ECL

Embeddable Common-Lisp

Juan José García-Ripoll
http://ecls.sourceforge.net
worm@arrakis.es

Embeddable Common Lisp

ECL

Embeddable Common-Lisp

Juan José García-Ripoll

Max-Planck-Institut for Quantum Optics

Munich, Germany

Personal backround:

- Self-educated in varios programming languages

  BASIC, Assembler, Logo, Pascal, C

Personal backround:

- Self-educated in varios programming languages

  BASIC, Assembler, Logo, Pascal, C

- Scientific computing for job

  Functional programing, interpreted languages: MATLAB, Mathematica, Yorick. . .

Personal backround:

- Self-educated in varios programming languages

  BASIC, Assembler, Logo, Pascal, C

- Scientific computing for job

  Functional programing, interpreted languages: MATLAB, Mathematica, Yorick…

- Experience on free software

  GNU autoconf, XFree86 & Doom on OS/2

Personal backround:

- Self-educated in varios programming languages

  BASIC, Assembler, Logo, Pascal, C

- Scientific computing for job

  Functional programing, interpreted languages: MATLAB, Mathematica, Yorick. . .

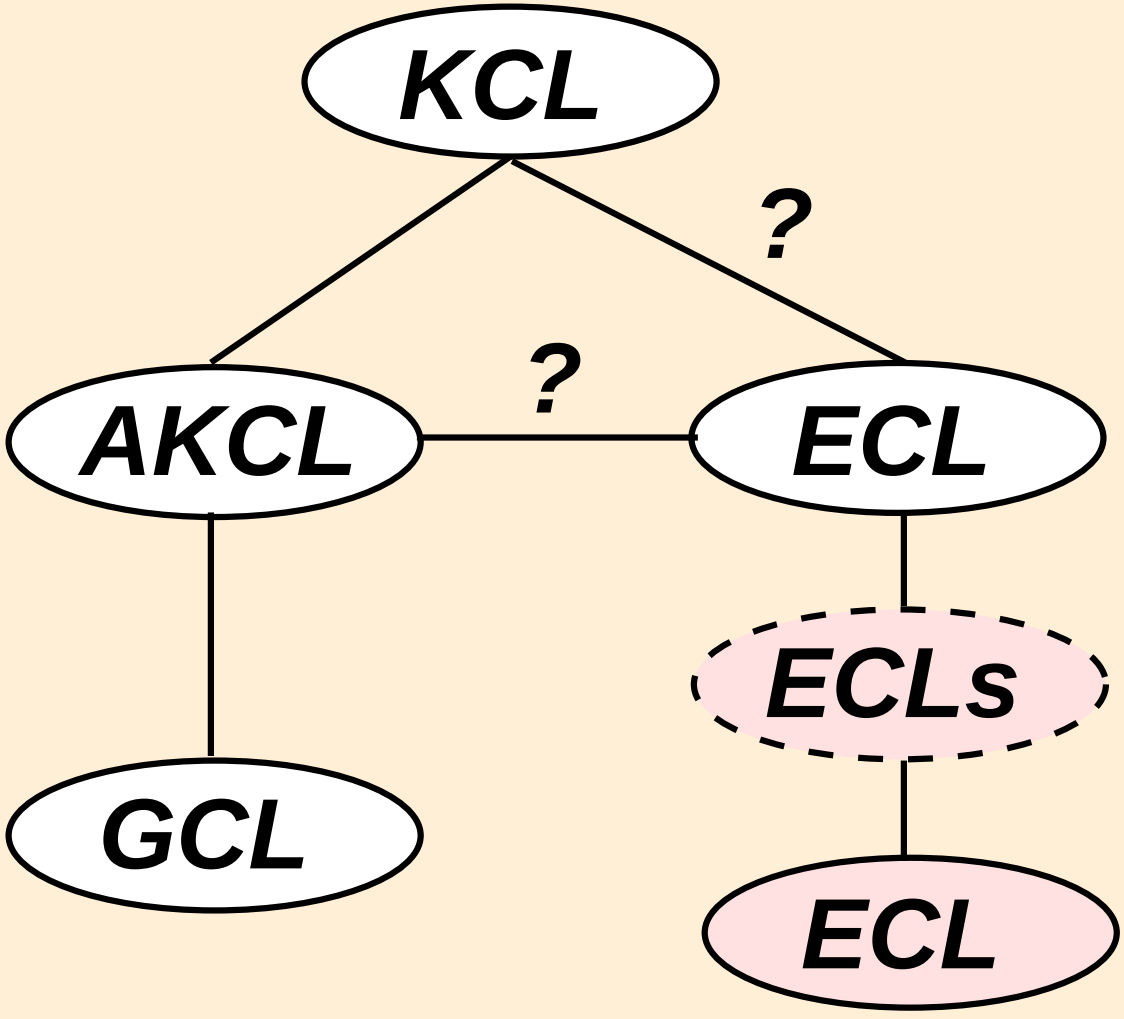
- Experience on free software

  GNU autoconf, XFree86 & Doom on OS/2

- Some spare time. . .

Where ECL comes from

Family tree:



Some names:

- Kyoto Common Lisp

  Taiichi Yuasa
  Masami Hagiya

- Austin KCL, GNU CL

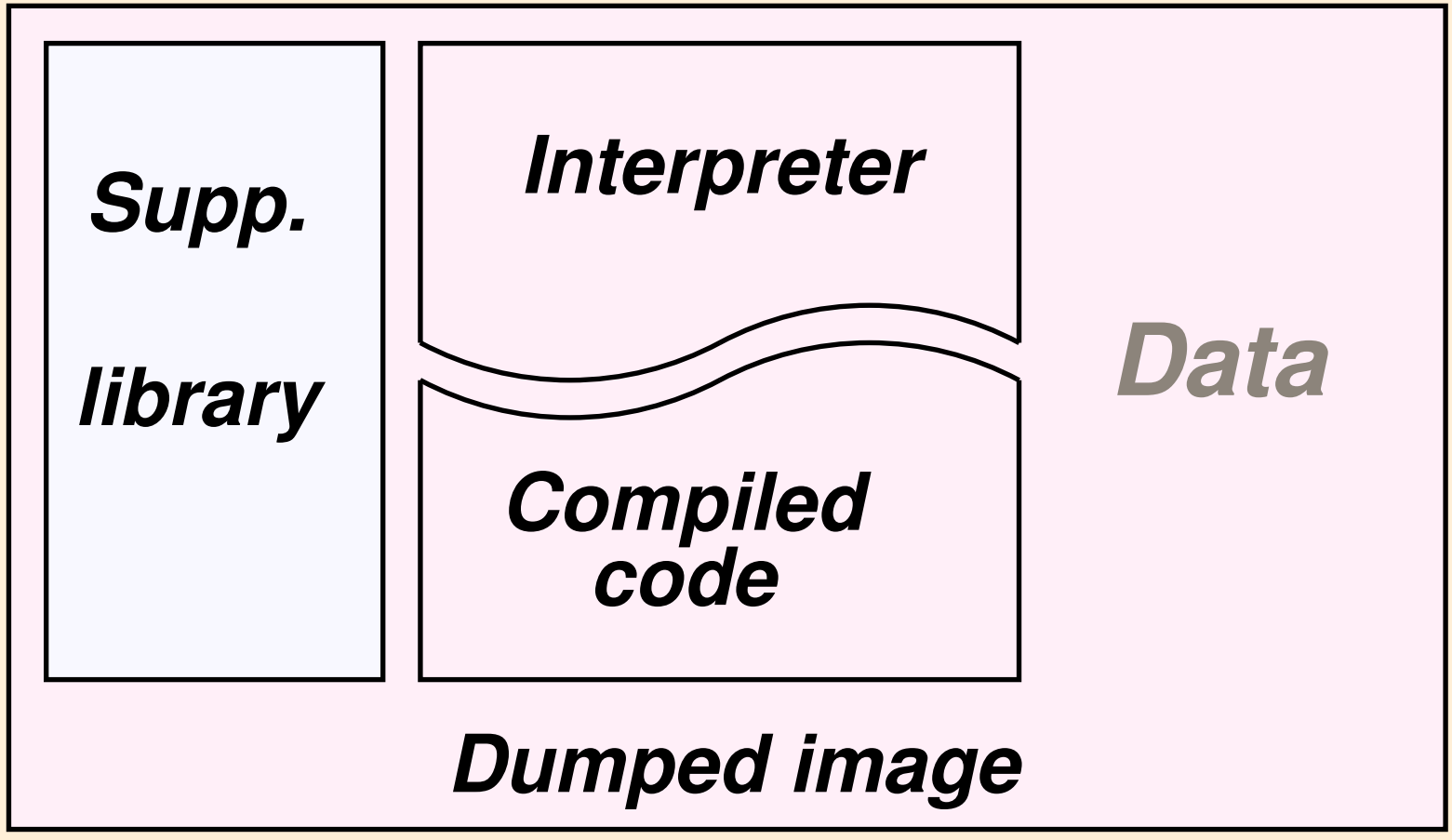
  William F. Schelter

- EcoCL (ECL)

  Giuseppe Attardi

The old ECL model

The architecture:



The Lego pieces:

- Supporting C functions

  To create & manipulate Lisp objects.

- An interpreter

  Code as lists, walked at evaluation time

- A compiler to C

  Object files may be loaded.

- A memory dumper

  Code and data may be dumped.
  The resulting image may be executed.

The interpreter:

- Code is stored as lists.
- Lists are walked at evaluation time.
- Environments are nested lists of variables.

The interpreter:

- Code is stored as lists.
- Lists are walked at evaluation time.
- Environments are nested lists of variables.

Pros & Cons:

- Unsafe.
- Inefficient in space & time.
- Nonstandard macroexpansion.

The interpreter:

- Code is stored as lists.
- Lists are walked at evaluation time.
- Environments are nested lists of variables.

Pros & Cons:

- – Unsafe.
- – Inefficient in space & time.
- – Nonstandard macroexpansion.
- \+ Macro dependencies are resolved.
- \+ The C stack is the interpreter stack.

The C compiler:

Given that the interpreter is written in C, lisp code may be translated to a sequence of calls to the supporting C code.

```
static object LI3(object V24)
{        VMB3 VMS3 VMV3
        base[1]= (V24);
        vs_top=(vs_base=base+1)+1;
        Limagpart();
        ...
        vs_top=sup;
        {object V25 = vs_base[0];
        VMR3(V25)}
}
```

However, we would like these functions to be easily used by the C programmer.

Creating Lisp images:

Basically, the content of data and code segments is dumped to a file, in executable format.

Creating Lisp images:

Basically, the content of data and code segments is dumped to a file, in executable format.

+ Very fast startup sequences

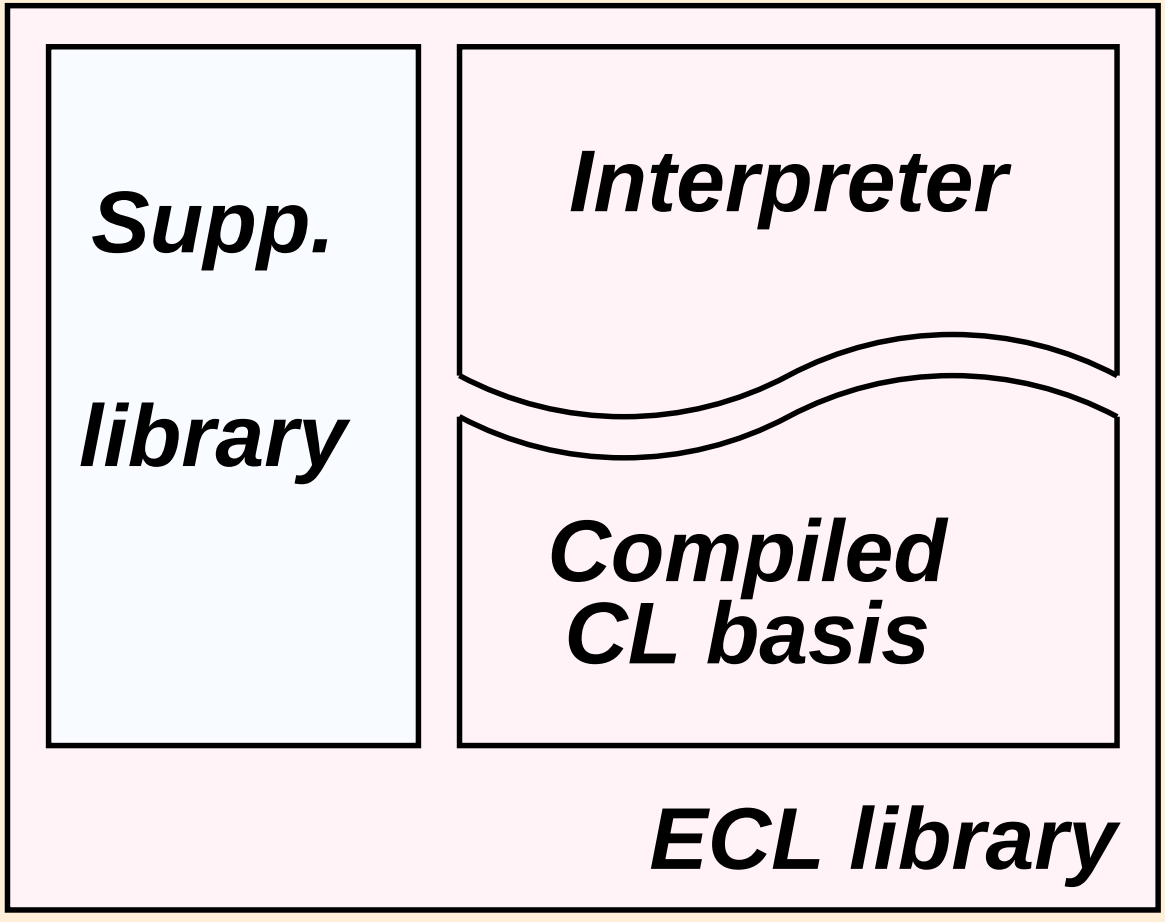
Creating Lisp images:

Basically, the content of data and code segments is dumped to a file, in executable format.

- \+ Very fast startup sequences
- – Highly nonportable techniques
- – Executable formats become obsolete
- – No sharing of code (DLLs)

ECL now

The ECL architecture



ECL library
+
Compiled Lisp code
+
User C/C++ code
=
Program

Goal 1: Portability

We achieve portability using the ANSI C language, and the ANSI C and POSIX libraries:

Linux, Mac OSX, FreeBSD,
NetBSD & OpenBSD, i386 & PPC

Goal 1: Portability

We achieve portability using the ANSI C language, and the ANSI C and POSIX libraries:

Linux, Mac OSX, FreeBSD,
NetBSD & OpenBSD, i386 & PPC

We only require these non-standard features:

- We need pointer $\leftrightarrow$ integer conversions.
- Functions must be called with any # args.

Goal 1: Portability

We achieve portability using the ANSI C language, and the ANSI C and POSIX libraries:

Linux, Mac OSX, FreeBSD,
NetBSD & OpenBSD, i386 & PPC

We only require these non-standard features:

- We need pointer $\leftrightarrow$ integer conversions.
- Functions must be called with any # args.
- A conservative garbage collector.

Goal 2: ANSI compliance

ECL is rather close to the ANSI CL specification

Goal 2: ANSI compliance

ECL is rather close to the ANSI CL specification

- Symbol macros
- Destructuring

Goal 2: ANSI compliance

ECL is rather close to the ANSI CL specification

- Symbol macros
- Destructuring
- Symbolics LOOP

Goal 2: ANSI compliance

ECL is rather close to the ANSI CL specification

+ Symbol macros
+ Destructuring
+ Symbolics LOOP

but still things to be revised

– FORMAT
– Pretty printer
– The type hierarchy

Goal 3: Self bootstrapping

The C library, the interpreter and the lisp sources for the rest of the library should form a standalone ANSI CL implementation, which can be used to compile the lisp sources.

Goal 3: Self bootstrapping

The C library, the interpreter and the lisp sources for the rest of the library should form a standalone ANSI CL implementation, which can be used to compile the lisp sources.

- Everybody has an ANSI C compiler.
- Non-experts may play with the code.
- Porting issues are reduced.
- Easy error recovery.

Goal 4: Shipped as a library

ECL is shipped with a set of libraries that can be embedded in other C/C++ programs.

Goal 4: Shipped as a library

ECL is shipped with a set of libraries that can be embedded in other C/C++ programs.

- CL as extension language.

Goal 4: Shipped as a library

ECL is shipped with a set of libraries that can be embedded in other C/C++ programs.

- CL as extension language.
- Deliver standalone applications.

Goal 4: Shipped as a library

ECL is shipped with a set of libraries that can be embedded in other C/C++ programs.

- CL as extension language.
- Deliver standalone applications.
- Helps spread Common-Lisp!

Goal 4: Shipped as a library

ECL is shipped with a set of libraries that can be embedded in other C/C++ programs.

- CL as extension language.
- Deliver standalone applications.
- Helps spread Common-Lisp!

The bottom line:

Keep ECL small!

Embeddable Common Lisp

Support libray

Memory management:

We use the Boehm-Weiser garbage collector:

Memory management:

We use the Boehm-Weiser garbage collector:

- It is a conservative, mark & sweep GC

Memory management:

We use the Boehm-Weiser garbage collector:

- It is a conservative, mark & sweep GC
- It can handle large amounts of memory

Memory management:

We use the Boehm-Weiser garbage collector:

- It is a conservative, mark & sweep GC
- It can handle large amounts of memory
- Good support for C/C++ programmers
- Supports many architectures

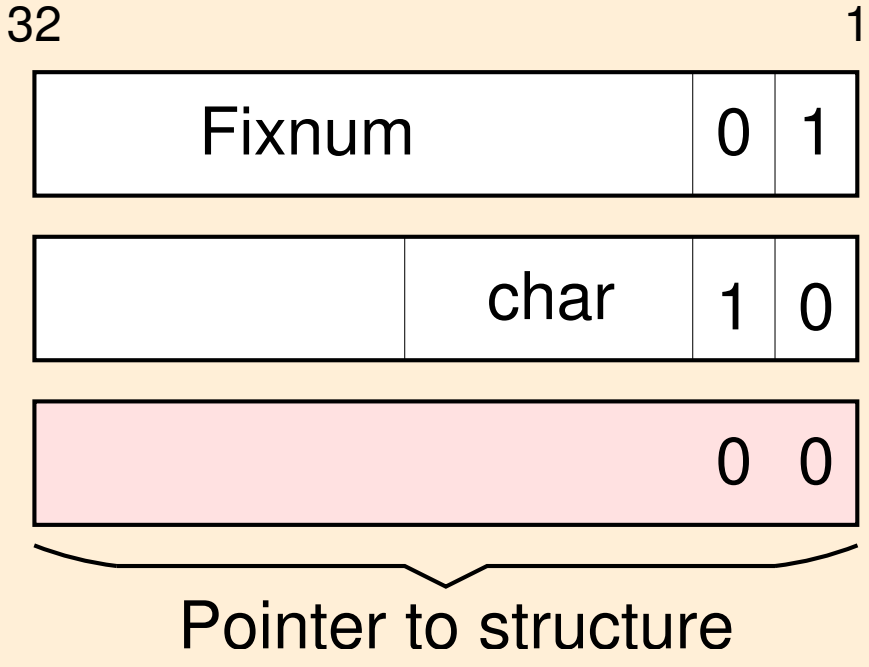
Memory management:

We use the Boehm-Weiser garbage collector:

- It is a conservative, mark & sweep GC
- It can handle large amounts of memory
- Good support for C/C++ programmers
- Supports many architectures

But any other garbage collector may be easily plugged in!!!.

Objects representation:



- Bignums provided by GNU MP v4.0
- 0-terminated strings

C core library:

Lots of functions are provided to create and manipulate lisp objects from C/C++ code.

```
cl_object form =
  c_string_to_object("(print 1)");
cl_object output = eval(form, NULL, Cnil);
cl_terpri(Cnil);
cl_make_constant_string("Some string");
```

C core library:

- The library implements all CL objects
- Can simulate Lisp control structures

  catch, throw, unwind protect

C core library:

- \+ The library implements all CL objects
- \+ Can simulate Lisp control structures

  catch, throw, unwind protect
- – An easier interface is being worked on

  Hide internal structure of objects

  Hide functions, prefix others (cl_*)
- – Some things can only be done in CL (CLOS)
- – User defined datatypes.

The interpreter

Bytecodes compiler:

```
(defun f (x)
  (print
    (if (< x 0)
      "negative"
      "positive")))
```

```
Name:            f
Required:        X
Documentation:   NIL
Declarations:    NIL
   0    PUSHVS   <
   2    PUSH     '0
   4    CALLG    2,X
   6    JNIL     9
   7    "negative"
   8    JMP      10
   9    "positive"
  10    PUSH     VALUES(0)
  11    CALLG    1,PRINT
  13    HALT
```

Bytecodes interpreter:

- About 26 instructions dealing with variables

  SETQ, SETQS, PBIND, PBINDS…
- About 14 instructions for code flow

  JMP, JEQ, CALLG, FCALL…
- Rest (∼10) simulate high level constructs

  BLOCK, TAGBODY, DO, DOLIST…
- Code is "stack" oriented, the stack being shared with the rest of the library.

Pros & Cons:

- \+ Code is processed once.
- \+ Syntax errors are detected early.
- \+ Code executes faster.

Pros & Cons:

- \+ Code is processed once.
- \+ Syntax errors are detected early.
- \+ Code executes faster.
- \+ Straightforward (< 4kloc).
- \+ Much can still be optimized.

Pros & Cons:

- \+ Code is processed once.
- \+ Syntax errors are detected early.
- \+ Code executes faster.
- \+ Straightforward (< 4kloc).
- \+ Much can still be optimized.
- – No stepping debugger yet.
- – No development environment.
- – Lexical binding still conses.

Compiled Lisp code

The look of translated Lisp code:

```
cl_object
clLenough_namestring(int narg, cl_object path, ...)
{
   cl_object defaults;
   cl_va_list args;
   cl_va_start(args, path, narg, 1);

   if (narg < 1) FEtoo_feew_arguments(narg);
   ...
   if (narg > 1) defaults = cl_va_arg(args);
   ...
   NValues = 1; return newpath;
}
```

Entry point:

The function may receive any # of arguments, but only 64 using C calling conventions:

```
cl_object
clLenough_namestring(int narg, cl_object path, ...)
{
   cl_object defaults;
   cl_va_list args;
   cl_va_start(args, path, narg, 1);
```

&Optional and &key arguments and anything above #64 is retrieved using the `cl_va_arg()` function.

Exit point:

A function may return one value directly

```
NValues = 1; return newpath;
```

or up to 64 on the "values array"

```
NValues = 2;
VALUES(1) = MAKE_FIXNUM(2);
return MAKE_FIXNUM(1);
```

The function always outputs the first value, so that functions may be transparently called from C.

The object files:

A file of Lisp code is translated into a file of C code with the following sections:

- A textual representation of all the constants
- An array which holds the constants
- The code for all local and exportable functions
- An entry function which sets everything up

In enviroments which support the `dlopen()` function, this code may be turned into a DLL and loaded at run-time.

To be done:

- C functions with a fixed number of arguments:

  ```
  cl_object cl_fboundp(cl_object only_arg)
  ```

  instead of

  ```
  cl_object clLfboundp(int n, cl_object only_arg)
  ```

- Find other ways for handling multiple values.
- Write a better FFI.

To be done:

- C functions with a fixed number of arguments:

  ```
  cl_object cl_fboundp(cl_object only_arg)
  ```

  instead of

  ```
  cl_object clLfboundp(int n, cl_object only_arg)
  ```

- Find other ways for handling multiple values.
- Write a better FFI.

Write code to unload a DLL

Work in progress

Calling conventions

Currently, a compiled function may have only 64 required arguments. Only optional, keyword and &rest arguments are allowed to be on the interpreter stack.

- Minor changes in the C translator
- More slots per structure & class
- CLX may be ported!

Threads

EcoCL had a userland implementation of threads.

+ This implementation may be rescued
+ The code may be recycled for POSIX threads
- With POSIX threads special variables become more complicated to handle.
- Some code in ECL is not reentrant

Safe evaluation

ECL is being considered on MUD and other gaming projects, where code is exchanged between computers and should be executed in a safe environment.

- Protect the CL package better (makunbound,. . .)
- Selectively disable access to filesystem
- Provide a means to restart ECL
- Integrate better conditions and restarts

Usability:

S.O.S.