Creating a Common Lisp implementation (Part 3)

Robert Strandh

September, 2020

Topics covered in presentation series

- Choices of implementation language.
- Implementation strategies for the evaluator.
- Division of code written in Common Lisp and code written in the implementation language.

Topics not covered in presentation series

- How a Common Lisp compiler works. If there is a demand, maybe in a different series.
- How different strategies for memory management work.
- Details about how an abstract machine could work.
- Details about how a typical concrete processor works.

Interest for some of these topics has been expressed. We may create talks on these topics in the future.

General assumptions (from Part 1)

- 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

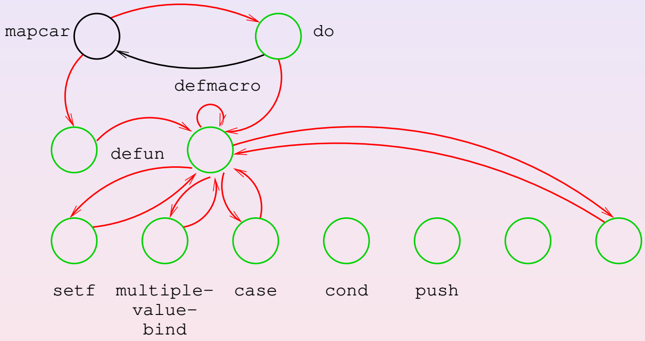
- Write a minimal subset in an existing language.
- Add more and more functionality, written in Common Lisp.

Strategy 1: Complications

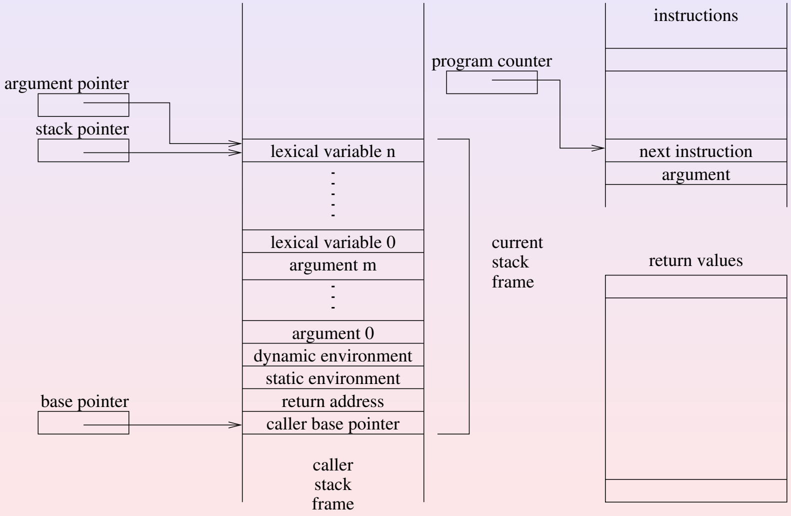
Summary of problems:

- 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.

Circular compile-time dependencies



Abstract machine



What can be done?

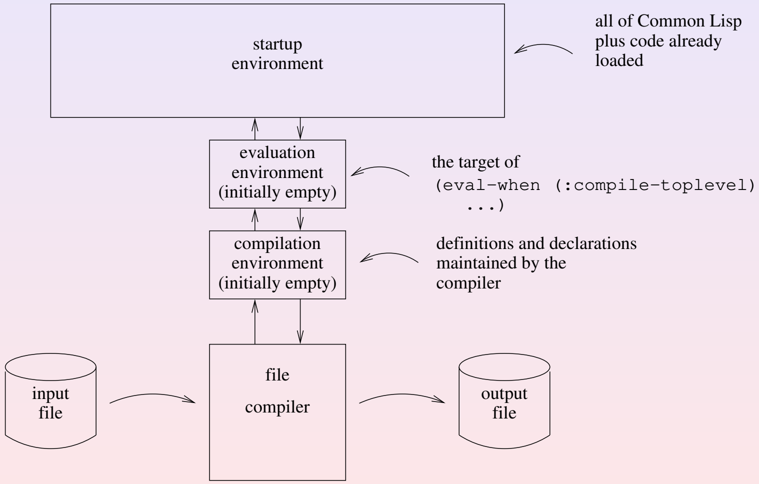
Hypothetical situation

Let us imagine the following scenario:

- We have managed to write a conforming Common Lisp implementation.
- We wrote an evaluator in C, as a direct interpreter.
- The result is slow, so now we want to write a bytecode compiler.
- Since we have a working Common Lisp implementation, we can write it in Common Lisp.

This bytecode compiler is the core of strategy 2.

Common Lisp file compiler



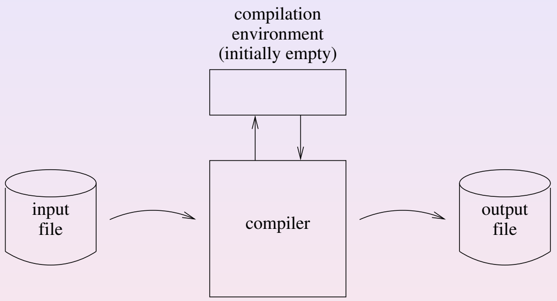
Contents of output file

The output file contains elements from three sources:

1. From the code in the input file: Names of functions, variables, etc.
2. From the code generator of the compiler: Instructions, lexical locations, etc.
3. From macros in the startup environment: Names of functions, variables, etc. that the macro expander generated.

The definitions of the macros are specific to the Common Lisp implementation.

Compiler for a traditional batch language



Essential difference

This difference between the two types of languages is crucial:

1. A compiler for a batch language can run anywhere.
2. Because of the macros, a Common Lisp compiler must in principle be executed from the very Common Lisp system that it is written for.

Aside from that pesky little problem, a Common Lisp compiler can be written in portable Common Lisp and can thus be executed from any conforming Common Lisp implementation.

What if a solution existed?

Imagine we found a solution to that pesky little problem.

- We could execute the compiler on a *host* Common Lisp system.
- We could feed the source of the compiler as input files.
- We would then have the compiler, compiled to bytecodes.
- With a simple bytecode loader, written in C, we could then load the compiled compiler.
- And there is no longer any need for our initial interpreter.

This is the essence of *cross compilation*.

Typical native package structure

```
(defpackage #:common-lisp
  (:use)
  (:export #:car #:cdr ... #:mapcar))
---
(defpackage #:target-cons
  (:use #:common-lisp))
---
(in-package #:target-cons)

(defun mapcar (fun list)
  ...)
```

Example file to compile

```
(in-package #:target-cons)

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

Package structure for cross compilation

```
(defpackage #:target-common-lisp
  ...
  ...
---
(defpackage #:target-cons
  (:use #:target-common-lisp))
---
(in-package #:target-cons)

(defun mapcar (fun list)
  ...)
```

ASDF system definition for cross compilation

```
(cl:in-package #:asdf-user)

(defsystem #:target
  :depends-on (#:cleavir-code-utilities)
  :serial t
  :components
  ((:file "target-common-lisp-package")
   (:file "target-cons-package")
   (:file "target-evaluation-and-compilation-package")
   (:file "target-data-and-control-flow-package")
   (:file "load-target-macros")))
```

Package structure for cross compilation

```
(cl:in-package #:common-lisp-user)

(defpackage #:target-cons
  (:use #:target-common-lisp))
---
(cl:in-package #:common-lisp-user)

(defpackage #:target-data-and-control-flow
  (:use #:target-common-lisp))
---
(cl:in-package #:common-lisp-user)

(defpackage #:target-evaluation-and-compilation
  (:use #:target-common-lisp))
```

The target-common-lisp package

```
(cl:in-package #:common-lisp-user)

(defpackage #:target-common-lisp
  (:use)
  (:import-from #:common-lisp
     .
     #.(let ((result '()))
         (do-external-symbols (symbol (find-package '#:cl))
           (when (or (member symbol lambda-list-keywords)
                     (special-operator-p symbol))
             (push (symbol-name symbol) result)))
         result))
  (:export
   .
   #.(let ((result '()))
       (do-external-symbols (symbol (find-package '#:cl))
         (push (symbol-name symbol) result))
       result)))
```

Load target macros

```
(cl:in-package #:common-lisp-user)

(defun load-target-macros ()
  (setf (macro-function 'target-common-lisp:defmacro)
        (macro-function 'defmacro))
  (do-external-symbols (symbol (find-package '#:common-lisp))
    (when (and (fboundp symbol)
               (null (macro-function symbol))
               (not (special-operator-p symbol)))
      (setf (fdefinition (find-symbol (symbol-name symbol)
                                      '#:target-common-lisp))
                         (fdefinition symbol))))
    ... [same for (setf <foo>) functions]
  (load "defmacro-defmacro.lisp")
  (load "lambda-defmacro.lisp")
  (load "return-defmacro.lisp")
  (load "when-defmacro.lisp")
  ...)
```

Load target macros

```
(cl:in-package #:target-evaluation-and-compilation)

(defmacro defmacro (name lambda-list &body body)
  `(setf (macro-function ',name)
         ,(cleavir-code-utilities:parse-macro
            name lambda-list body)))
```

The parse-macro function is from CLtL2, and can be written in portable Common Lisp.

Load target macros

```
(cl:in-package #:target-evaluation-and-compilation)

(defmacro lambda (lambda-list &body body)
  `(function (lambda ,lambda-list ,@body)))
```

Load target macros

```
(cl:in-package #:target-data-and-control-flow)

(defmacro return (&optional (form nil))
  `(return-from nil ,form))
```

Load target macros

```
(cl:in-package #:target-data-and-control-flow)

(defmacro when (test &body body)
  `(if ,test (progn ,@body) nil))
```

Minor issues

The compiled files will refer to package `target-common-lisp` rather than to `common-lisp`.

To handle that issue, when the compiled file is loaded into the target system, make `target-common-lisp` a nickname of the package `common-lisp`.

Target-specific function executed by target macro expanders must also be loaded on the host.

What we have accomplished

We can compile any target code on the host.

And *any* includes the evaluator and the reader.

The only C code left is the bytecode interpreter and the abstract machine.

We can do even better

We don't have to compile to bytecodes. We can compile to native code.

Building the target system from compiled files can be a bit tricky, but it can be done.

This technique is essentially what SBCL uses for its bootstrapping procedure.

Topics for part 4

Recall that we are creating a pre-ANSI Common Lisp implementation, so it has no classes and no generic functions.

In part 4, we examine the impact on our implementation of not having CLOS from the start.