Variables

sequence.d sequence routines

serialize.d serialize a bunch of lisp data

sse2.d SSE2 vector type support

stacks.d binding/history/frame stacks

string.d string routines

structure.d structure interface

symbol.d symbols

symbols_list.h

symbols_list2.h	The latter is generated from the first. The first has to contain all symbols on the system which aren't local.
tcp.d	stream interface to TCP
time.d	time routines
typespec.d	type specifier routines
unicode/*	unicode definitions
unixfsys.d	Unix file system interface
unixsys.d	Unix shell interface
vector_push.d	vector optimizations

threads/

atomic.d	atomic operations
barrier.d	wait barriers
condition_variable.d	condition variables for native threads
ecl_atomics.h	alternative definitions for atomic operations
mailbox.d	thread communication queue
mutex.d	mutually exclusive locks.
process.d	native threads
queue.d	waiting queue for threads
rwlock.d	POSIX read-write locks
semaphore.d	POSIX-like semaphores

2.2 Contributing

2.3 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 complex symbol pack hash array vector base_string string stream random readtable pathname bytecodes bclosure cfun cfunfixed cclosure d instance process queue lock rwlock condition_variable semaphore barrier mailbox cblock foreign frame weak sse* [C/C++ identifier]

Union containing all first-class ECL types.

2.3.1 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.

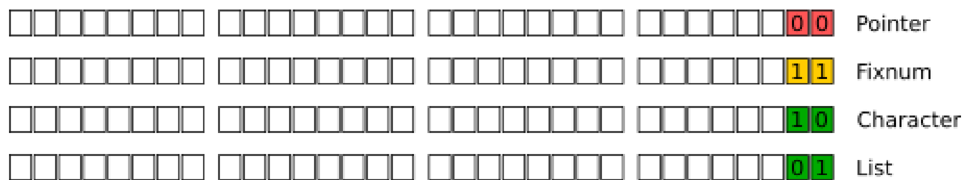

Figure 2.1: Immediate types

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 on 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", fix(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->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`.

`bool ECL_FIXNUMP` (*cl_object o*) [Function]
`bool ECL_CHARACTERP` (*cl_object o*) [Function]
`bool ECL_BASE_CHAR_P` (*cl_object o*) [Function]
`bool ECL_CODE_CHAR_P` (*cl_object o*) [Function]
`bool ECL_BASE_CHAR_CODE_P` (*cl_object o*) [Function]
`bool ECL_NUMBER_TYPE_P` (*cl_object o*) [Function]
`bool ECL_REAL_TYPE_P` (*cl_object o*) [Function]
`bool ECL_CONSP` (*cl_object o*) [Function]
`bool ECL_LISTP` (*cl_object o*) [Function]
`bool ECL_ATOM` (*cl_object o*) [Function]
`bool ECL_SYMBOLP` (*cl_object o*) [Function]
`bool ECL_ARRAYP` (*cl_object o*) [Function]
`bool ECL_VECTORP` (*cl_object o*) [Function]
`bool ECL_BIT_VECTOR_P` (*cl_object o*) [Function]
`bool ECL_STRINGP` (*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.

2.3.2 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.

`cl_object c_string_to_object` (*const char *s*) [Function]
`cl_object string_to_object` (*cl_object o*) [Function]

`c_string_to_object` builds a lisp object from a C string which contains a suitable representation of a lisp object. `string_to_object` performs the same task, but uses a lisp string, and therefore it is less useful.

Example

Using a C string

```
cl_object array1 = c_string_to_object("#(1 2 3 4)");
```

Using a Lisp string

```
cl_object string = make_simple_string("#(1 2 3 4)");
cl_object array2 = 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_plusp` (*cl_fixnum a*) [Function]
`bool ecl_fixnum_minusp` (*cl_fixnum a*) [Function]

Operations on `fixnums` (comparison and predicates).

`cl_object ecl_make_fixnum` (*cl_fixnum n*) [Function]
`cl_fixnum ecl_unfix` (*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 `cl_make_fixnum`
- **DEPRECATED** `fix` – equivalent to `cl_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`.

`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** `CODE_CHAR_LIMIT` – equivalent to `ECL_CHAR_CODE_LIMIT`

`cl_fixnum ECL_CHAR_CODE` (*cl_object o*) [Function]
`cl_fixnum ECL_CODE_CHAR` (*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 it's argument. `ecl_base_char_code` is an optimized version for base chars. Checks it's 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.

`bool ecl_char_cmp` (*cl_object x, cl_object y*) [Function]
`bool ecl_char_compare` (*cl_object x, cl_object y*) [Function]

Compare the relative order of two characters. `char_cmp` takes care of case and `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 functions.

`ecl_vector` [C/C++ identifier]

If `x` contains a vector, you can access the following fields:

`x->vector.elttype`
: The type of the elements of the vector.

`x->vector.displaced`
: Boolean indicating if it is displaced.

`x->vector.dim`
: The maximum number of elements.

`x->vector.fillp`
: Actual numer 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.elttype*.

`ecl_array` [C/C++ identifier]

If `x` contains a multidimensional array, you can access the following fields:

`x->array.elttype`
: The type of the elements of the array.

`x->array.rank`
: The number of array dimensions.

`x->array.displaced`
: Boolean indicating if it is displaced.

`x->vector.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 numer 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.elttype*.

`cl_elttype` *ecl_aet_object ecl_aet_sf ecl_aet_df 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* [C/C++ identifier]

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.elttype` or `x->vector.elttype`. We list some of those types together with the C constant that denotes that type:

T `ecl_aet_object`

BASE-CHAR `ecl_aet_object`

SIGNLE-FLOAT `ecl_aet_sf`

DOUBLE-FLOAT `ecl_aet_df`

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(c_string_to_object("\"AAA\""));  /* returns ecl_aet_ch */
ecl_array_elttype(c_string_to_object("#(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 = c_string_to_object("#2A((1 2) (3 4))");
cl_object x = aref(array, 3);
cl_print(1, x); /* Outputs 4 */
aset(array, 3, MAKE_FIXNUM(5));
cl_print(1, array); /* Outputs #2A((1 2) (3 5)) */
```

`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 similar to aref and aset, but they operate on vectors.

Example

```
cl_object array = c_string_to_object("#(1 2 3 4)");
cl_object x = aref1(array, 3);
cl_print(1, x);     /* Outputs 4 */
aset1(array, 3, 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 integers and chars).

`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 clear_input clear_output finish_output force_output input_p output_p interactive_p element_type length get_position set_position column close* [C/C++ identifier]

`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. Doesn't check type of it's argument.

`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_bytecode are implicitly constructed by a call to eval, but can also be explicitly constructed with the make_lambda function.

`cl_object si_safe_eval` (*cl_object form, cl_object env, ...*) [Function]

`si_safe_eval` evaluates `form` in the lexical environment `env`, which can be *ECL_NIL*. Before evaluating it, the expression form must be bytecompiled.

DEPRECATED `cl_object cl_eval (cl_object form)`

`cl_eval` is the equivalent of `si_safe_eval` but without environment and with err_value set to nil. It exists only for compatibility with previous versions.

DEPRECATED `cl_object cl_safe_eval (cl_object form)`

Equivalent of `si_safe_eval` (macro define).

Exmaple

```
si_object form = c_string_to_object("(print 1)");
si_safe_eval(form, ECL_NIL);
si_safe_eval(form, ECL_NIL, 3); /* on error function will return 3 */
```

`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 = c_string_to_object("((x y) (+ x y))");
cl_object name = _intern("foo")
cl_object fun = si_make_lambda(name, def);
return funcall(fun, MAKE_FIXNUM(1), 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.

2.4 Environment implementation

2.5 Removed features

In-house DFFI

Commit `10bd3b613fd389da7640902c2b88a6e36088c920`. Native DFFI was replaced by a `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 superseeded 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 unifiction module (logic programming) from EcoLisp times.

3 Standards

3.1 Overview

3.1.1 Reading this manual

Common Lisp users

Embeddable Common Lisp supports all Common-Lisp data types exactly as defined in the [ANSI, see [Bibliography], page 95]. All functions and macros are expected to behave as described in that document and in the HyperSpec [HyperSpec, see [Bibliography], page 95] which is the online version of [ANSI, see [Bibliography], page 95]. 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 95].
- 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 4.1 [System building], page 37), interface with foreign libraries (Section 4.3 [Foreign Function Interface], page 49), manage memory (Section 4.6 [Memory Management], page 82), etc. These concepts are explained in a different (Section 1.4 [Embedding ECL], page 12) part of the book.

3.1.2 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 4.6 [Memory Management], page 82. For object implementation details See Section 2.3 [Manipulating Lisp objects], page 16.

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`) 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 narg, ...)`, 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 3 [Standards], page 29] 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();
CL_UNWIND_PROTECT_BEGIN(env) {
    /* protected code goes here */
} CL_UNWIND_PROTECT_EXIT {
    /* exit code goes here */
} CL_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.

3.2 Evaluation and compilation

3.2.1 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 compie 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 3.1, 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 3.1: Behavior for different levels of `DEBUG`

A bit more critical is the value of `SAFETY` because as shown in Table 3.2, 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`, 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 `CAR`, `CDR`, etc	Y	Y	N	N
We believe types defined before compiling a file 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 `PATHNAME-TYPE`, `CAR`, `REST`, etc.	Y	N	N	N

Table 3.2: Behavior for different levels of `SAFETY`

3.2.2 C Reference

`cl_object cl_env_ptr ()` [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.

3.3 Types and classes

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

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

3.4 Data and control flow

3.4.1 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 the error is signalled.

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

3.4.2 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.

3.4.3 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 drops the source code of a function unless the global variable si:*keep-definitions* was true when the function was translated into bytecodes. Therefore, if you wish to use compile and disassemble on defined functions, you should issue `(setq si:*keep-definitions* t)` at the beginning of your session.

`*keep-definitions*` [SI]

If set to `T` ECL will preserve the compiled function source code for disassembly and recompilation.

In Table 3.3 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 3.3: Function related constants

3.5 Hash tables

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.

`hash-table-weakness` *ht* [ext]

Returns type of the hash table weakness. Possible return values are: `:key`, `:value`, `:key-and-value`, `:key-or-value` or `NIL`.

Thread-safe hash tables

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

`hash-table-synchronized-p` *ht* [ext]

Predicate answering whenever hash table is synchronized or not.

Hash tables serialization

`hash-table-content` *ht* [ext]

Returns freshly consed list of pairs `(key . val)` being contents of the hash table.

`hash-table-fill` *ht values* [ext]

Fills `ht` with `values` being list of `(key . val)`. Hash table may have some content already, but conflicting keys will be overwritten.

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>))
```

4 Extensions

4.1 System building

4.1.1 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 4.3 [Foreign Function Interface], page 49. We explain file types generated by some compilation approaches. GNU/Linux system and gcc as a development environment are assumed.

You can generate 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:

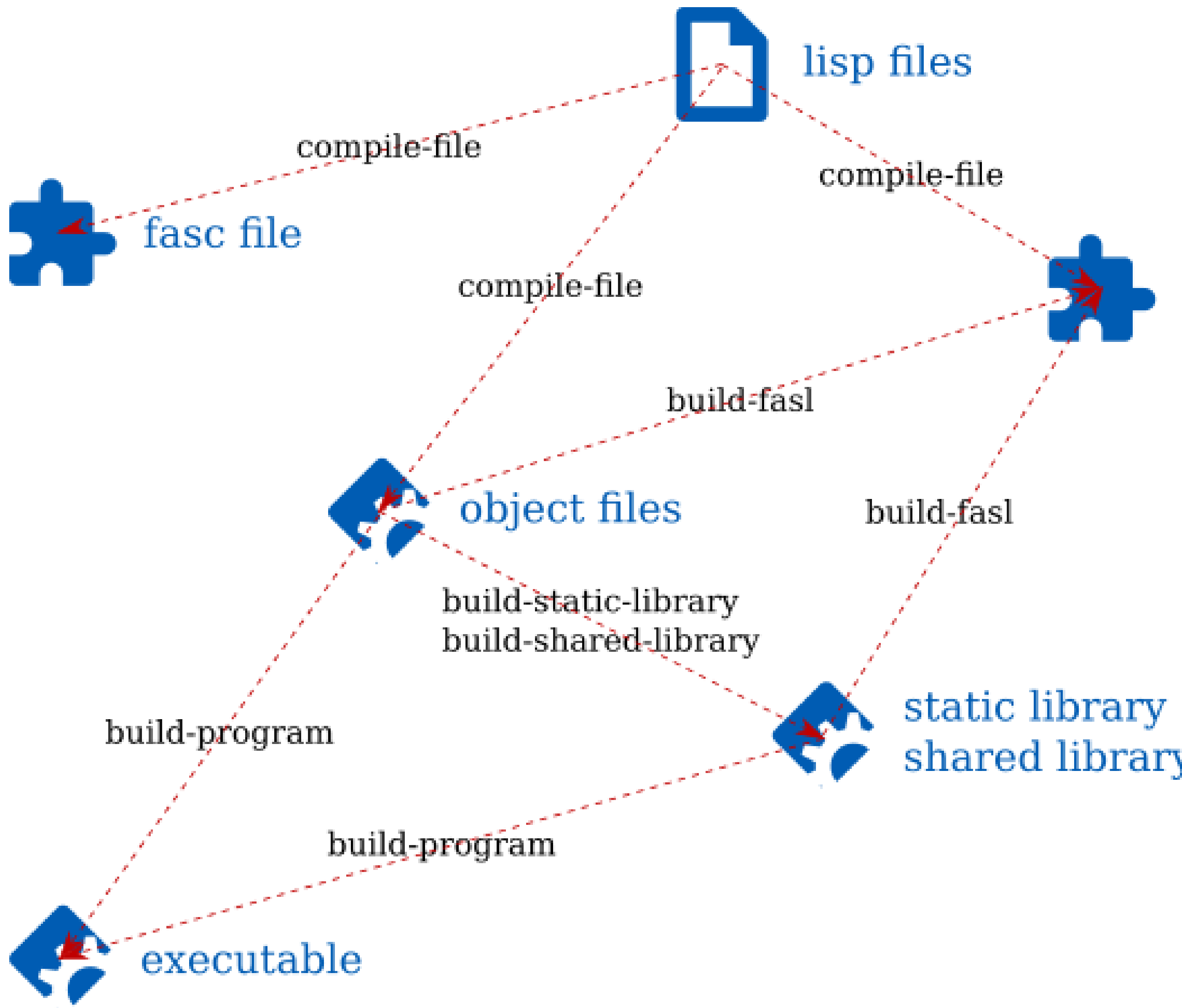

Figure 4.1: Build file types

4.1.1.1 Portable FASL

ECL provides two compilers (bytecodes compiler, and C/C++ compieler). Portable FASL files are build from source lisp files by the bytecodes compiler. Generally FASC files are portable across architectures and operating systems providing 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")
```

4.1.1.2 Native FASL

If you want to make a library which is loaded dynamically from lisp program, you should choose fasl file format. Under the hood native fasls are just a 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. Recommended usage is loading fasl files by calling 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"))
```

4.1.1.3 Object file

Object file works 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 for generating 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 executable), and in each approach, object files act as intermediate files.

4.1.1.4 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 static/shared library, you have to call init functions. The name of the function is specified by `:init-name` option. In this example, `init_hello_goodbye` is it. 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 read_VV */
    read_VV(OBJNULL, 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, invoke `init_hello_goodbye` via `read_VV`. `init_hello_goodbye` takes a argument, and read_VV supplies an appropriate one. Now that the initialization is finished, we can use functions and other stuffs defined in the library.

4.1.1.5 Shared library

Almost the same as the case of static library. 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")
```

4.1.1.6 Executable

ECL supports executable file generation. To create a standalone executable from lisp programs, 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 native FASL, program may be built also from libraries.

4.1.1.7 Summary

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

- Object file: intermediate file format for others
- Fasl files: loaded dynamically via 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* bytecodes compiler is needed.

4.1.2 Compiling with ASDF

Besides the simple situation that we write Lisp without depending on any other Lisp libraries, a more practical example is build a library depends on other asdf systems or Quicklisp projects. ECL provides a useful extension for asdf called `asdf:make-build`, it's almost as easy as build a library without dependencies. Because Quicklisp also uses asdf to load systems with dependencies, just make sure you have successfully load and run your library in ECL REPL (or `*slime-repl*`). Don't worry Quicklisp, asdf, swank and other unused libraries are packed into the executable or library, ECL will only build and pack libraries your project depends on (that is, all dependence you put in your `.asd` file, and their dependencies, nothing more even you are build in a image already load with lots of other libraries).

4.1.2.1 Example code to build

We use a simple project depends on alexandria to demonstrate the steps. Consists of `example-with-dep.asd`, `package.lisp` and `example.lisp`. For convenience, we list these files here:

```
;;;; example-with-dep.asd
(defsystem :example-with-dep
  :serial t
  :depends-on (:alexandria)
  :components ((:file "package")
       (:file "example")))
```

```
;;;; package.lisp
(in-package :cl-user)

(defpackage :example
  (:use :cl)
  (:export :test-function))
```

```
;;;; example.lisp
(in-package :example)

(defun test-function (n)
  (format t "Factorial of ~a is: ~a~%" n (alexandria:factorial n)))
```

Before any kind you build, you need to push full path of this directory (`asdf_with_dependence/`) into `asdf:*central-registry*`.

4.1.2.2 Build it as an single executable

Use this in REPL to make a 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 what to do after loading our library, we can use arbitrary Lisp forms here. You can also write this code in your Lisp files and directly build them without this `:epilogue-code` option to have the same effect. Run the program in console will display the following and exit:

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

4.1.2.3 Build it as shared library and use in C

Use this in REPL to make a shared library:

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

Here `:monolithic t` means to let ECL solve dependence and build all dependence into one library named `example-with-dep--all-systems.so` in this directory.

To use it, we use a simple C program:

```
/* test.c */
#include <ecl/ecl.h>

int main (int argc, char **argv) {
  extern void init_dll_EXAMPLE_WITH_DEP__ALL_SYSTEMS(cl_object);

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

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

  cl_shutdown();
  return 0;
}
```

Note the name convention here: an asdf system named `example-with-dep` will compiled to `example-with-dep--all-systems.so` and in the C code should be init with `init_dll_EXAMPLE_WITH_DEP__ALL_SYSTEMS`. Compile it using:

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

ECL's library path and current directory may not be in your `LD_LIBRARY_PATH`, so call `./test` using:

```
LD_LIBRARY_PATH=/usr/local/lib/:. ./test
```

This will show:

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

You can also build all dependent libraries separately as several `.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 also do these in ECL REPL:

```
(asdf:make-build :complex-example
 :type :shared-library
 :move-here #P"./")
(asdf:make-build :alexandria
 :type :shared-library
 :move-here #P"./")
(asdf:make-build :cl-fad
 :type :shared-library
 :move-here #P"./")
(asdf:make-build :bordeaux-threads
 :type :shared-library
 :move-here #P"./")
```

Note here is no `:monolithic t` and we also need to build `bordeaux-threads` because `cl-fad` depends on it. The building sequence doesn't matter and the result `.so` files can also be used in your future program if these libraries are not modified. And We need to initialize all these modules using `ecl_init_module`, the name convention is to initialize `cl-fad` you need:

```
extern void init_dll_CL_FAD(cl_object);

/* after cl_boot(argc, argv);
   and if B depends on A, you should first init A then B. */
ecl_init_module(NULL, init_dll_CL_FAD);
```

You can easily figure out name conventions with other libraries.

4.1.2.4 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)
```

That will generate a `example-with-dep--all-systems.a` in current directory and we need to replace `init_dll_EXAMPLE_WITH_DEP__ALL_SYSTEMS` with `init_lib_EXAMPLE_WITH_DEP__ALL_SYSTEMS`. (The code is given in test-static.c) And compile it using:

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

Then run it:

```
LD_LIBRARY_PATH=/usr/local/lib/ ./test-static
```

Note we don't need to give current path in `LD_LIBRARY_PATH` here, since our Lisp library is statically bundled to the executable. The result is same as the shared library example above. You can also build all dependent libraries separately to static libraries. To use them you also need replace names like `init_dll_CL_FAD` to `init_lib_CL_FAD`.

4.2 Operating System Interface

4.2.1 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 ths filename to be resolved.

`ext:process-command-args` **&key** *args rules* [Function]

`args` [argument]

A list of strings. Defaults to the output of `ext:command-args`.

`rules` [argument]

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` [opt]

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

`nargs` [opt]
: A non-negative integer denoting the number of arguments taken by this option.

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

`:STOP` [opt]
: 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` [opt]
`:LOADRC` [opt]
: Determine whether the lisp initalization file `(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 succeeeds.

A special name `*DEFAULT*`, matches any unknown command line option. If there is no `*DEFAULT*` rule and no match is found, an error is signalled. 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 occurences of `:noloadrc`, one of the files 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 `ecl/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)
```

4.2.2 External processes

ECL provides several facilities for invoking and communicating with `ext:external-process`. 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 proces 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` *proces 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 environ if-input-does-not-exist if-output-exists if-error-exists external-format #+windows escape-arguments* [Function]

`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).

`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 ⟨undefined⟩ [Finalization], page ⟨undefined⟩, 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` [argument]
: 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 - stream must be ANSI stream (no in-memory virtual streams). If `:stream`, the `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` [argument]
: can be one of: `:error` to generate an error `:create` to create an empty file `nil` (the default) to return nil from `run-program`

`output` [argument]
: 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 - stream must be ANSI stream (no in-memory virtual streams). If `:stream`, the `external-process-output` slot is filled in with a stream that can be read to get the output. Defaults to `stream`.

`if-output-exists` [argument]

`error` [argument]
: 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` [argument]
: Same as `:if-output-exists`.

`wait` [argument]
: If non-NIL (default), wait until the created process finishes. If `nil`, continue running Lisp until the program finishes.

`environ` [argument]
: 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` [argument]
The external-format to use for `:input`, `:output`, and `:error` STREAMs.

Windows specific options:

`escape-arguments` [argument]
Controls escaping of the arguments passed to CreateProcess.

4.2.3 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` [Function]
`ext:chdir` [Function]
`ext:file-kind` [Function]
`ext:copy-file` [Function]
`ext:chmod` [Function]
Common operating system functions.

4.3 Foreign Function Interface

4.3.1 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 (Section 3.2, Section 3.3), and of the higher level interface (Section 3.4). It is highly recommended that you read all sections.

4.3.2 Two kinds of FFI

ECL allows for two different appraoches 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 invoked.

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.

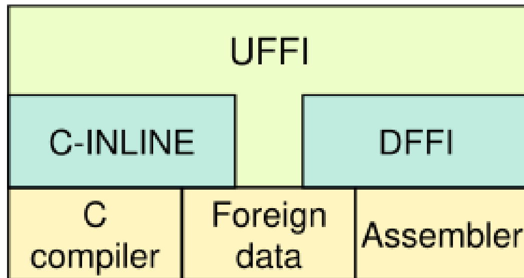

Figure 4.2: FFI components

As you see, the first approach uses rather portable technices 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**.

4.3.3 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 (Section 3.4).

The most important component of the object is the memory region where data is stored. By default ECL assumes that the user will perform automatic 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 provide tools for doing this. For instance, in the following example adapted from the UFFI documentation, the string NAME is automatically deallocated

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

(if (zerop (c-gethostname (ffi:char-array-to-pointer name) 256))
    (format t "Hostname: ~S" (ffi:convert-from-foreign-string name))
    (error "gethostname() failed."))
```

4.3.4 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 >=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.
- The 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 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.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)
(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:clines` and `ffi:c-inline`.

```
#|
Build and load this module with (compile-file "ecl.lsp" :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:clines "#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)))
```

4.3.5 SFFI Reference

Reference

`ffi:clines` *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:CLINES` 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)" :on-liner t))
```

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

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 true 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 true, 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 it's 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))
    (ffi: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 (one of the symbols OBJECT, INT, CHAR, CHAR*, FLOAT, DOUBLE).

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 FUNCTION-NAME.

The interpreter ignores this form. ARG-TYPES are argument types of the C function and RESULT-TYPE is its return type. Symbols OBJECT, INT, CHAR, CHAR*, FLOAT, DOUBLE are allowed for these types.

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

Provide Lisp alternative for interpreted code.

Description

Used to DEFine Lisp Alternative. For the interpreter, DEFLA is equivalent to DEFUN, but the compiler ignores this form.

4.3.6 UFFI Reference

4.3.6.1 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).

'`: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.

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

Creats 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

```
(def-foreign-type my-generic-pointer :pointer-void)
(def-foreign-type a-double-float :double-float)
(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
```

4.3.6.2 Aggregate Types

Overview

Aggregate types are comprised of one or more primitive types.

Reference

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

Defines a C enumeration

name A symbol that names the enumeration.

fields A list of field defintions. 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. It the field definition is a list, the first position is the symbol and the second position is the value to assign the 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 the 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 defintions. 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. It's 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 A name of the foreign structure.

field A name of the desired field in 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`-able.

Examples

```
(get-slot-value foo-ptr 'foo-structure 'field-name)
(setf (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 A name of the foreign structure.

field A name of the desired field in foreign structure.

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

Description

This is similar to get-slot-value. It is used when the value of a slot is a pointer type.

Examples

```
(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

```
(def-array-pointer byte-array-pointer :unsigned-char)
```

Side effects

Defines a new foreign type.

`ffi:deref-array` *array type position* [Function]

Deference 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 fields-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.

4.3.6.3 Foreign Objects

Overview

Objects are entities that can 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]

Deferences 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 `WITH-CAST-POINTER` together with the `DEREF-POINTER`/`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 are 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.

`+null-cstring-pointer+` [FFI]

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* is will be bound to this value during the execution of *BODY*.

Examples

```
(ffi:with-foreign-object (size :int)
  ;; FOO 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 specificying 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 in case-insensitive Lisp implementations.

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 A string specifying the module (or library) the foreign variable resides in.

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
```

4.3.6.4 Foreign Strings

Overview

UFFI has functions to two types of C-compatible strings: `cstring` 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 `cstring`

returns 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 Lisp string

returns `cstring`

Description

Converts a Lisp string to a cstring. The cstring should be freed with 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 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* (*null-terminated-p* **t**) [Macro]

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 translated ASCII and binary strings.

`ffi:convert-to-foreign-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 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 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-string`s created from conversion of a *string*s. Automatically frees the *foreign-string*s. This macro works similar to `LET*`. Based on `with-foreign-string`.

4.3.6.5 Functions and Libraries

Reference

`ffi:def-function` *name args* **&key** *module* (*returning* **:void**) (*call* **:cdecl**) [Macro]

name A string or list specificying 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 A string specifying which module (or library) that the foreign function resides.

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. If `:void` indicates module does not return any value.

Description

Declares a foreign function.

Examples

```
(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 types* [Function]

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"
```

4.4 Native threads

4.4.1 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 avaiable. The use of native threads has advantanges. 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.

4.4.2 Processes (native threads)

Process is a primitive representing native thread.

4.4.3 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 has expired before this copy is returned.

`cl_object mp_all_processes` () *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.

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 narg, ...*) [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`, set of variable bindings `initial-bindings` and no function to run. See also `mp:process-run-function`. Returns newly created process.

`cl_object mp_make_process` (*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.

```
(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. If `process` is the current thread, signal an error with.

`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 narg, 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:enable-process` and `mp:process-run-function`.

`cl_object mp_process_run_function` (*cl_narg narg, cl_object name, cl_object function, ...*) [Function]

`mp:process_run_function` *name function* **&rest** *funciton-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` [Function]

Returns 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:restor-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-thread`. 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 `allow-with-interrupts`, `with-local-interrupts` and `with-restored-interrupts` as a local macros.

`with-restored-interrupts` executes the body with interrupts enabled if and only if the `without-interrupts` was in an environment in which interrupts were allowed.

`allow-with-interrupts` allows the `with-interrupts` to take effect during the dynamic scope of its body, unless there is an outer `without-interrupts` without a corresponding `allow-with-interrupts`.

`with-local-interrupts` executes its body with interrupts enabled provided that for there is an `allow-with-interrupts` for every `without-interrupts` surrounding the current one. `with-local-interrupts` is equivalent to:

```
(allow-with-interrupts (with-interrupts ...))
```

Care must be taken not to let either `allow-with-interrupts` or `with-local-interrupts` appear in a function that escapes from inside the `without-interrupts` in:

```
(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 () (allow-with-interrupts ...)))

(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 () (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 `with-interrupts` only makes sense if there is an outer `without-interrupts` with a corresponding `allow-with-interrupts`: interrupts are not enabled if any outer `without-interrupts` is not accompanied by `allow-with-interrupts`.

4.4.4 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).

4.4.5 Locks dictionary

`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 `name`. If `recursive` isn't `nil`, then the created lock is recursive.

`cl_object mp_recursive_lock_p` (*cl_object lock*) [Function]

`mp:recursive-lock-p` *lock* [Function]

Predicate verifying if `lock` is recursive.

`cl_object mp_holding_lock_p` (*cl_object lock*) [Function]

`mp:holding-lock-p` *lock* [Function]

Predicate verifying if the current thread holds `lock`.

`cl_object mp_lock_name` (*cl_object lock*) [Function]

`mp:lock_name` *lock* [Function]

Returns `lock` name.

`cl_object mp_lock_owner` (*cl_object lock*) [Function]

`mp:lock_owner` *lock* [Function]

Returns process owning `lock` (or `nil` if it is free). For testing whether the current thread is holding a lock see `holding-lock-p`.

`cl_object mp_lock_count` (*cl_object lock*) [Function]

`mp:lock-count` *lock* [Function]

Returns number of processes waiting for `lock`.

`cl_object mp_get_lock_wait` (*cl_object lock*) [Function]

Grabs a lock (blocking if `lock` is already takene). Returns `ECL_T`.

`cl_object mp_get_lock_nowait` [Function]

Grabs a lock if free (non-blocking). If `lock` is already taken returns `ECL_NIL`, otherwise `ECL_T`.

`mp:get-lock` *lock* **&optional** (*wait* **t**) [Function]

Tries to acquire a lock. `wait` indicates whenever function should block or give up if `lock` is already taken. If `wait` is `nil` and `lock` can't be acquired returns `nil`. Succesful operation returns `t`.

`cl_object mp_giveup_lock` (*cl_object lock*) [Function]

`mp:giveup_lock` [Function]

Releases `lock`.

`mp:with-lock` (*lock-form*) **&body** *body* [Macro]

Acquire lock for the dynamic scope of `body`, which is executed with the lock held by current thread, and `with-lock` returns the values of body.

4.4.6 Readers-writer locks

Readers-writer (or shared-exclusive) lock allows concurrent access for read-only operations and write operations require exclusive access. `mp:rwlock` is non-recursive.

4.4.7 Read-Write locks dictionary

`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 with `name`.

`cl_object mp_rwlock_name` (*cl_object lock*) [Function]

`mp:rwlock_name` *lock* [Function]

Returns `lock` name.

`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 op*) **&body** *body* [Macro]

Acquire rwlock for the dynamic scope of `body` for operation `op`, which is executed with the lock held by current thread, and `with-rwlock` returns the values of body.

Valid values of argument `op` are `:read` or `:write` (for reader and writer access accordingly).

4.4.8 Condition variables

Condition variables are used to wait for a particular condition becoming true (e.g new client connects to the server).

4.4.9 Condition variables dictionary

`cl_object mp_make_condition_variable` () [Function]

`mp:make-condition-variable` [Function]

Creates a condition variable.

`cl_object mp_condition_variable_wait` (*cl_object cv*, *cl_object lock*) [Function]

`mp:condition-variable-wait` *cv lock* [Function]

Release `lock` and suspend thread until condition `mp:condition-variable-signal` is called on `cv`. When thread resumes re-aquire `lock`.

`cl_object mp_condition_variable_timedwait` (*cl_object cv*, *cl_object lock*, *cl_object seconds*) [Function]

`mp:condition-variable-timedwait` *cv lock seconds* [Function]

`mp:condition-variable-wait` which timeouts after `seconds` seconds.

`cl_object mp_condition_variable_signal` (*cl_object cv*) [Function]

`mp:condition-variable-signal` *cv* [Function]

Signal `cv` (wakes up only one waiter). After signal, signaling thread keeps lock, waking thread goes on the queue waiting for the lock.

See `mp:condition-variable-wait`.

`cl_object mp_condition_variable_broadcast` (*cl_object cv*) [Function]

`mp:condition-variable-broadcast` *cv* [Function]

Signal `cv` (wakes up all waiters).

See `mp:condition-variable-wait`.

4.4.10 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.

4.4.11 Semaphores dictionary

`cl_object ecl_make_semaphore` (*cl_object name*, *cl_fixnum count*) [Function]

C/C++ equivalent of `mp:make-sempahore` without `key` arguments.

See `mp:make-sempahore`.

`mp:make-semaphore` **&key** *name count* [Function]

Creates a counting semaphore `name` with a resource count `count`.

`cl_object mp_semaphore_name` (*cl_object semaphore*) [Function]

`mp:semaphore-name` *semaphore* [Function]

Returns `semaphore` name.

`cl_object mp_semaphore_count` (*cl_object semaphore*) [Function]

`mp:semaphore-count` *semaphore* [Function]

Returns `semaphore` count of resources.

`cl_object mp_semaphore_wait_count` (*cl_object semaphore*) [Function]

`mp:semaphore-wait-count` *semaphore* [Function]

Returns number of threads waiting on `semaphore`.

`cl_object mp_wait_on_semaphore` (*cl_object semaphore*) [Function]

`mp:wait-on-semaphore` *semaphore* [Function]

Waits on semaphore until it can grab the resource (blocking). Returns resource count before semaphore was acquired.

`cl_object mp_try_get_semaphore` (*cl_object semaphore*) [Function]

`mp:try_get_semaphore` *semaphore* [Function]

Tries to get a semaphore (non-blocking). If there is no resource left returns `NIL`, otherwise returns resource count before semaphore was acquired.

`cl_object mp_signal_semaphore` (*cl_narg n, cl_object sem, ...);* [Function]

`mp:signal-semaphore` *semaphore* **&optional** (*count* **1**) [Function]

Releases `count` units of a resource on `semaphore`.

4.5 Signals and Interrupts

4.6 Memory Management

4.7 Meta-Object Protocol (MOP)

4.8 Gray Streams

4.9 Tree walker

4.10 Package locks

4.10.1 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.

4.10.2 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.

4.10.3 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*]]* ⇒ *package* [Macro]

Options are extended to include the following:

- `: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 "FOO" (:export "BAR") (:lock t))

;;; is equivalent to

(defpackage "FOO") (:export "BAR"))
(lock-package "FOO")
```

4.11 CDR Extensions

Indexes

Concept index

A

ANSI Dictionary ... 31

B

Bytecodes eager compilation ... 33

C

C/C++ code inlining ... 54
Command line processing ... 45
Common Lisp functions limits ... 34
Compiler declarations ... 31
Creating executables and libraries ... 37
`cstring` and `foreign string` differences ... 68

D

`disassemble` and `compile` on defined functions ... 34

E

Eager compilation implications ... 33
Environment implementation ... 27
External processes ... 47

F

Foreign aggregate types ... 60
Foreign function interface ... 49
Foreign functions and libraries ... 72
Foreign objects ... 63
Foreign primitive types ... 58
Foreign strings ... 68

H

Hash table serialization ... 35

N

Native FASL ... 39
Native threads ... 74

O

Object file internal layout ... 40
One type for everything: `cl_object` ... 29
Only in Common Lisp ... 31

P

Package locks ... 82
Parsing arguments in standalone executable ... 46
Portable FASL ... 38

R

Readers-writer locks ... 80

S

Shadowed bindings in LET, FLET, LABELS and lambda-list ... 33
Shared-exclusive locks ... 80
Static foreign function interface ... 54
Synchronized hash tables ... 35
System building ... 37

T

Thread-safe hash tables ... 35
Two kinds of FFI ... 50

U

Universal foreign function interface ... 58

W

Weak hash tables ... 35

Configure option index

--enable-shared [YES|no]] 39
--enable-small-cons [YES|no] 17
--enable-threads [yes|no|AUTO] 74
--enable-unicode [YES|no|32] 21
--with-dffi [system|included|AUTO|no] 50
--with-libffi-prefix=path 50

Feature index

D

DFFI 51
DLOPEN 39

E

ecl-read-write-lock 80
ECL-WEAK-HASH 35

F

FFI 49

L

LONG-FLOAT 58
LONG-LONG 58

P

PACKAGE-LOCKS 82

T

THREADS 74

U

UINT16-T 58
UINT32-T 58
UINT64-T 58

Example index

A

Accessing underlying cl_object structure 18

B

Building executable 41
Building native FASL 39
Building Portable FASL file 38
Building shared library 41
Building static library 40

C

c_string_to_object constructing Lisp objects in C 19
CFFI usage 53
cl_object checking the type with ecl_t_of 18
cl_safe_eval 27
Conversion between foreign string and cstring 69
cstring used to send and return a value 68

D

Defpackage :lock option 84
distinguishing between base and Unicode character 21

E

Eager compilation impact on macros 33
ecl_aref and ecl_aset accessing arrays 23
ecl_aref1 and ecl_aset1 accessing vectors 24
ecl_array_elttype different types of objects ... 23

ffi:find-foreign-library 74
ffi:get-slot-value manipulating a struct field 62
ffi:get-slot-value usage 62
ffi:load-foreign-library 73
ffi:null-char-p example 60
ffi:size-of-foreign-type 65
ffi:with-cast-pointer 67
ffi:with-foreign-object macro usage 64
foreign string used to send and return a value 68

H

Hash table extensions example 35

I

Initializing static/shared library in C/C++ 40

K

Keeping lambda definitions with si:*keep-definitions 34
Killing process 76

L

LS implementation 46

M

mp:process-run-funciton usage 77

P

Function index

(

(cl_object ... 79, 80

C

c_string_to_object ... 19
cl:defpackage ... 84
cl_env_ptr ... 32

E

ecl_aref ... 23, 24
ecl_array_elttype ... 23
ecl_aset ... 23, 24
ecl_char_cmp ... 21
ecl_char_compare ... 21
ecl_char_eq ... 21
ecl_char_equal ... 21
ecl_fixnum_geq ... 20
ecl_fixnum_greater ... 20
ecl_fixnum_leq ... 20
ecl_fixnum_lower ... 20
ecl_fixnum_minusp ... 20
ecl_fixnum_plusp ... 20
ecl_make_fixnum ... 20
ecl_make_semaphore ... 81
ecl_t_of ... 19
ecl_unfix ... 20
ECL_ADJUSTABLE_ARRAY_P ... 22
ECL_ANSI_STREAM_P ... 26
ECL_ANSI_STREAM_TYPE_P ... 26
ECL_ARRAY_HAS_FILL_POINTER_P ... 22
ECL_ARRAYP ... 19
ECL_ATOM ... 19
ECL_BASE_CHAR_CODE_P ... 19
ECL_BASE_CHAR_P ... 19
ECL_BASE_STRING_P ... 24
ECL_BIT_VECTOR_P ... 19
ECL_CHAR_CODE ... 21
ECL_CHARACTERP ... 19
ECL_CLASS_CPL ... 26
ECL_CLASS_INFERIORS ... 26
ECL_CLASS_NAME ... 26
ECL_CLASS_OF ... 26
ECL_CLASS_SLOTS ... 26
ECL_CLASS_SUPERIORS ... 26
ECL_CODE_CHAR ... 21
ECL_CODE_CHAR_P ... 19
ECL_CONSP ... 19
ECL_EXTENDED_STRING_P ... 24
ECL_FIXNUMP ... 19
ECL_IMMEDIATE ... 19
ECL_INSTANCEP ... 26
ECL_LISTP ... 19
ECL_NUMBER_TYPE_P ... 19
ECL_REAL_TYPE_P ... 19
ECL_SPEC_FLAG ... 26
ECL_SPEC_OBJECT ... 26
ECL_STRINGP ... 19
ECL_STRUCT_LENGTH ... 26
ECL_STRUCT_NAME ... 26
ECL_STRUCT_SLOT ... 26
ECL_STRUCT_SLOTS ... 26
ECL_STRUCT_TYPE ... 26
ECL_SYMBOLP ... 19
ECL_VECTORP ... 19
ext:chdir ... 49
ext:chmod ... 49
ext:command-args ... 45
ext:copy-file ... 49
ext:environ ... 49
ext:external-process-error-stream ... 47
ext:external-process-input ... 47
ext:external-process-output ... 47
ext:external-process-pid ... 47
ext:external-process-status ... 47
ext:external-process-wait ... 47
ext:file-kind ... 49
ext:getcwd ... 49
ext:getenv ... 49
ext:getpid ... 49
ext:getuid ... 49
ext:lock-package ... 83
ext:make-pipe ... 49
ext:package-locked-p ... 83
ext:process-command-args ... 45
ext:quit ... 49
ext:run-program ... 47
ext:setenv ... 49
ext:system ... 49
ext:terminate-process ... 47
ext:unlock-package ... 83
ext:with-unlocked-packages ... 84
ext:without-package-locks ... 83

F

ffi:allocate-foreign-object ... 63
ffi:allocate-foreign-string ... 71
ffi:c-inline ... 55
ffi:c-progn ... 56
ffi:clines ... 54
ffi:convert-from-cstring ... 69, 70
ffi:convert-from-foreign-string ... 71
ffi:convert-to-cstring ... 69
ffi:convert-to-foreign-string ... 71
ffi:def-array-pointer ... 62
ffi:def-constant ... 59
ffi:def-enum ... 60
ffi:def-foreign-type ... 59

ffi:def-foreign-var 67
ffi:def-function 72
ffi:def-struct 61
ffi:def-union 63
ffi:defcallback 57
ffi:defcbody 57
ffi:defentry 57
ffi:defla 58
ffi:deref-array 62
ffi:deref-pointer 65
ffi:ensure-char-character 65
ffi:ensure-char-integer 66
ffi:find-foreign-library 73
ffi:free-foreign-object 64
ffi:get-slot-pointer 62
ffi:get-slot-value 61
ffi:load-foreign-library 73
ffi:make-null-pointer 66
ffi:null-char-p 60
ffi:null-pointer-p 66
ffi:pointer-address 65
ffi:size-of-foreign-type 64
ffi:with-cast-pointer 67
ffi:with-cstring 70
ffi:with-cstrings 70
ffi:with-foreign-object 64
ffi:with-foreign-string 71
ffi:with-foreign-strings 72
fixint 21
fixnint 21

H

hash-table-content 35
hash-table-fill 35
hash-table-synchronized-p 35
hash-table-weakness 35

M

mp:all-processes 75
mp:block-signals 77
mp:condition-variable-broadcast 81
mp:condition-variable-signal 81
mp:condition-variable-timedwait 81
mp:condition-variable-wait 81
mp:current-process 77
mp:exit_process 75
mp:get-lock 79
mp:get-rwlock-read 80
mp:get-rwlock-write 80
mp:giveup_lock 79
mp:giveup_rwlock_read 80
mp:giveup_rwlock_write 80
mp:holding-lock-p 79
mp:interrupt_process 75
mp:lock-count 79
mp:lock_name 79
mp:lock_owner 79
mp:make-condition-variable 81
mp:make-lock 79
mp:make-process 75
mp:make-semaphore 81
mp:make_rwlock 80
mp:process-active-p 75
mp:process-enable 75
mp:process-join 76
mp:process-kill 76
mp:process-name 77
mp:process-preset 77
mp:process-resume 77
mp:process-suspend 76
mp:process-yield 76
mp:process_run_function 77
mp:recursive-lock-p 79
mp:restor-signals 77
mp:rwlock_name 80
mp:semaphore-count 82
mp:semaphore-name 82
mp:semaphore-wait-count 82
mp:signal-semaphore 82
mp:try_get_semaphore 82
mp:wait-on-semaphore 82
mp:with-interrupts 78
mp:with-lock 79
mp:with-rwlock 80
mp:without-interrupts 78
mp_all_processes 75
mp_block_signals 77
mp_condition_variable_broadcast 81
mp_condition_variable_signal 81
mp_condition_variable_timedwait 81
mp_condition_variable_wait 81
mp_current_process 77
mp_get_lock_nowait 79
mp_get_lock_wait 79
mp_get_rwlock_read_nowait 80
mp_get_rwlock_read_wait 80
mp_get_rwlock_write_nowait 80
mp_get_rwlock_write_wait 80
mp_giveup_lock 79
mp_giveup_rwlock_read 80
mp_giveup_rwlock_write 80
mp_holding_lock_p 79
mp_interrupt_process 75
mp_lock_count 79
mp_lock_name 79
mp_lock_owner 79
mp_make_condition_variable 81
mp_make_process 75
mp_process_enable 75
mp_process_join 76
mp_process_kill 76
mp_process_name 77
mp_process_preset 77
mp_process_resume 77

mp_process_run_function 77
mp_process_suspend 76
mp_process_yield 76
mp_recursive_lock_p 79
mp_restore_signals 77
mp_rwlock_name 80
mp_semaphore_count 82
mp_semaphore_name 82
mp_semaphore_wait_count 82
mp_signal_semaphore 82
mp_try_get_semaphore 82
mp_wait_on_semaphore 82

S

si_make_lambda 27
si_safe_eval 26
string_to_object 19

Variable index

*

keep-definitions 34

+

+null-cstring-pointer+ 66

:

:LOADRC 46
:NOLOADRC 46
:STOP 46

A

args 45

E

ECL_CHAR_CODE_LIMIT 21
environ 48
error 48
escape-arguments 49
ext:*help-message* 45
ext:*lisp-init-file-list* 45
ext:+default-command-arg-rules+ 45
external-format 49

I

if-error-exists 48
if-input-does-not-exist 48
if-output-exists 48
input 48

M

MOST_NEGATIVE_FIXNUM 20
MOST_POSITIVE_FIXNUM 20

N

nargs 46

O

option-name 45
output 48

R

rules 45

T

template 46

W

wait 48

Type index

C

cl_elttype ... 23
cl_fixnum ... 20
cl_index ... 20
cl_lispunion ... 17
cl_object ... 18
cl_type ... 18

E

ecl_array ... 22
ecl_base_string ... 24
ecl_character ... 21
ecl_file_ops ... 25
ecl_stream ... 25
ecl_string ... 24
ecl_vector ... 22

Common Lisp symbols

:

:* ... 58
:byte ... 58
:char ... 58
:cstring ... 58
:double ... 58
:float ... 58
:int ... 58
:int16_t ... 58
:int32_t ... 58
:int64_t ... 58
:long ... 58
:pointer-void ... 58
:short ... 58
:uint16_t ... 58
:uint32_t ... 58
:uint64_t ... 58
:unsigned-byte ... 58
:unsigned-char ... 58
:unsigned-int ... 58
:unsigned-long ... 58
:unsigned-short ... 58
:void ... 58

A

allow-with-interrupts ... 78

C

call-arguments-limit ... 34

D

debug ... 31

E

ext:lock-package ... 83
ext:package-locked-p ... 83
ext:run-program ... 47
ext:unlock-package ... 83
ext:with-unlocked-packages ... 84
ext:without-package-locks ... 83

F

ffi:+null-cstring-pointer+ ... 66
ffi:allocate-foreign-object ... 63
ffi:allocate-foreign-string ... 71
ffi:c-inline ... 55
ffi:c-progn ... 56
ffi:clines ... 54
ffi:convert-from-cstring ... 69
ffi:convert-from-foreign-string ... 71
ffi:convert-to-cstring ... 69
ffi:convert-to-foreign-string ... 71
ffi:def-array-pointer ... 62
ffi:def-constant ... 59
ffi:def-enum ... 60
ffi:def-foreign-type ... 59
ffi:def-foreign-var ... 67
ffi:def-function ... 72
ffi:def-union ... 63
ffi:defcallback ... 57
ffi:defcbody ... 57
ffi:defentry ... 57
ffi:defla ... 58
ffi:deref-array ... 62
ffi:deref-pointer ... 65
ffi:ensure-char-character ... 65
ffi:ensure-char-integer ... 66
ffi:find-foreign-library ... 73
ffi:free-cstring ... 70
ffi:free-foreign-object ... 64
ffi:get-slot-pointer ... 62
ffi:get-slot-value ... 61
ffi:load-foreign-library ... 73
ffi:make-null-pointer ... 66
ffi:null-char-p ... 60
ffi:null-pointer-p ... 66
ffi:pointer-address ... 65
ffi:size-of-foreign-type ... 64
ffi:with-cast-pointer ... 67
ffi:with-cstring ... 70
ffi:with-cstrings ... 70
ffi:with-foreign-object ... 64
ffi:with-foreign-string ... 71
ffi:with-foreign-strings ... 72

H

hash-table-content 35
hash-table-fill 35
hash-table-synchronized-p 35
hash-table-weakness 35

L

lambda-list-keywords 34
lambda-parameters-limit 34

M

MOST-NEGATIVE-FIXNUM 20
MOST-POSITIVE-FIXNUM 20
mp:all-processes 75
mp:block-signals 77
mp:condition-variable-broadcast 81
mp:condition-variable-signal 81
mp:condition-variable-timedwait 81
mp:condition-variable-wait 81
mp:current_process 77
mp:exit-process 75
mp:get-lock 79
mp:get-rwlock-read 80
mp:get-rwlock-write 80
mp:giveup-lock 79
mp:giveup-rwlock-read 80
mp:giveup-rwlock-write 80
mp:holding-lock-p 79
mp:interrupt-process 75
mp:lock-count 79
mp:lock-name 79
mp:lock-owner 79
mp:make-condition-variable 81
mp:make-lock 79
mp:make-process 75
mp:make_rwlock 80
mp:make_semaphore 81
mp:process-active-p 75
mp:process-enable 75
mp:process-kill 76
mp:process-name 77
mp:process-preset 77
mp:process-resume 77
mp:process-run-function 77
mp:process-suspend 76
mp:process_join 76
mp:process_yield 76
mp:recursive-lock-p 79
mp:restore-signals 77
mp:rwlock-name 80
mp:semaphore-count 82
mp:semaphore-name 82
mp:semaphore-wait-count 82
mp:signal-semaphore 82
mp:try-get-semaphore 82
mp:wait-on-semaphore 82
mp:with-interrupts 78
mp:with-lock 79
mp:with-rwlock 80
mp:without-interrupts 78
mp_lock_owner 79
mp_signal_semaphore 82
multiple-values-limit 34

O

optimize 31

S

safety 31
si:*keep-definitions* 34
si::make-lambda 33
space 31
speed 31

W

with-local-interrupts 78
with-restored-interrupts 78

C/C++ index

C

c_string_to_object 19
CHAR_CODE 21
CHAR_CODE_LIMIT 21
cl_elttype 23
cl_env_ptr 27
cl_env_struct 27
cl_eval 26
cl_fixnum 20, 21
cl_index 20, 21
cl_lispunion 16
cl_object 18
cl_safe_eval 26
CODE_CHAR 21
CODE_CHAR_P 19

E

ecl_aref 23
ecl_aref1 24
ecl_array 22
ecl_array_elttype 23
ecl_aset 23

ecl_aset1 24
ecl_base_char 21
ecl_base_char_code 21
ecl_base_string 24
ecl_char_cmp 21
ecl_char_code 21
ecl_char_compare 21
ecl_char_eq 21
ecl_char_equal 21
ecl_character 21
ecl_file_pos 25
ecl_fixnum 20
ecl_fixnum_geq 20
ecl_fixnum_greater 20
ecl_fixnum_leq 20
ecl_fixnum_lower 20
ecl_fixnum_minusp 20
ecl_fixnum_plusp 20
ecl_make_fixnum 20
ecl_make_lock 79
ecl_make_rwlock 80
ecl_make_semaphore 81
ecl_process_env 32
ecl_stream 25
ecl_string 24
ecl_t_of 19
ecl_vector 22
ECL_ADJUSTABLE_ARRAY_P 22
ECL_ANSI_STREAM_P 26
ECL_ANSI_STREAM_TYPE_P 26
ECL_ARRAY_HAS_FILL_POINTER_P 22
ECL_ARRAYP 19
ECL_ATOM 19
ECL_BASE_CHAR_CODE_P 19
ECL_BASE_CHAR_P 19
ECL_BASE_STRING_P 24
ECL_BIT_VECTOR_P 19
ECL_CHAR_CODE 21
ECL_CHAR_CODE_LIMIT 21
ECL_CHARACTERP 19
ECL_CLASS_CPL 26
ECL_CLASS_INFERIORS 26
ECL_CLASS_NAME 26
ECL_CLASS_OF 26
ECL_CLASS_SLOTS 26
ECL_CLASS_SUPERIORS 26
ECL_CODE_CHAR 21
ECL_CONSP 19
ECL_EXTENDED_STRING_P 24
ECL_FIXNUMP 19
ECL_IMMEDIATE 19
ECL_INSTANCEP 26
ECL_LISTP 19
ECL_NUMBER_TYPE_P 19
ECL_REAL_TYPE_P 19
ECL_SPEC_FLAG 26
ECL_SPEC_OBJECT 26
ECL_STRINGP 19
ECL_STRUCT_LENGTH 26
ECL_STRUCT_NAME 26
ECL_STRUCT_SLOT 26
ECL_STRUCT_SLOTS 26
ECL_STRUCT_TYPE 26
ECL_SYMBOLP 19
ECL_VECTORP 19

F

fix 20

M

MAKE_FIXNUM 20
MOST_NEGATIVE_FIXNUM 20
MOST_POSITIVE_FIXNUM 20
mp_all_processes 75
mp_block_signals 77
mp_condition_variable-broadcast 81
mp_condition_variable_signal 81
mp_condition_variable_timedwait 81
mp_condition_variable_wait 81
mp_current_process 77
mp_exit_process 75
mp_get_lock_nowait 79
mp_get_lock_wait 79
mp_get_rwlock_read_nowait 80
mp_get_rwlock_read_wait 80
mp_get_rwlock_write_nowait 80
mp_get_rwlock_write_wait 80
mp_giveup_lock 79
mp_giveup_rwlock_read 80
mp_giveup_rwlock_write 80
mp_holding_lock_p 79
mp_interrupt_process 75
mp_lock_count 79
mp_lock_name 79
mp_make_condition_variable 81
mp_make_process 75
mp_make_semaphore 81
mp_process-join 76
mp_process_active_p 75
mp_process_enable 75
mp_process_kill 76
mp_process_name 77
mp_process_preset 77
mp_process_resume 77
mp_process_run_function 77
mp_process_suspend 76
mp_process_yield 76
mp_recursive_lock_p 79
mp_restore_signals 77
mp_rwlock_name 80
mp_semaphore_count 82
mp_semaphore_name 82
mp_semaphore_wait_count 82

mp_try_get_semaphore 82
mp_wait_on_semaphore 82

S

si_make_lambda 27
si_safe_eval 26
string_to_object 19

T

t_array 18
t_barrier 18
t_base_string 18
t_bclosure 18
t_bignum 18
t_bitvector 18
t_bytecodes 18
t_cclosure 18
t_cfun 18
t_cfunfixed 18
t_character 18
t_codeblock 18
t_complex 18
t_condition_variable 18
t_contiguous -- contiguous block 18
t_end 18
t_fixnum 18
t_foreign 18
t_frame 18
t_hashtable 18
t_instance 18
t_list 18
t_lock 18
t_longfloat 18
t_mailbox 18
t_other 18
t_package 18
t_pathname 18
t_process 18
t_random 18
t_ratio 18
t_readtable 18
t_rwlock 18
t_semaphore 18
t_singlefloat 18
t_sse_pack 18
t_start 18
t_stream 18
t_string 18
t_structure = t_instance 18
t_symbol 18
t_vector 18
t_weak_pointer 18

Bibliography

ANSI ANSI Common-Lisp Specification, 1986.

AMOP Gregor Kickzales 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 Giusseppe 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.