# ECL
# =
# *(not only)* Embeddable Common Lisp

Juan José García Ripoll

`http://ecls.sourceforge.net`

# Outline

- Introduction
- ECL's family tree and history
- ECL's current philosophy & design
- Salient features
- Future trends
- Outlook & questions

# About the maintainer

- Self-educated in different languages

  Lisp/Scheme, C, C++, ML...

- Work heavily focused on numerical analysis

  MATLAB, Yorick, ...

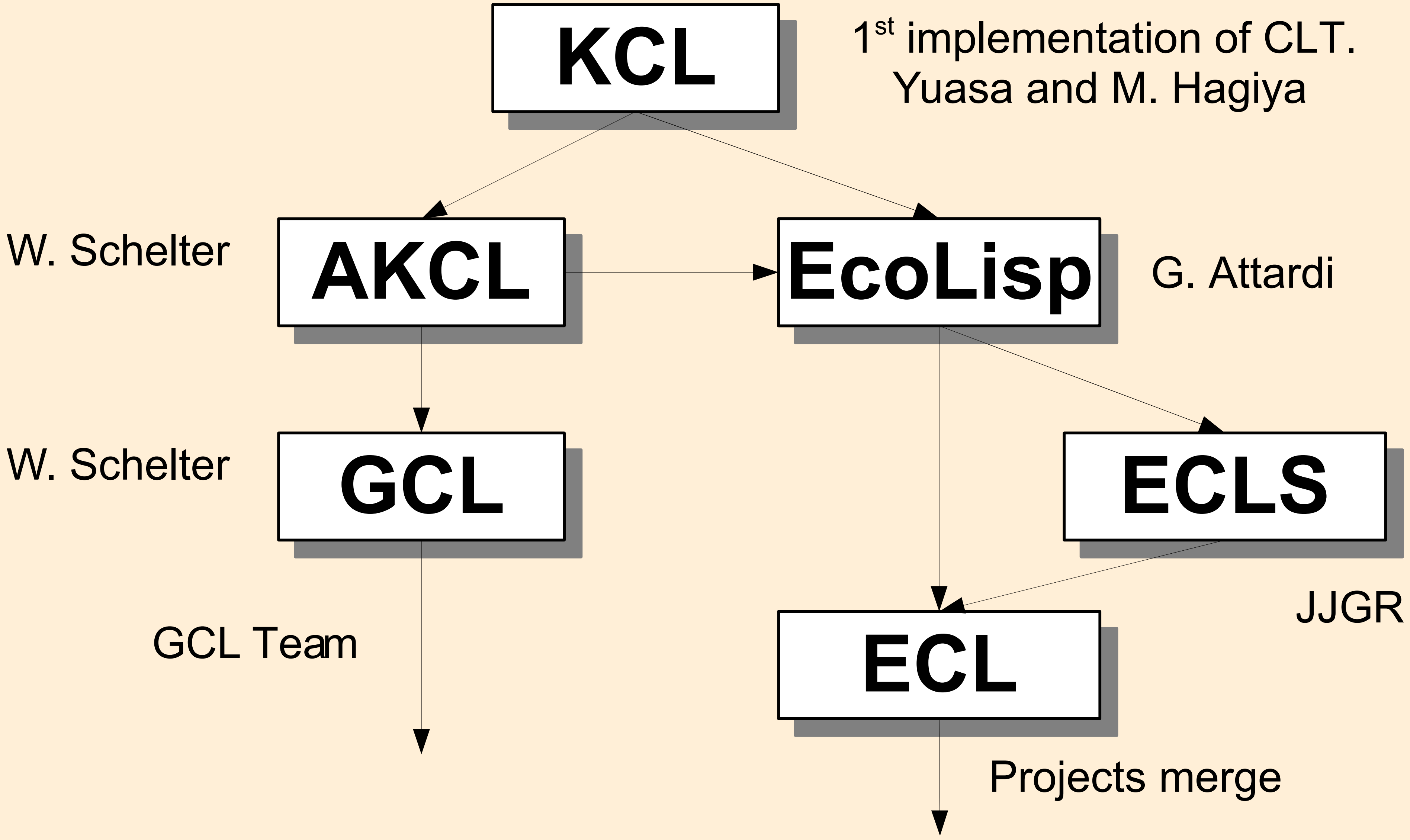
- Came to lisp searching for interactive environments that could evolve into numerical programming ones

  Scheme, ML, CMUCL, GCL, EcoLisp, ...

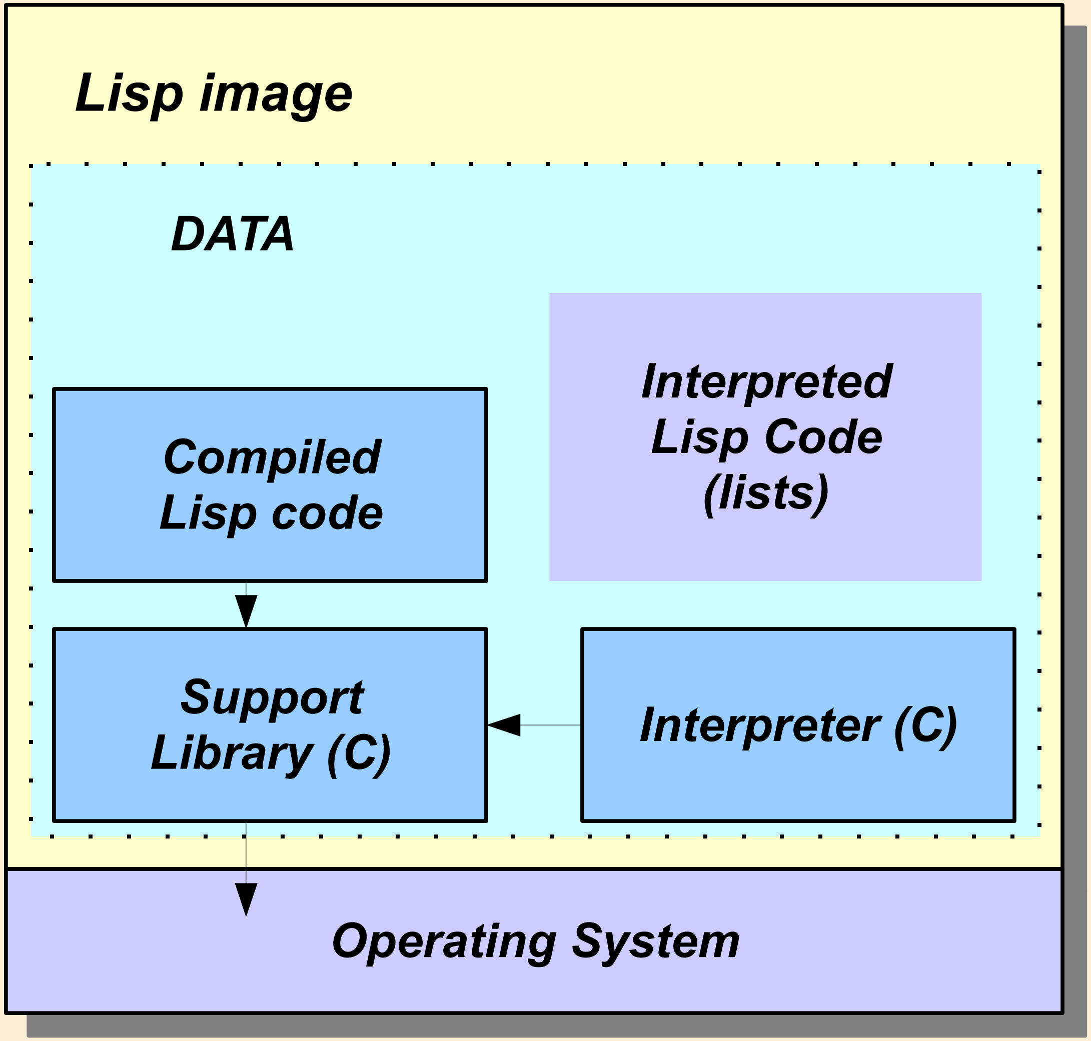
- Some experience on free projects

  ECL, OS/2 Gnu ports, Doom port, ...

# ECL Family tree



# Traditional *CL design



- Big chunk of memory contains everything
- A core is written in C manually.
- Lisp code compiled to C by a compiler written in lisp.
- Binaries loaded as data
- Whole image can be dumped and restored.

# Traditional *CL design

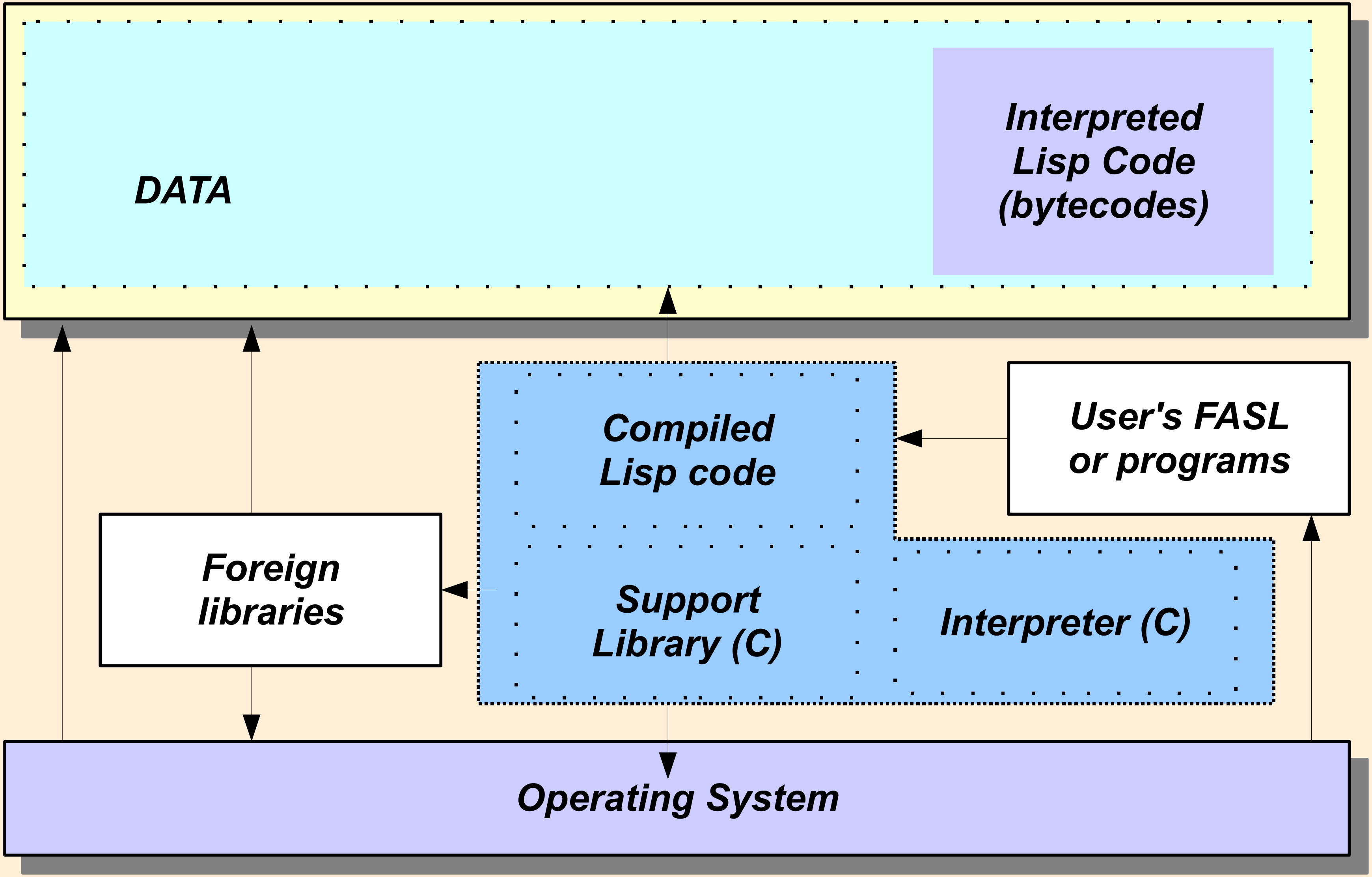
**Pros:**

- ✓ Portable (C) backend.
- ✓ Fast loading of images.
- ✓ Full control of memory & any GC strategy.
- ✓ It follows “tradition”

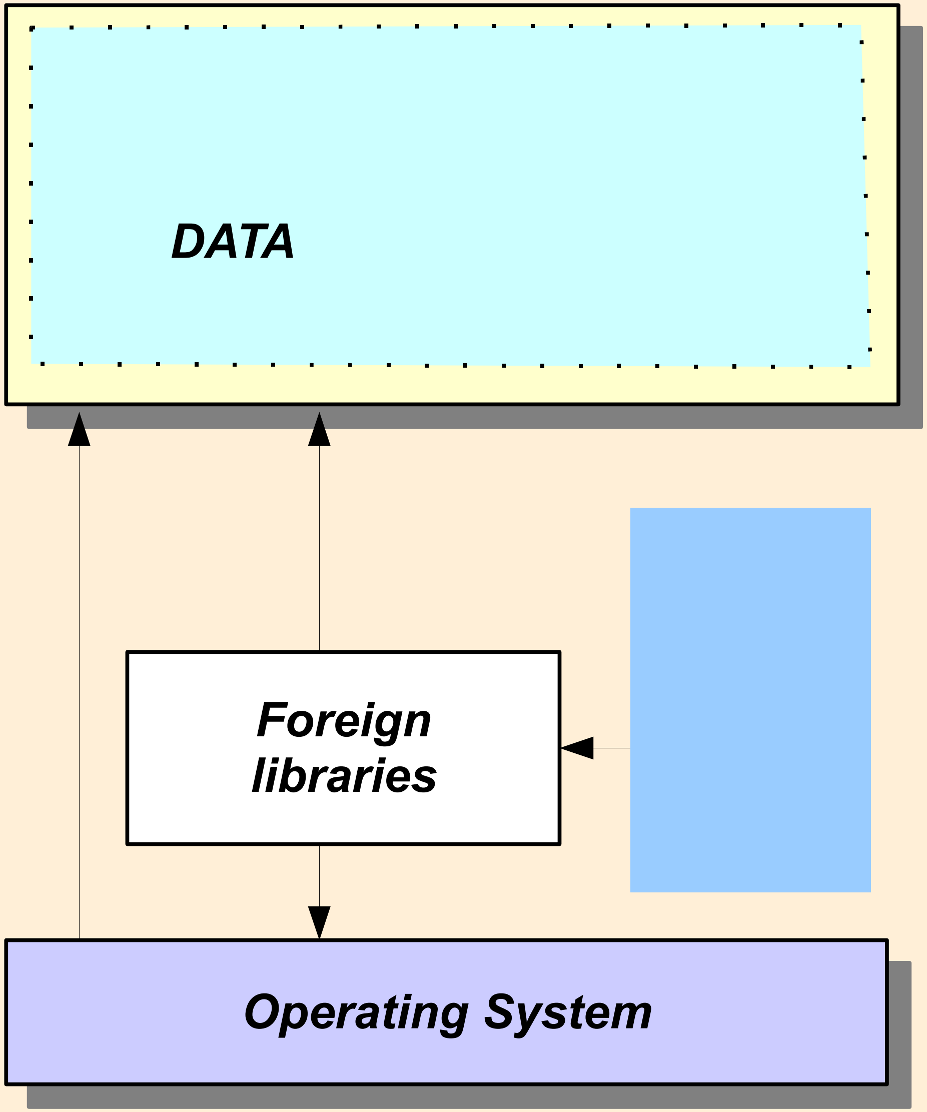
**Cons:**

- ✗ What is *portable* C?
- ✗ Low-level knowledge of each OS
- ✗ Randomized memory, non-exec memory...
- ✗ Need to talk to other libraries.
- ✗ Maintainability

# ECL's new design

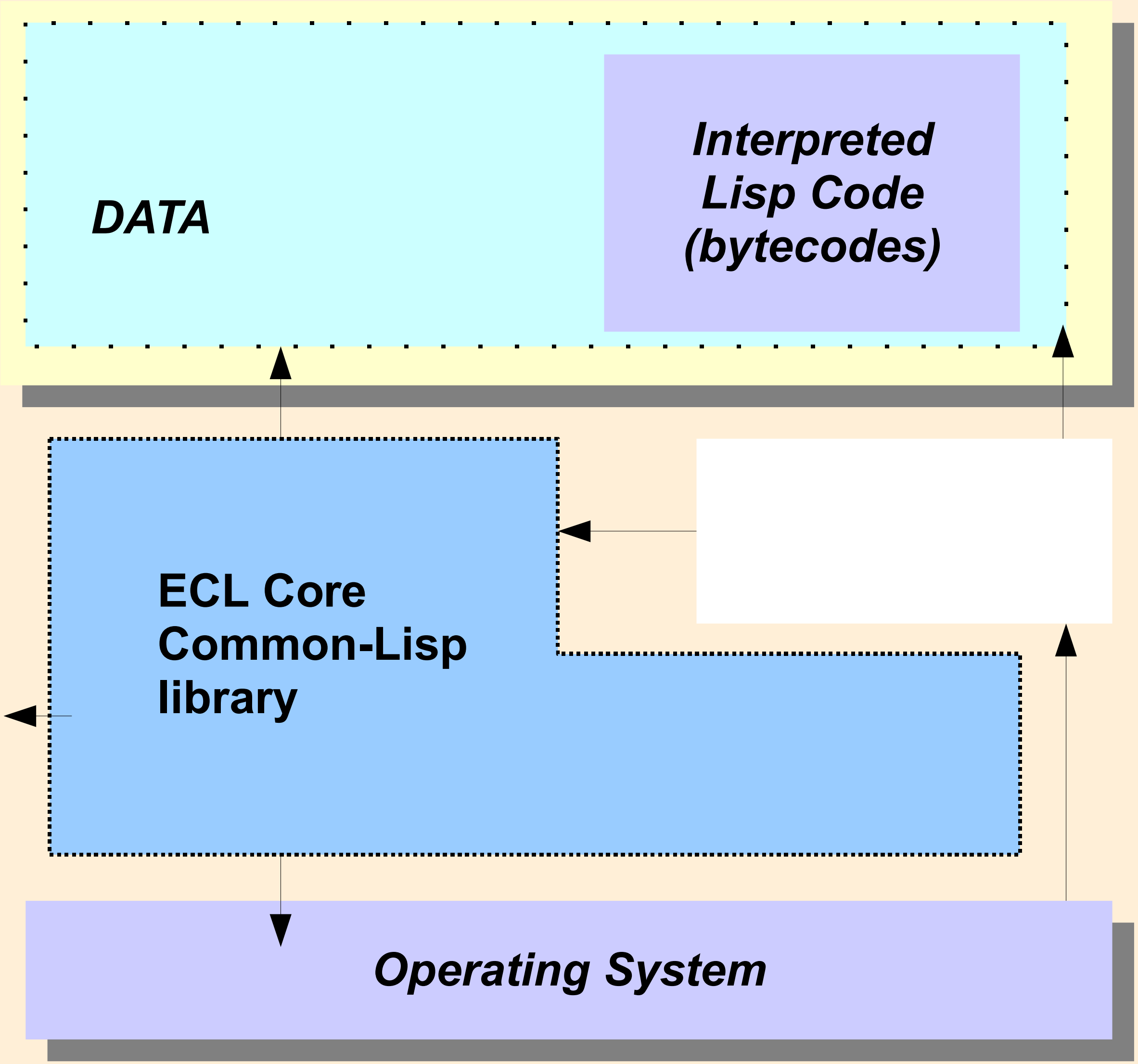


# ECL's design



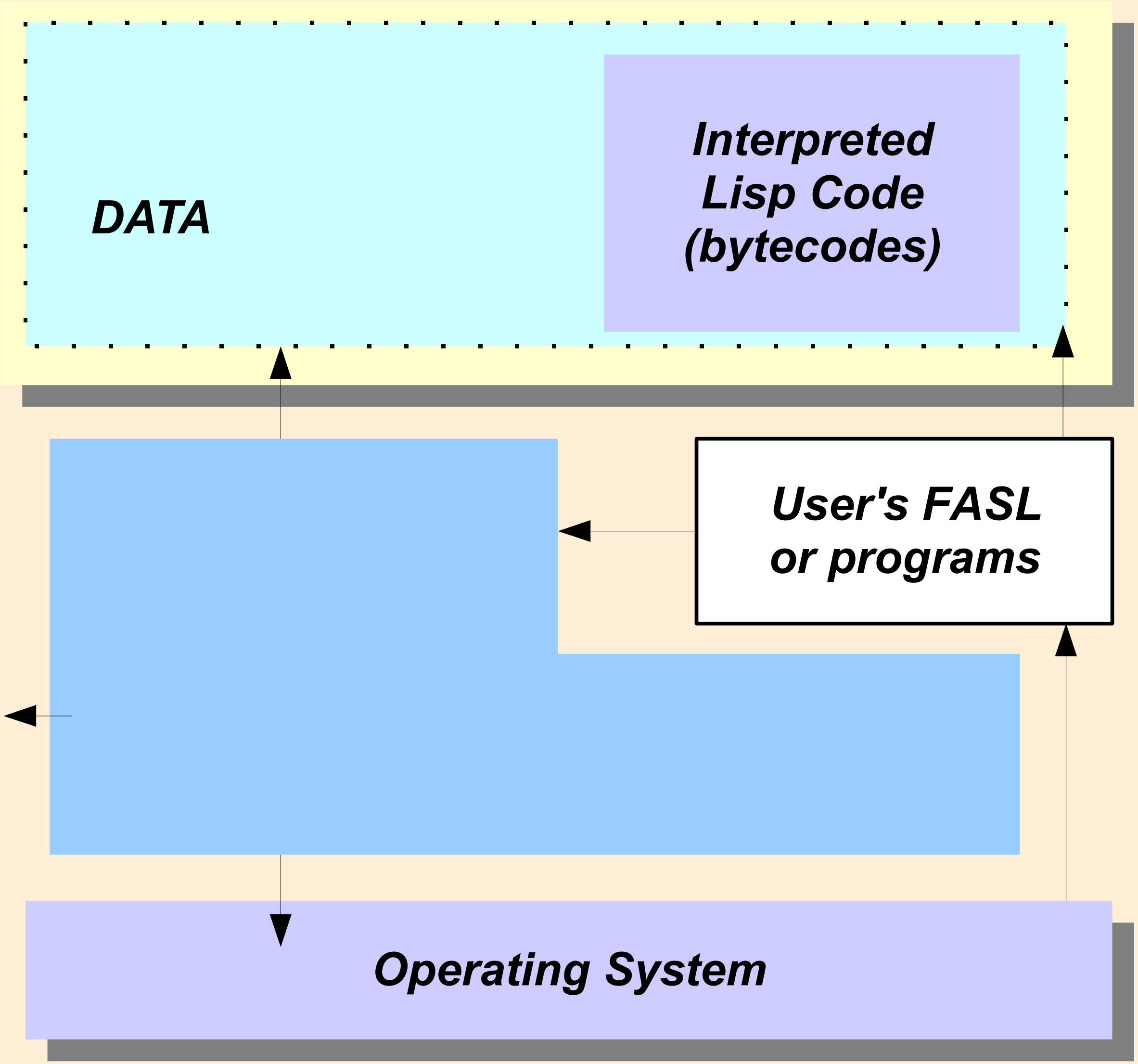
- Lisp has to coexist with other libraries.
- ECL knows about foreign datatypes.
- We know how to find and talk to those libraries
- OS will not allow us full control of memory.
- GC can be performed by external libraries.

# ECL's design


DATA
Interpreted Lisp Code (bytecodes)
ECL Core Common-Lisp library
Operating System

- Binary and data separate
- Pack everything into a standalone library
- Library can be used from other binaries / applications
- Similar to C (C++) philosophy
- Comes with a bytecodes interpreter and compiler

# ECL's design



- Compiled files are just binaries loaded by the OS.
- They are linked to the ECL library
  - lisp objects creation and manipulation
  - talk to other binaries
- No difference bw. embedding ECL and its ordinary use.

# Portability

- Memory management delegated to a GC library
- Use of standard C compilation and linking facilities
- Minimalistic assumptions on architecture
  - We can make pointer ⇆ integer conversions
  - C functions can be called with any # arguments
- Anything nonportable is optional & detected at configuration time.
  - Binary file handling using OS facilities if available
    - dlopen, Mach Kernel, etc
  - Sockets, CLX, long floats, ...

# Portability

**Linux, Net/Free/OpenBSD,**
**Windows' MSVC++,**
**Cyg/Mingwin,**
**Mac OS X, Solaris...**

**Intel 32/64 bits, PPC,**
**Sparc, ARM...**

Wishlist: Cell(PS3),
AIX, iPhone

- Memory rary
- Use of st acilities
- Minimalis
  - We ca s
  - C func ents
- Anything at configura
  - Binary ilable
    - dlop
  - Sockets, CLX, long floats, ...

# Compiled code

- One C function per lisp function.
- Use of standard C constructs.
- Up to 64 args in C stack, rest in interpreter stack.
- Return first value directly, rest in a thread-local array.
- Also closures, unboxed types, inlined C code...

```
cl_object
cl_negate(cl_object x)
{
   cl_object y =
      ecl_minus(MAKE_FIXNUM(0),
                x);
   NVALUES = 1;
   return (VALUES(0)=x);
}

cl_object
cl_floor(cl_narg narg, ...)
{
   cl_va_list args;
   cl_va_start(args,narg,0,narg);
   ...
   NVALUES = 2;
   VALUES(1) = rem;
   return (VALUES(0)=div);
}
```

# Interpreter

- Interpreter, compiler, code walker, stepper & tracer in under 4kloc.
- Handles all special forms
- Support for some macros such as do, dotimes,...
- The C library supplies object handling functions.
- Lisp library adds macros and remaining functions.

- Uses 45 bytecodes, but only about 20 essential

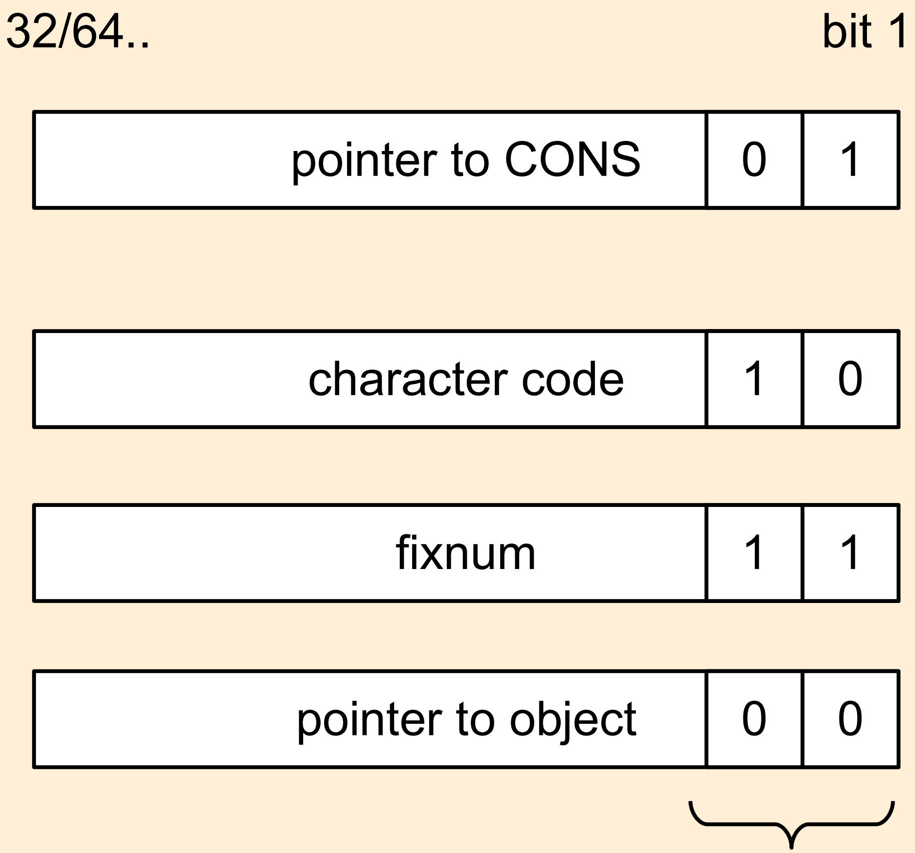
```
> (defun plus1 (x) (+ x 1))
> (si:bc-disassemble #'plus1)
Name:      PLUS1
Required: X
Documentation:    NIL
Declarations: NIL
   0   BLOCK   11,PLUS1
   3   PUSHV   1
   5   PUSH    '1
   7   CALLG   2,+
  10   EXIT    FRAME
  11   EXIT
```

- Very stable, but can be improved.

# Memory management

- Can be completely abstracted
  - alloc_atomic(), alloc(), finalization registration,...
- Currently focused on Boehm-Weiser GC
  - Conservative → works well with foreign libraries
  - Fast, supports heavy loads
  - Used in other projects: GCJ, w3m, ...
  - We still do not use 100% potential
- But you could plug in your favourite GC library

# Data representation (0.9k)



32/64.. bit 1

| | | |
|---|---|---|
| pointer to CONS | 0 | 1 |
| character code | 1 | 0 |
| fixnum | 1 | 1 |
| pointer to object | 0 | 0 |

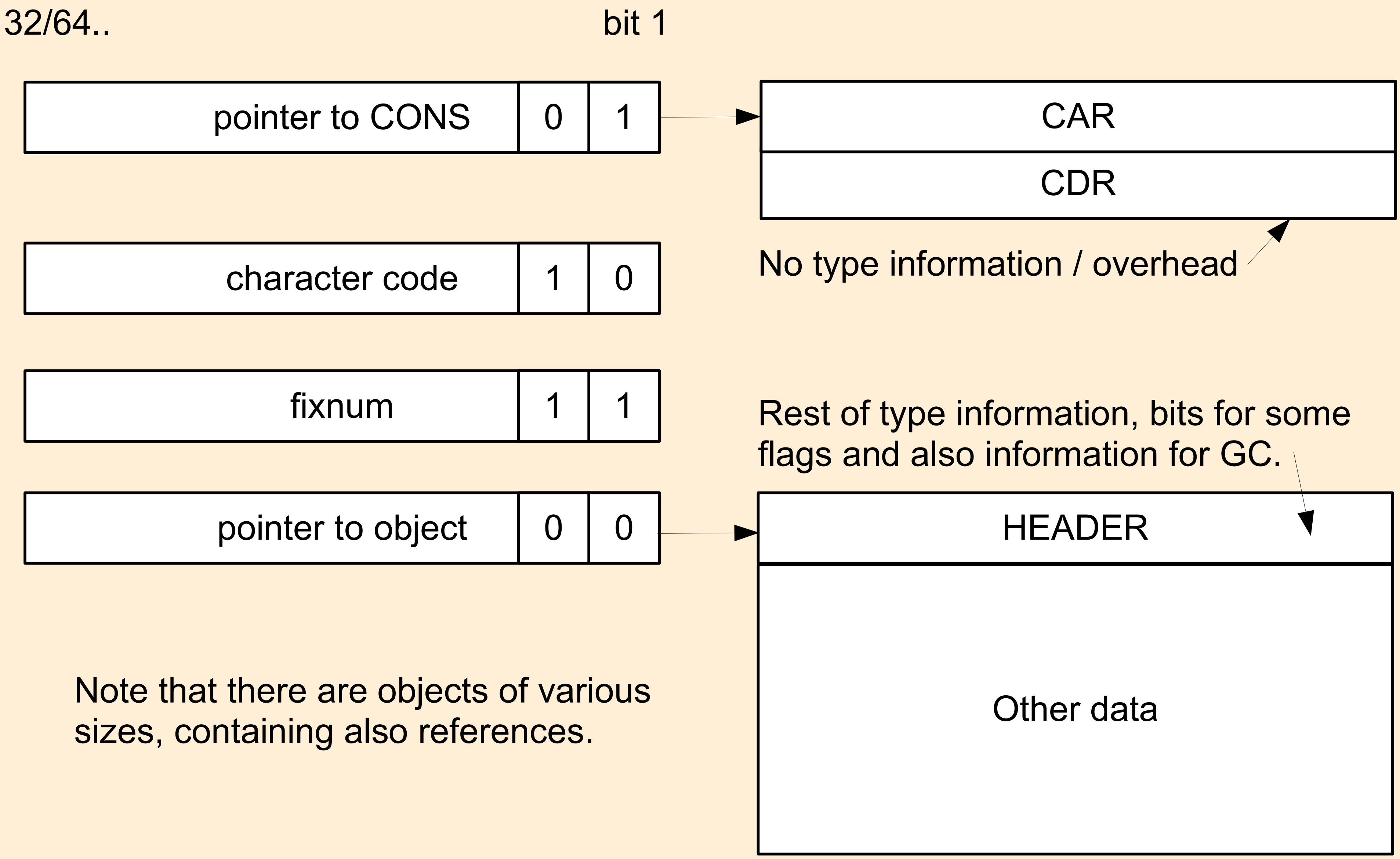
Large enough to fit most of Unicode characters: 30 bits

Immediate integers.

All other boxed types: bignums, arrays, instances, functions, ...

Two bits of information contain some type information and distinguish immediate types.

# Data representation (0.9k)



32/64..

bit 1

| | | |
|---|---|---|
| pointer to CONS | 0 | 1 |
| character code | 1 | 0 |
| fixnum | 1 | 1 |
| pointer to object | 0 | 0 |

CAR

CDR

No type information / overhead

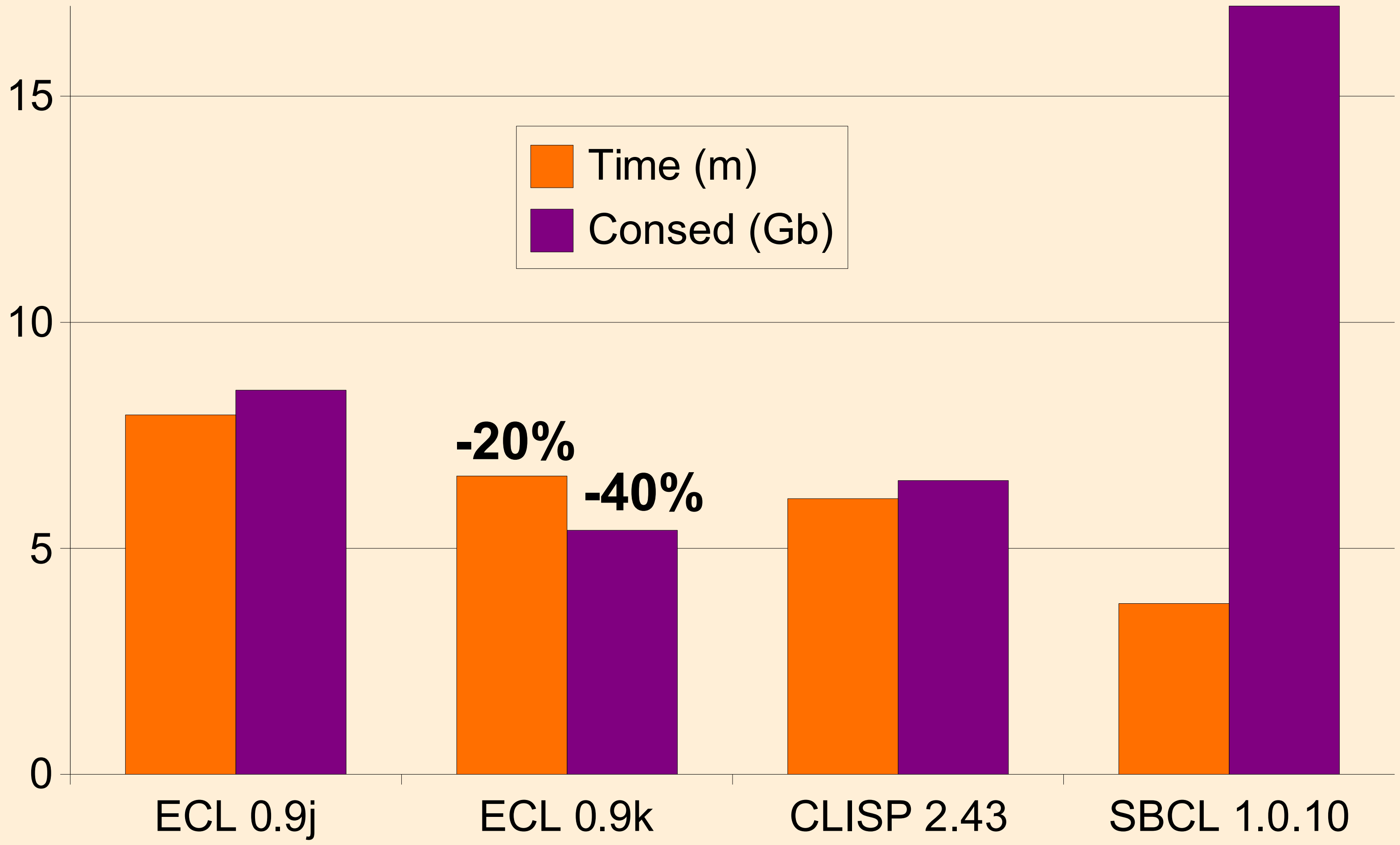
Rest of type information, bits for some flags and also information for GC.

HEADER

Other data

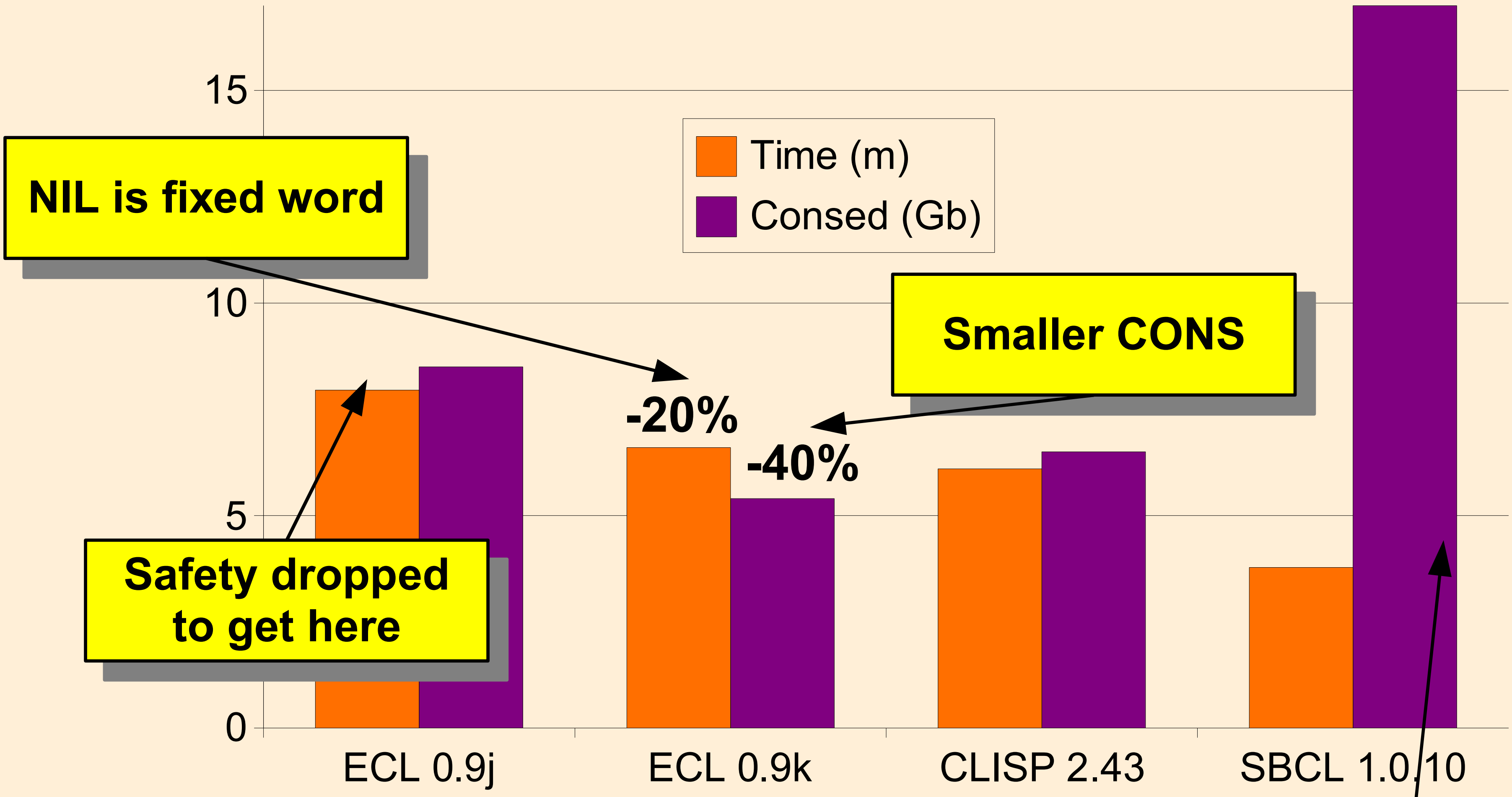
Note that there are objects of various sizes, containing also references.

# Data representation (0.9k)



Performance of Paul Dietz's ANSI Common Lisp, with various implementations, all tested in a Mac OS X 10.4.8 (Tiger)

# Data representation (0.9k)



**NIL is fixed word**

**Smaller CONS**

**Safety dropped to get here**

Performance of Paul Dietz's ANSI Common Lisp, with vario
all tested in a Mac OS X 10.4.8 (Tiger)

**We can learn from this GC!**

# SUBTYPEP: Type lattice

- Following Henry Baker's paper, types are represented as sets, with some types being elementary.
- To each set a binary tag is associated
  - (tag (AND T1 T2)) = (LOGIAND (tag T1) (tag T2))
  - (tag (OR T1 T2)) = (LOGIOR (tag T1) (tag T2))
  - (tag (NOT T1)) = (LOGNOT (tag T1))
- SUBTYPEP **only** fails with recursive types
  - T1 = (OR (CONS INTEGER T1) NIL)
- Works with CLOS.

# CLOS

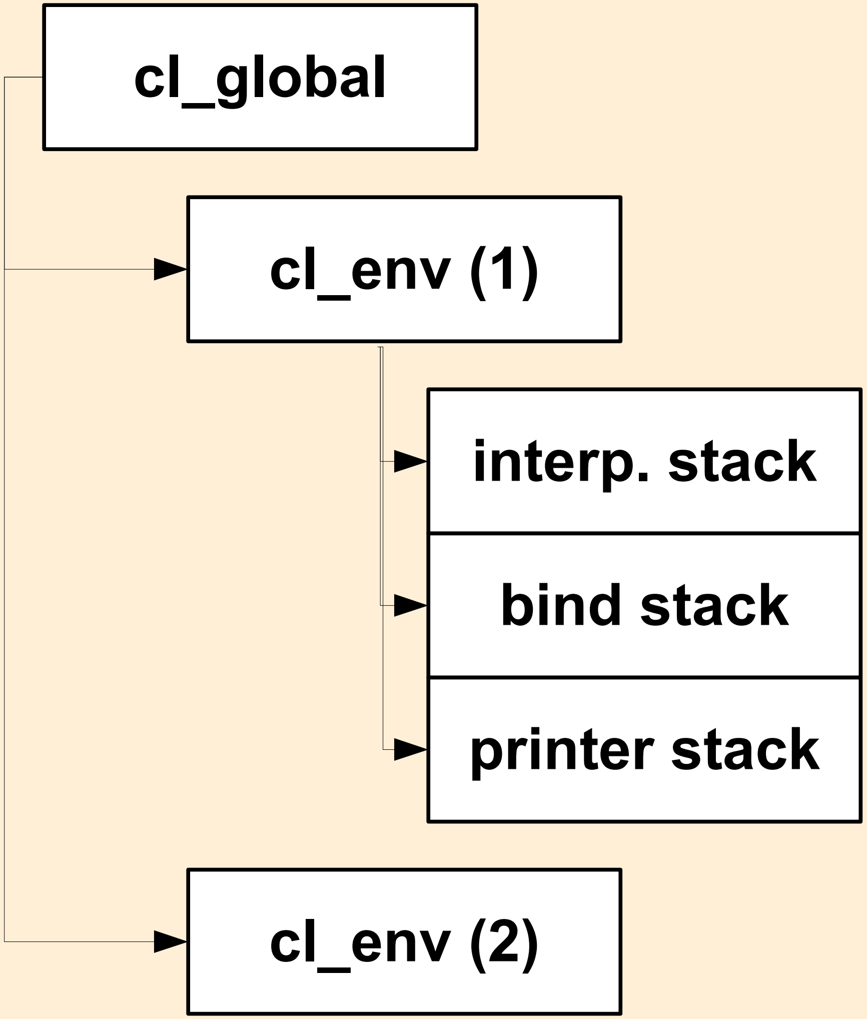
- ECL's implementation derives from a stripped down Portable Common Loops (PCL)
- We have redesigned and extended everything
  - Remember to avoid use of COMPILE!
- Everything in ANSI specification is now provided:
  - standard classes and objects
  - generic functions
  - complex method combinations
- Everything in AMOP, except for custom dispatch.

# CLOS dispatch

- **Thread local method dispatch cache**, shared by all generic functions
  - It can be larger and thus more efficient
  - It cleans itself based on a generation counter
- Function call objects
  - Collect arguments to a generic function
  - Are passed around without further consing
  - Can be efficiently used to invoke a C function
  - Dynamic extent

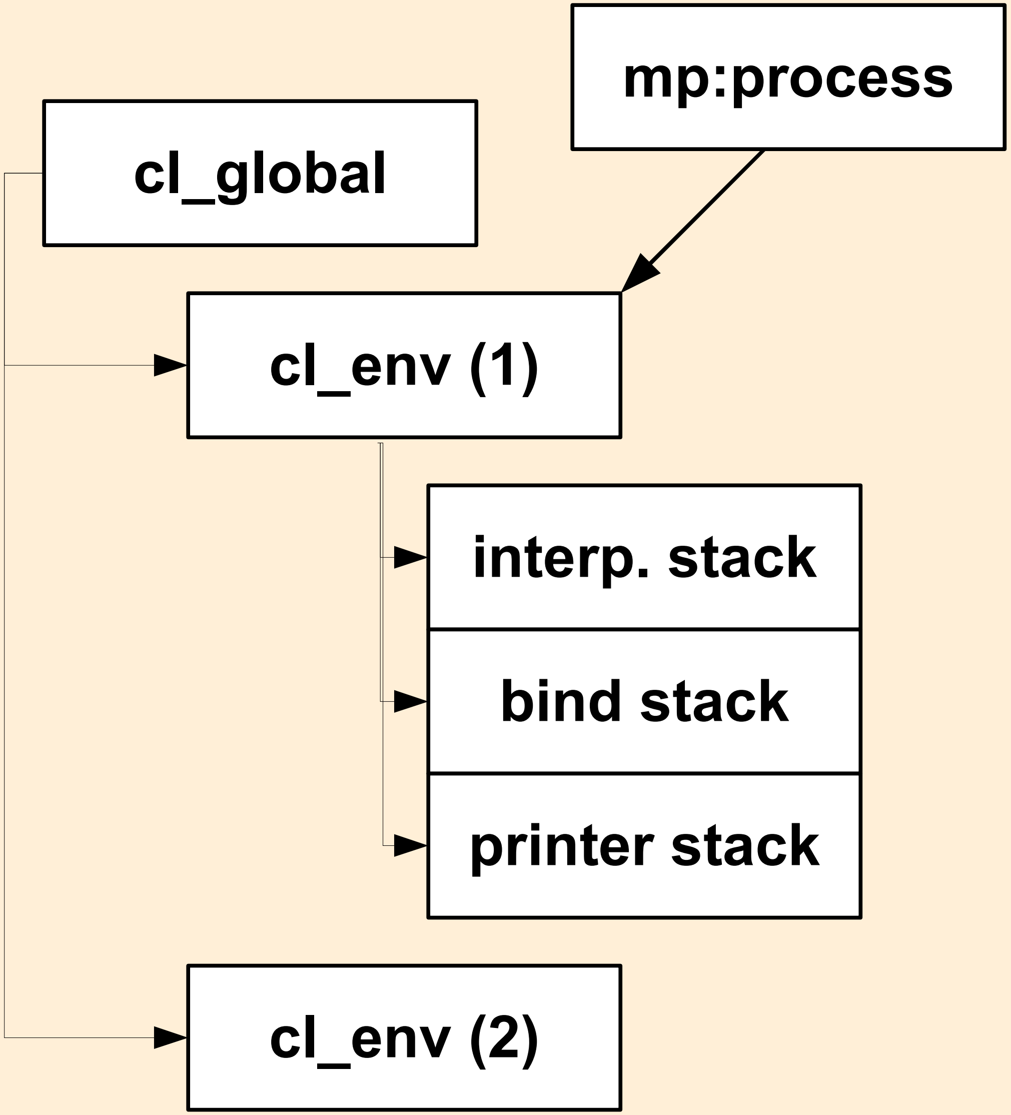
# Environments

- Contain roots to all data
- One global environment
  - packages, symbols, list of libraries, etc.
- One environment per execution line
  - Stacks, bindings, lexical environment...
- Might be sandboxed at different levels.



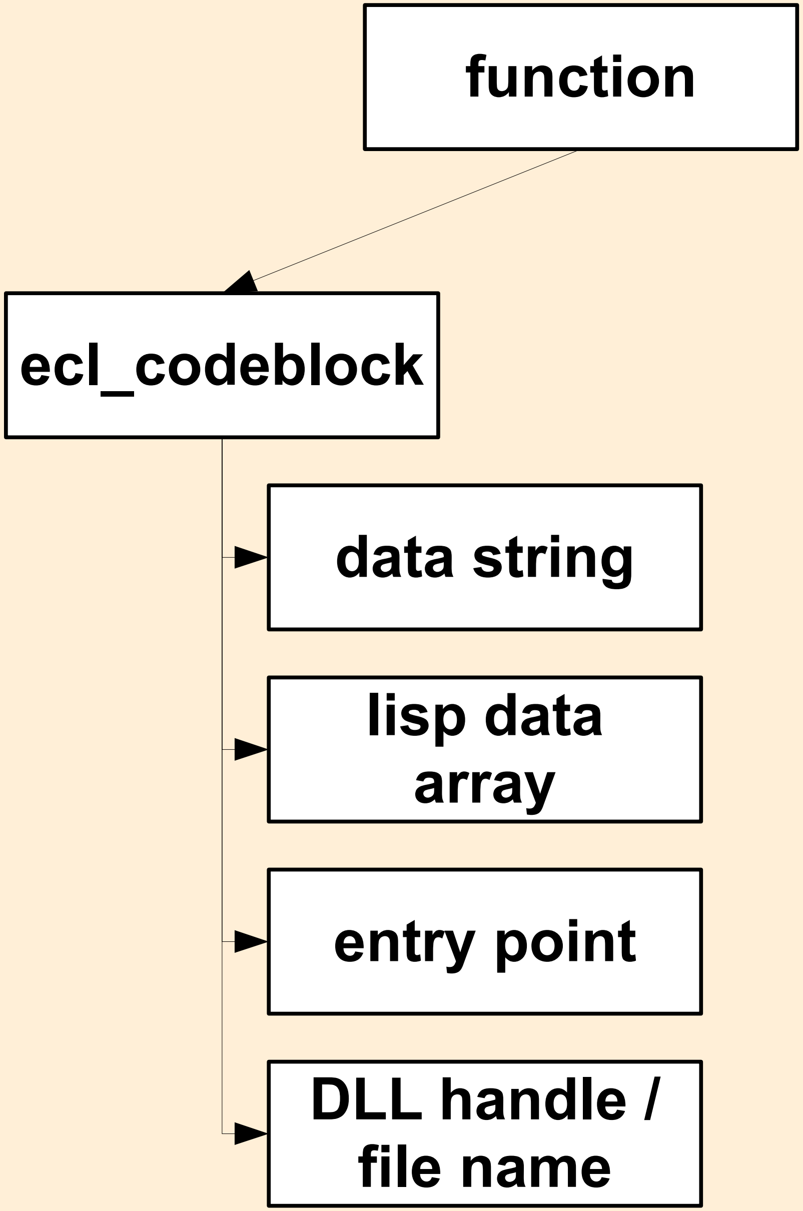
# Multithreading

- Native POSIX threads
- Each thread has access to its own environment
- Global variable bindings in a hash
  - Not too inefficient
- Still a lot to improve:
  - Signals, safety...



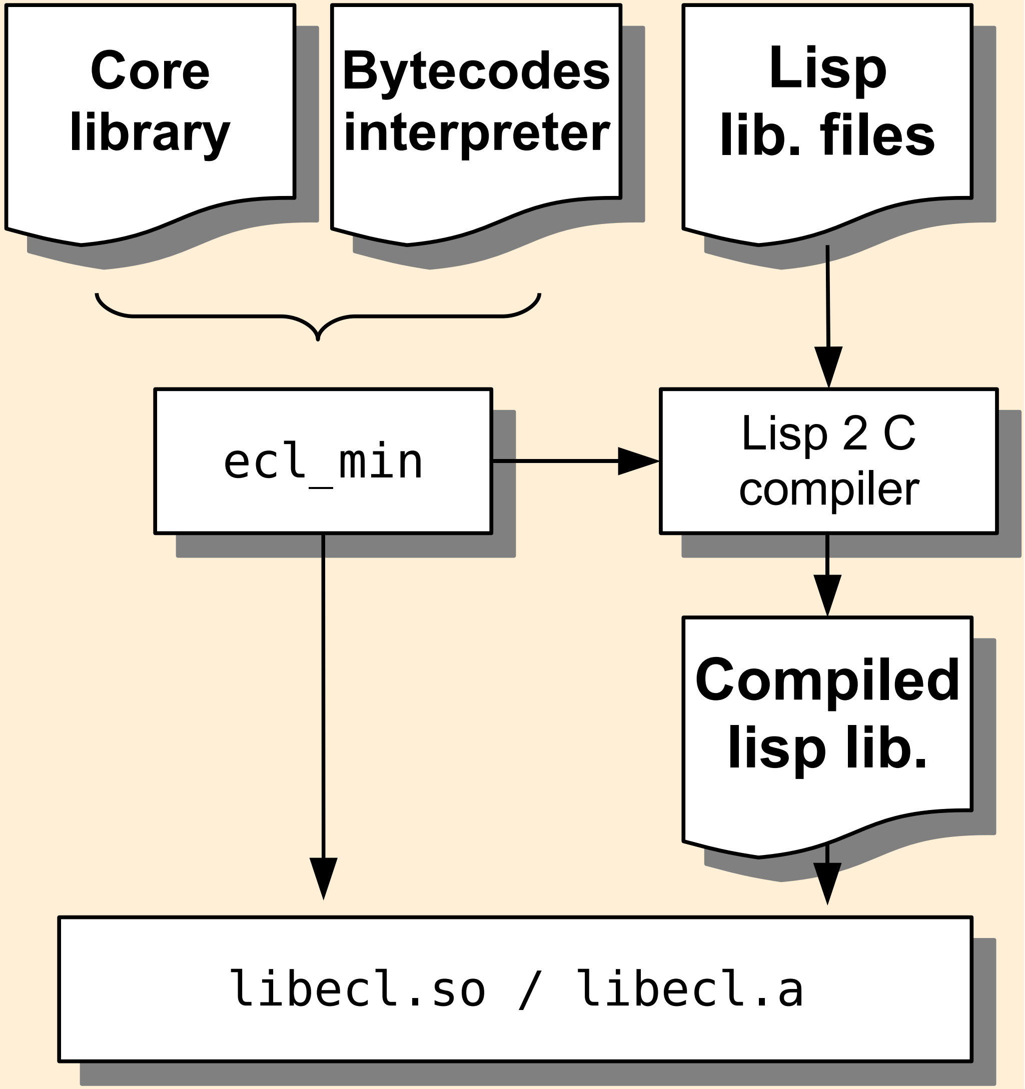
# Binary files

- One entry function.
- Constants in text form.
- Each binary associated to a lisp structure.
- When all functions are garbage collected, the binary file is as well.
- If the binary file was in a DLL, it is closed.
- Completely independent of binaries' purpose.



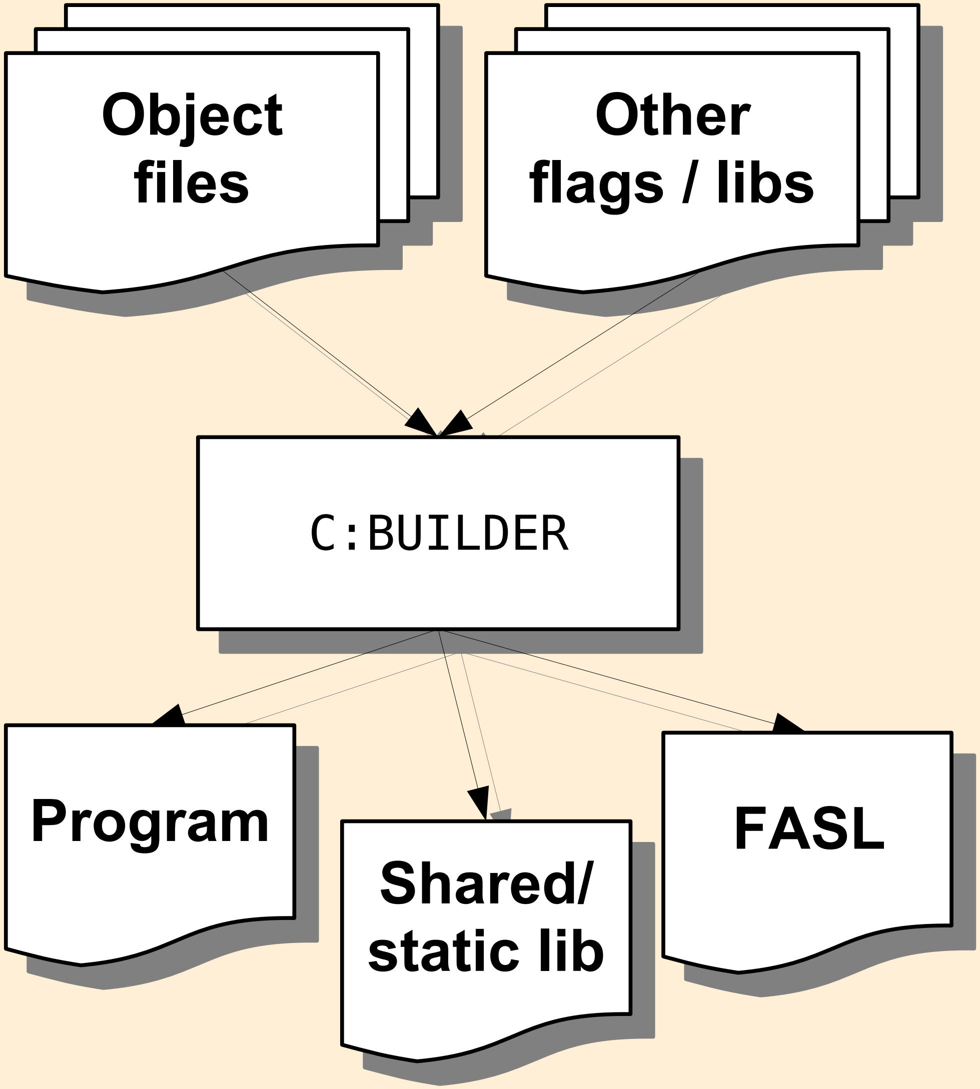
# System building: bootstraping

- The ECL interpreter can handle all Common Lisp.
- Core functionality provided by C library.
- Rest by the lisp library interpreted.
- With this we can run the compiler and compile the whole library.
- Extremely robust



# System building

- ECL knows about the linking abilities of each system
  - no libtool (sucks!)
- A function links object files creating
  - programs
  - static libraries
  - shared libraries
  - bundles (FASL)

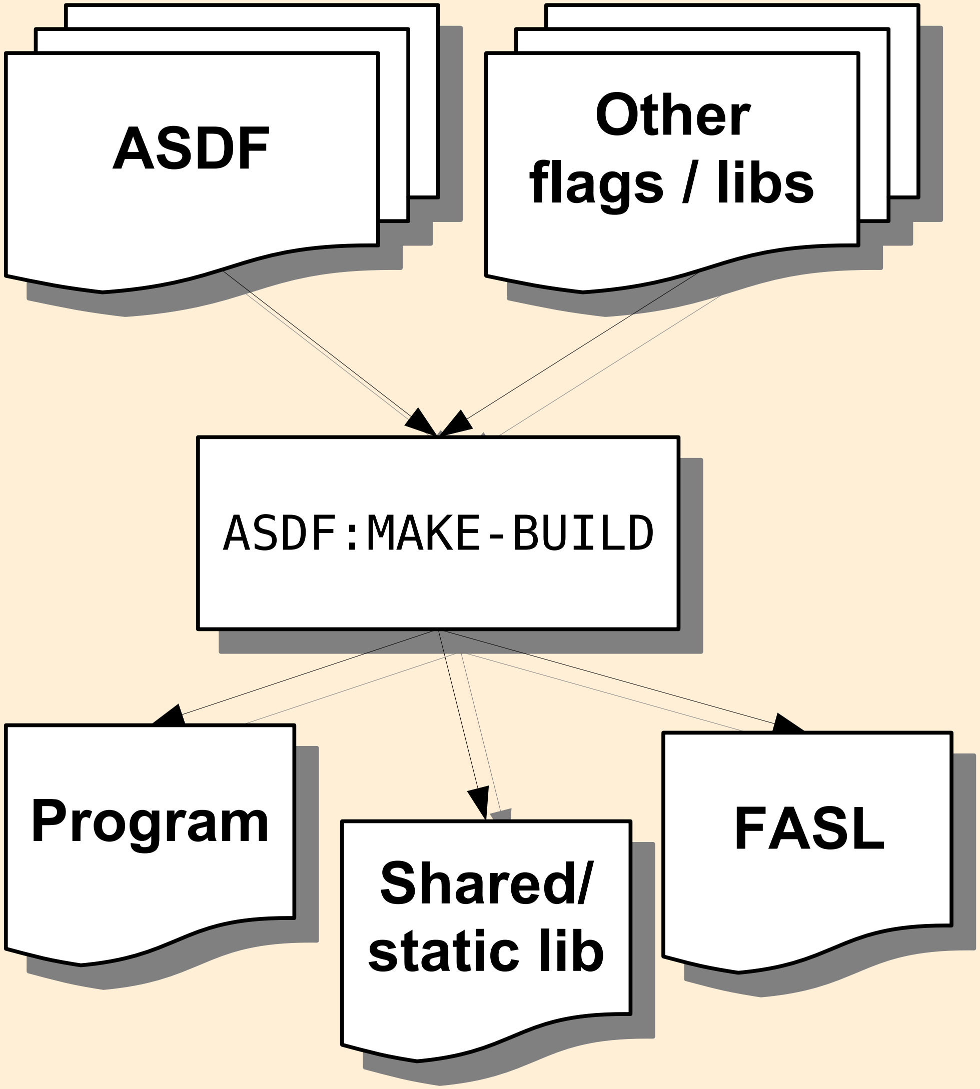


# System building

```
(require 'c)

(defvar *sources*
  '("file1.lsp" "file2.lsp"))

(defvar *objects*
  (loop for i in *sources*
     collect (compile-file i :system-p t)))

(c::builder :program "test"
       :lisp-files *objects*
       :epilogue-code '(format t "~%CLOSING~%"))
```

# System building

- Similar features built into our port of ASDF
- MAKE-BUILD takes a system definition file and builds programs, libraries, FASL
- Can build monolithic systems containing **all** dependencies.
- Still under development



# System building

```
(require 'asdf)
(require 'c)

(asdf:defsystem test
 :components
 ((:file "file1")
  (:file "file2")))

(asdf:make-build :test :type :program
      :epilogue-code '(format t "~%CLOSING~%"))
```

# FFI = foreign functions & callbacks

**Way 1: use C**

- Generate wrappers for each function.
- Code to translate lisp object into C and viceversa.
- Portable.
- Not so much space efficient.

**Problems:**

- Lispers themselves:
  - too “static”
  - wrappers must be compiled.
- Wrong assumptions out there:
  - vararg C functions are just like ordinary ones

# FFI = foreign functions & callbacks

**Way 2: use assembler**

- Code that invokes arbitrary functions.
- Only requires the “signature” of the function.
- Rather fast.

**Problems:**

- Not portable: low level details of API.
- Non-exec memory.
- Really gory details about registers and argument passing: ABI

# FFI = foreign function interface

- Both backends with choice at run time
  - C interface is provided everywhere.
  - Assembler only for Intel 32 and 64 bits API.
- High level interface is UFFI
  - Quasi standard when developed
  - Reasonably featured. Supports C interface very well.
- Allows most of CFFI
  - ECL provides callbacks, which are outside UFFI.
  - More problems regarding hidden assumptions.

# Embedding: ECL in 12 lines

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

int
main(int narg, char **argv) {
  const char *lisp_code = “(si:top-level)”;
  cl_object output;

  cl_boot(narg, argv);

  si_select_package(make_simple_base_string("CL-USER"));
  output = cl_safe_eval(c_string_to_object(lisp_code), Cnil,
                        OBJNULL);

  cl_shutdown();
  return (output != OBJNULL);
}
```

# The road ahead...

# The simple things

- Finish AMOP support
  - User defined dispatch

  →From 2 to 4 man-week
- Finish ASDF system building interface.

  →From 1 to 2 man-week
- Programatic API to the interpreter & debugger

  →About 1 month
  Needed by Slime
- Polish C interface

  →About 1 month + doc time
- ECL deployment w.o. compiler

  →Couple of days.

# Streams & Unicode

- Move from using C FILE to using open(), read()...
  - → Faster & more flexible I/O strategies
- Implement own buffering techniques.
  - → Needed for Unicode.
- Implement input/output formats.
  - → UTF-8, ISO-Latin,...
- Redesign streams as CLOS objects.
  - → Simple streams? Gray?
  - → Easier extensibility.
  - → Requires faster dispatch.

# Lisp2C compiler

| | |
|---|---|
| • Clean up code | → Still a lot of legacy code |
| • Introduce environments | → Branch local type info. |
| | → Access to compiler info. |
| • Better type inference | → Database for CL library. |
| • Unboxed functions and data, with less consing. | |
| • Improve usability | → Clean environment |
| | → Use conditions |
| | → Better specified behavior |

# Function calls optimization

- Implement call dispatch using assembler:
  - Currently a big C switch statement & too many layers
  - Should be faster and avoid duplication of data in stack
- Improve CLOS dispatch
  - Specialized functions for single object dispatch
  - More efficient method combinations
- Improve interpreter
  - Should use tail call optimizations
  - Handle calls to interpreter functions without recursion

# Image dumper

- ECL has two nice features:
  - It knows the structure of all its data
  - It knows the set with all its data
- It is possible to dump all memory data into a file with a relocatable format
  - The equivalent of “lisp image dump”
  - Works with randomized memory and even if ECL does not have control where data will reside
- The data format and serializer routines have already been developed.

# Some wild ideas

- Lisp objects with C unboxed types
  - All objects are CLOS / DEFSTRUCT extensible
- JIT using Tiny C (TCC)
  - Functions are compiled to machine code on the run
- Embedding experiments: Xemacs
  - Already merged Boehm-Weiser gc in Xemacs (2 nights)
  - Would probably simplify Xemacs codebase a lot
  - Initially both languages can coexist.
  - Then, with minor changes to interpreter, ECL takes over

# Need for a “community”

- ECL evolved through periods of one-man maintenance
  - Personal circumstances (job, country switch) slowed development for two years.
- We have had successful “private” collaborations
  - Contributions from companies that use ECL
  - Good license for doing so: LGPL
- A single developer does not have such a wide scope
  - Restricted kind of skills: no web, no GL, no UI
  - Different motivations & interests

# Conclusions

- ECL is a complete Common Lisp implementation.
- **Embedability is an option, not a limitation.**
- ANSI compliance and evolving bells & whistles
- Powerful framework for developing and distributing applications.
- Extremely portable, with little and well isolated system dependencies.
- Its future strongly depends on how the community reacts & contributes.