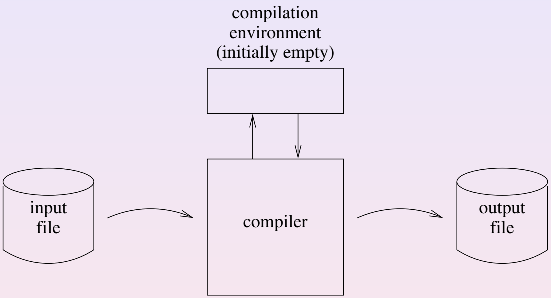
Creating a Common Lisp implementation (Part 1)

Robert Strandh

June, 2020

Compiler for a traditional batch language

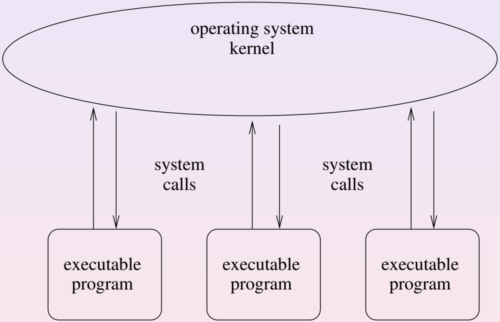


Compiler for a traditional batch language

Characteristics:

- Macros and declarations (implicit or explicit) are entered into the environment.
- The compiler uses the environment to emit warnings, and to determine how to generate code.

Run-time support for a traditional batch language

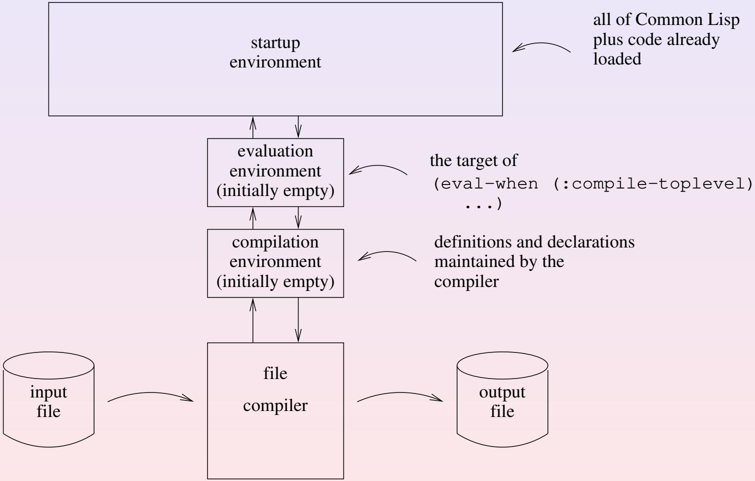


Run-time support for a traditional batch language

Characteristics:

- Each program executes in a separate address space
- Systems calls are used for file I/O, communication between programs, configuration, etc.
- Communication between programs uses pipes, requiring transitions through the kernel.

Common Lisp file compiler



startup
environment

all of Common Lisp
plus code already
loaded

evaluation
environment
(initially empty)

the target of
`(eval-when (:compile-toplevel) ...)`

compilation
environment
(initially empty)

definitions and declarations
maintained by the
compiler

input
file

file
compiler

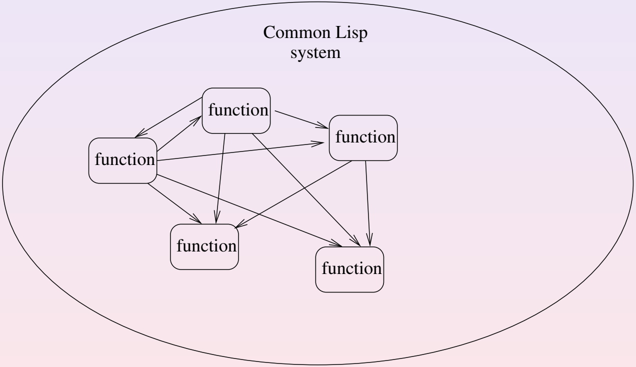
output
file

Common Lisp file compiler

Characteristics:

- Requires an existing Common Lisp implementation (at least partial).
- more?

Run-time support for Common Lisp functions



Run-time support for Common Lisp functions

Characteristics:

- All functions share a common address space.
- Functions call other functions without “kernel” intervention.
- Arbitrary data structures can be passed as arguments.

Creating a Common Lisp implementation

- A compiler for a traditional language is a “simple” file translator.
- Run-time support for a traditional language is provided by the operating-system kernel.
- A compiler for a Common Lisp system is a bit more involved.
- Run-time support for Common Lisp is the Common Lisp system.

Creating a Common Lisp implementation involves writing a compiler, but also creating the run-time support, which has some of the aspects of a traditional operating-system kernel.

General assumptions

- We want to use Common Lisp as much as possible for the implementation.
- The resulting system should not be too slow, but we do not need extremely good performance.
- The implementation we are creating has no classes and no generic functions. This assumption will be revisited later.

Strategy 1: Start with a small core in (say) C

- Write a minimal subset in an existing language.
- Add more and more functionality, written in Common Lisp.

Strategy 1: Core functionality

We need to figure out what initial functionality the core must have.

- A memory manager and garbage collector.
- Code for managing the dynamic run-time environment.
- Allocators, predicates, and accessors for built-in data types.
- Arithmetic functions for fixnums and floats.
- A reader. It must be possible to read additional Common Lisp code.
- An evaluator. The additional Common Lisp code must be executed.
- A printer, i.e., the `print` function of Common Lisp.

Strategy 1: Memory manager and garbage collector

A natural choice:

- Use C malloc() to allocate Common Lisp objects.
- Use the Boehm-Demers-Weiser conservative garbage collector to reclaim memory of dead objects.

Strategy 1: Managing the dynamic run-time environment

This environment consists of:

- Bindings for special variables.
- Tags for `catch` used by `throw`.
- Exit points defined by `tagbody` and `block`.
- Entries for `unwind-protect`.
- Signal handlers and restarts.

Strategy 1: Managing the dynamic run-time environment

The dynamic environment can be allocated on the heap as a linked list of entries:

- An entry type for bindings of a special variables.
- An entry type for `catch` tags.
- An entry type for exit points defined by `tagbody` and `block`.
- An entry type for for `unwind-protect`.

However:

- `catch` and `throw` can be implemented using `block` and `return-from`.
- Signal handlers and restarts can be implemented using special variables.

Unwinding the stack (`throw`, `go`, `return-from`) can be implemented using `setjmp`/`longjmp`.

Strategy 1: Allocators, predicates, accessors

Each built-in data type needs a unique representation:

- Fixnums, bignums, ratios, floats, complex numbers.
- Characters.
- Symbols, packages.
- Conses, arrays, hash tables.
- Streams.

The representation of each type must be determined.

An allocator function, a predicate, and accessors must be defined in the core.

Strategy 1: Reader

The Common Lisp reader is a complicated module, and is best written in Common Lisp, but that's not a choice for the core. Two options:

- Write a subset of the reader in C, capable of reading additional source code. Replace with a full reader written in Common Lisp later.
- Write the final reader in C, but leave out complicated standard reader macros that can be written in Common Lisp later.

Strategy 1: Printer

Like the reader, the Common Lisp printer is a complicated module, and is best written in Common Lisp, but that's not a choice for the core.

The same two options are possible. We will not discuss the printer any further.

Strategy 1: Evaluator

Several possible implementations:

- A direct interpreter written in C.
- A compiler generating native machine code.
- A compiler generating byte codes, combined with a byte-code interpreter written in C.

Strategy 1: Direct interpreter

- Relatively simple.
- Slow.
- “Cross evaluation” is not possible.

Strategy 1: Compiler generating native code

- Hard to write.
- Requires knowledge of the C ABI.
- Makes tail-call optimization somewhat difficult.

Strategy 1: Compiler generating byte codes

- Relatively simple to write.
- Reasonably fast.
- Tail-call optimization is relatively easy.

This is our recommended choice, at least for the time being.

Strategy 1: Complications

Common Lisp does not have a unique set of basic operators. There are many possible choices.

The following link is to a page with a long list of possible choices: http://home.pipeline.com/ hbaker1/MetaCircular.html

Strategy 1: Complications

Perhaps we would like to implement `mapcar` in Common Lisp. Here is a reasonable-looking (simplified) implementation:

```
(defun mapcar (function list)
  (do ((sublist list (rest sublist))
       (result '()))
      ((null sublist) (nreverse result))
    (push (funcall function (first sublist))
          result)))
```

But for this implementation to work, the `do` macro must exist.

And the macro expander for `do` may very well use `mapcar` to extract the local variables, here `sublist` and `result`. SBCL does it that way, for example.

Strategy 1: Complications

Possible solutions:

- Write `mapcar` in the implementation language of the core (C), but this solution is contrary to our goal to use Common Lisp as much as possible.
- Use a simpler iteration construct, hoping its expansion does not require `mapcar`.
- Use recursion, perhaps tail recursion.
- Use `tagbody` and `go` to implement `mapcar`.

A similar decision has to be made for almost all non-trivial function to be implemented.

Worse, during maintenance, these choices have to be known so as to avoid violations of the constraints.

Strategy 1: Complications, example

Example from ECL:

- `mapcar` is defined in C with a lot of macros (extension .d) in a file `mapfun.d` that also defines `mapc`, `maplist`, `mapl`, `mapcan`, and `mapcon`.
- The file has 170 lines of code.
- Executing `sort mapfun.d | uniq | wc -l` gives a bit over 60 unique lines of code.

Disclaimer: It could be written like that for speed.

Strategy 1: Complications

Major complication: How and when do we define the macro `defmacro`?

- The macro expander for `defmacro` is quite complicated. In particular, parsing and generating code for the macro lambda list.
- This task is done by a function, traditionally called `parse-macro` (from the book Common Lisp, the Language).
- Since the macro expander is complicated, it very likely needs to use macros, like `setf`, `multiple-value-bind`, `case`, `cond`, `push`, `when`, `unless`, `dolist`, `dotimes` (from ECL).

We may have to define many of those macros using some mechanism other than `defmacro`.

Strategy 1: Complications, example

Example from ECL:

- Many macros are defined using a mechanism similar to `(setf macro-function)` to avoid the use of `defmacro` before the latter is defined.
- These macros do not use `parse-macro` (or `expand-defmacro` as it is called in ECL).
- The lambda lists of these macros are parsed with special-purpose code.

Again, we have code that is not “natural”, and we end up with duplicated code.

Strategy 1: Complications

Summary of problems:

- We end up with a lot of C code, despite our best intentions.
- We may end up with multiple versions of the same module, one (simplified) C version, and one (complete) Common Lisp version.
- Much of the Common Lisp code is not written in the most “natural” way.
- There are many, often implicit, dependencies between modules.
- The resulting system is hard to maintain, especially when dependencies are implicit.

Thank you