The Common Lisp Object System

Michael McThrow

San Jose State University
Computer Science Department
CS 152 – Programming Paradigms

December 1, 2021

Table of Contents

Table of Contents

History of Early Lisp

- 1958 – Lisp was first specified by MIT professor John McCarthy
- 1962 – First complete Lisp compiler was implemented
- 1962 – Lisp 1.5 released; first version of Lisp to see widespread distribution.

As Lisp spread beyond MIT, various users started extending the language, sometimes in incompatible ways.

Two competing dominant dialects of Lisp emerged:

- MacLisp (from MIT. The "Mac" long predates the Apple Macintosh. "Mac" refers to Project MAC)
- Interlisp (from Xerox PARC)

There were also many other dialects that were less widespread.

The Scheme Programming Language

- Scheme emerged in the mid-1970s, originally as part of a research project exploring actor models, which were influenced by Smalltalk's message-passing model of object-oriented programming.
- One major contribution of Scheme to Lisp was introducing lexical scoping from Algol. Traditionally Lisp used dynamic scoping, which will be explained in a later slide.
- While Scheme originally came about from research in actor models, Scheme turned into a project that focused on how to simplify and refine Lisp to make it closer to its lambda calculus roots.
- Scheme became very influential in CS education (for decades it was the introductory language at MIT and UC Berkeley) and in programming language research.

Lisp and Artificial Intelligence

- What was fueling the development of all of these incompatible dialects of Lisp?
- AI was Lisp's major application from Lisp's inception until roughly the 2000's when AI started shifting from symbolic AI (e.g., symbolic algebra, knowledge engines, path-finding algorithms, genetic programming, etc.) to statistical-based AI (e.g., machine learning, deep neural networks).
- There were regional differences. American AI researchers tended to prefer Lisp, while European and Japanese AI researchers tended to prefer Prolog.
  - However, there have been many American contributions to Prolog, and there have been many European and Japanese contributions to Lisp.

The Need for a Common Lisp

By the 1980's, there was a need for cooperative development of a Lisp dialect that can be run on a variety of hardware configurations and can meet present and future programming needs while being able to have some degree of compatibility with existing Lisp dialects.

The Need for a Common Lisp

- Thus, Common Lisp was born, designed by a committee of prominent Lisp developers and researchers from a variety of institutions.
- Some famous members included:
  - Richard Gabriel (most famous for his "Worse is Better" thesis explaining how Unix and C edged out Lisp in the marketplace)
  - John McCarthy (creator of Lisp)
  - Richard Stallman (yes, the same RMS who'd later start the GNU Project)
  - Guy L. Steele, Jr. (who'd later work on Java at Sun, which later got acquired by Oracle; he is still at Oracle).

Table of Contents

Common Lisp Overview

- Common Lisp is a huge language.
  - According to The Common Lisp HyperSpec, "[t]he ANSI Common Lisp standard contains nearly 1100 pages describing nearly a thousand functions and variables...."
  - By comparison, the R5RS Scheme specification is only 50 pages.
  - The ISO C++20 standard is 1,853 pages.
- Common Lisp is a multi-paradigm language; it supports procedural, functional, and object-oriented styles and does not favor one over others.

Common Lisp Implementations

- Like C and C++, and unlike languages like Java and Python, there is no "blessed" reference implementation of Common Lisp; there are many competing implementations.
  - The most well-known open source implementations are Steel Bank Common Lisp (SBCL), CMU Common Lisp, Armed Bear Common Lisp (which runs on the JVM), CLISP, Clasp (compiles to LLVM), Embeddable Common Lisp (ECL), and Clozure Common Lisp (not to be confused with Clojure, an entirely different dialect of Lisp).
  - Allegro Common Lisp and LispWorks are feature-rich proprietary implementations.
  - Historical implementations include those written for 1980's Lisp machines, such as Symbolics Genera and Xerox Interlisp-D.

At first glance, Common Lisp looks quite similar to Scheme:

```
(+ 2 5)
(append '(1 2 3) '(4 5 6))
((lambda (x) (* x x)) 5)
```

However, there are many differences between Common Lisp and Scheme. I won't cover all of them, but I will cover the most relevant ones.

Defining Things in Scheme vs. Common Lisp

- In Scheme, we can use `define` to define variables and functions in the parent environment of the current scope.
- The `define` function does not exist in Common Lisp, however. Instead, there are separate functions for defining variables and functions.

Defining Functions in Common Lisp

```
; Scheme example
(define (add-three a b c)
  (+ a b c))

; Common Lisp example
(defun add-three (a b c)
  (+ a b c))
```

Note the name change from `define` to `defun`, and also notice that in Common Lisp the function name is made separate from the function's parameters.

Defining Variables in Common Lisp

- To define local, lexically-scoped variables in Common Lisp, we can use `let` and `let*`, which work similarly to their Scheme counterparts.
- To define a global constant, use `defconstant` (e.g., `(defconstant PI 3.14159)`)
- To define dynamically-scoped variables, we can use either `defparameter` or `defvar`.
  - `defparameter` can change the value of already assigned variables, while `defvar` can't.
  - Example: `(defparameter *games-played* 100)`. Note that the asterisks are part of the variable name; this is a convention used to indicate dynamically-scoped variables.

Lexical vs. Dynamic Scoping

- In lexical scoping, the scope of variables can be determined statically and is based on the location where variables are first defined.
- In dynamic scoping, the scope of variables is determined at run-time and is based on the location of the call stack where variables are first used.

Example

Borrowed from Professor Thomas Austin's CS 152 slides and adapted for Common Lisp:

```
(defparameter x 42)
(defun foo () (prin1 x))
(defun bar ()
  (let ((x 665))
    (foo)))
(bar)
```

What should `(bar)` print?

Example

Borrowed from Professor Thomas Austin's CS 152 slides and adapted for Common Lisp:

```
(defparameter x 42)
(defun foo () (prin1 x))
(defun bar ()
  (let ((x 665))
    (foo)))
(bar)
```

What should `(bar)` print?

`(bar)` should print 665 since under dynamic scoping, `x` was last set to 665. This holds even though x was locally defined as a lexically-scoped variable in `bar`.

Example

Now let's try the following in Scheme:

```
(define x 42)
(define (foo) (display x))
(define (bar)
  (let ((x 665))
    (foo)))
(bar)
```

What should `(bar)` print?

`(bar)` should print 42 since under lexical scoping, the value of `x` is based on its location in the program.

I just realized that the instructions I gave you for implementing environments in Program 3 implemented *dynamic* scoping instead of lexical scoping, which is part of the Scheme standard. Mea culpa.

Dynamic Scoping Today

Currently dynamic scoping is rare among commonly-used programming languages (though it is still encountered in Bash shell scripts and EMACS Lisp), due to it being easier for programmers to reason with lexical scoping; dynamic scoping can lead to "gotchas" that are hard to debug. (For a similar reason, this is why the use of global variables is discouraged.)

Lisp-1 versus Lisp-2 Environments

- Originally, Lisp implementations only have one environment structure, which contains variable and function definitions. These are known as Lisp-1 variants.
  - Examples include Lisp 1.5, Scheme, and Clojure.
- Common Lisp is a Lisp-2; there are separate environments for variables and functions.

As a consequence of Common Lisp being a Lisp-2, whenever we pass functions as arguments, we need to preface them with the #' prefix:

```
 ; returns (1 4 9 16 25)
(mapcar #'(lambda (x) (* x x)) '(1 2 3 4 5))
; returns 15
(reduce #'+ '(1 2 3 4 5))
```

NIL, Empty Lists, and Booleans

- In traditional Lisp dialects, there are no Boolean symbols `#t` and `#f` (this was introduced in Scheme). Instead, there is a predefined symbol `t` (representing Boolean true) and the symbol `NIL`.
- In Common Lisp, `NIL` is synonymous with the empty list `()`. Unlike in Scheme, the empty list does not need to be quoted in Common Lisp.
  - `(cons 1 NIL)` and `(cons 1 ())` are equivalent
- This can be troublesome (for example, parsing JSON fields where there's a distinct difference in meaning between `false` and `[]`).

NIL, Empty Lists, and Booleans

- John McCarthy would later lament this design choice of treating Boolean false and the empty list as equivalent (but, to be fair, Lisp was invented in 1958).
- There was an opportunity to correct this when Common Lisp was being designed, but this was retained to ease porting of applications made in older Lisp dialects.
- Thus, be careful in Common Lisp in situations where you need to distinguish Boolean false from the empty list.

Imperative Style in Common Lisp

- Unlike Scheme, which is a functional programming language, Common Lisp is multi-paradigm and thus does not promote one paradigm over another.
- Mutation and the use of looping constructs are not only allowed in Common Lisp, they are encouraged.
- There's even a complex `loop` macro that supports a wide range of looping possibilities with fine-grained control.
- While many Common Lisp implementations optimize tail recursive calls, this is not required by the standard, which also is a major factor that guides the community's support for the use of looping constructs.
- While iterative programming is supported in Common Lisp, it is also acceptable to program in functional styles. The goal of Common Lisp is to provide the programmer complete freedom.

Table of Contents

Overview of CLOS

- The Common Lisp Object System (CLOS; rhymes with "loss" when pronounced) is Common Lisp's standard object-oriented interface.
- Like Java and Smalltalk, objects in CLOS are class-based.
- However, unlike Java, in CLOS methods are implemented as *generic functions* that are not embedded in the class definition.
  - This means that methods can be added on-the-fly by using `defmethod`.
- Also, unlike Java, there's no concept of encapsulation. Instance variables, known as *slots*, are accessible via the `slot-value` function.

Example

```
(defclass point
  ()      ; no superclasses
  (x y)) ; x and y are slots

; creates a point object *p1*
(defparameter *p1* (make-instance 'point))

; sets *p1*.x = -1 and *p1*.y = 2
(setf (slot-value *p1* 'x) -1
      (slot-value *p1* 'y) 2)
```

- Some of this is syntactically quite inconvenient compared to what we've seen from Java, Smalltalk, and Self.
- Thankfully CLOS does provide some conveniences in `defclass` to make setting and getting fields (slots) less verbose.

```
(defclass point
  ()    ; no superclasses
  ((x :initarg :x
      :accessor point-x)
   (y :initarg :y
      :accessor point-y)))

(defparameter *p1*
  (make-instance 'point :x -1 :y 2))

(point-x *p1*) ; returns -1
(point-y *p1*) ; returns 2
```

Defining Methods

We can use `defmethod` to define a method.

```
(defmethod distance ((p1 point) (p2 point))
  (let ((x1 (point-x p1)) (x2 (point-x p2))
        (y1 (point-y p1)) (y2 (point-y p2)))
    (sqrt (+ (expt (- x2 x1) 2)
             (expt (- y2 y1) 2)))))

(defparameter *p2*
  (make-instance 'point :x 5 :y 10))

(distance *p1* *p2*)
```

Defining Methods

We can use `defmethod` to define a method.

```
(defmethod distance ((p1 point) (p2 point))
  (let ((x1 (point-x p1)) (x2 (point-x p2))
        (y1 (point-y p1)) (y2 (point-y p2)))
    (sqrt (+ (expt (- x2 x1) 2)
             (expt (- y2 y1) 2)))))

(defparameter *p2*
  (make-instance 'point :x 5 :y 10))

(distance *p1* *p2*)
```

Notice something very interesting in the above definition of the `distance` method?

Dispatching in CLOS

- According to *The Art of the Metaobject Protocol*, "A method is associated with a class if a parameter of the method is *specialized* [emphasis original] to that class. Semantically, what this means is that an argument bound to that parameter must be an instance of that class (or any of its subclasses)" [Kiczales et al. 1991, p. 247].
- In addition, a method is associated not with a specific class, but with the list of classes in the parameter list.

Example of Multiple Dispatch in CLOS

Taken from p. 249-250 of *The Art of the Metaobject Protocol*

```
(defmethod paint ((shape rectangle)
                  (medium vector-display))
...)
(defmethod paint ((shape rectangle)
                  (medium bitmap-display))
...)
(defmethod paint ((shape circle)
                  (medium bitmap-display))
...)
```

The `paint` method that gets executed all depends on the types of the parameters `shape` and `medium`. Because the parameter list has more than one class associated with it, this is an example of *multiple dispatch*.

Multiple Dispatch in CLOS

"Multi-methods (i.e., methods with multiple parameters) are a useful extension of the notion of object-oriented programming, but are the primary place where CLOS breaks the intuition that a method belongs to exactly one class. Multi-method dispatch for generic functions allows a finer breakdown of the operation into appropriate pieces" [Kiczales et al. 1991, p. 250].

Other languages with multiple dispatch include, but are not limited to:

- Julia, a relatively new programming language from the 2010s that is growing in popularity among data scientists as an alternative to Python and R.
- Dylan, a language Apple developed in the early- and mid-1990s that was supposed to support the Apple Newton.

Both Julia and Dylan are heavily influenced by Common Lisp, yet have conventional Algol-style syntaxes.

There are many more features in CLOS, but due to time constraints, I must move on to the metaobject protocol.

Table of Contents

What is a Metaobject Protocol

The book *The Art of the Metaobject Protocol* defines a *metaobject protocol* as "[an] interface to the language that gives users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language" [Kiczales et al. 1991, p. 1].

Motivation Behind Common Lisp's MOP

- Most programming languages enforce a sharp divide between the user of the language and the designer of the language.
- If you're not happy with the features of the language, your options are usually the following (in increasing levels of difficulty):
  - "Suck it up and deal with it."
  - Build tools on top of the language that compile down to the target language (for example, TypeScript sits on top of JavaScript to enable coding in a statically-typed style).
  - Modify the language to add/change the features you want.
    - Not always possible, especially if you don't have the source code to the interpreter/compiler.
  - Create your own programming language.

Motivation Behind Common Lisp's MOP

Some languages do have support for modifying the syntax of the language:

- C++ supports operator overloading.
- C has preprocessor macros.
- Many Lisp dialects have support for hygienic and unhygienic macros.

Even with this, the changes are limited to adding new syntactical constructs to the language. They do not allow changing the semantics of the language.

But what if it were possible to modify the language's semantics without having to modify the source code of the interpreter or compiler?

The Metaobject Protocol

"In a language based upon our metaobject protocols, the language implementation itself is structured as an object-oriented program. This allows us to exploit the power of object-oriented programming techniques to make the language implementation adjustable and flexible. In effect, the resulting implementation does not represent a signle point in the overall space of language designs, but rather an entire region within that space" [Kiczales et al. 1991, p. 1].

The Metaobject Protocol

- To put it in other terms, the MOP presents an API to the programming language. Using this API, a program can access and change the constructs of the language at will and on-the-fly.
- This is far more powerful than macros, which does not change the semantics of the underlying programming language.
- In Common Lisp, the entire Common Lisp Object System is built on top of the MOP.


That's all Folks!
SUPERB wallpapers

Table of Contents

Final Exam

- The exam will be posted Wednesday, December 8 at midnight Pacific Standard Time. You have until 11:59pm PST to turn in the exam via Canvas.
- If you have more than two final exams scheduled on December 8 (not including this one), or if you have extenuating circumstances, then you may take the makeup final exam on Wednesday, December 15. Except in emergencies, please send me an email no later than Monday, December 6 if this applies to you.
- The exam is structured the same way as the midterm, with a mixture of conceptual and programming questions.
- The exam is written under the assumption that it will take two hours to complete.

Final Exam Topics

The following topics are guaranteed to NOT be on the final:

- Trivia about FORTRAN, Algol (but trivia about Scheme, Prolog, Haskell, Smalltalk-80, Self, and Common Lisp covered in class and in the reading assignments is fair game)
- Operational semantics
- Lambda calculus
- Continuations in Scheme
- Unhygienic macros in Scheme (but hygienic macros are fair game)
- Cuts in Prolog, as well as other features not covered in class or the assigned chapters in *The Art of Prolog*.

Also, you will only be asked to write code in either Scheme or Prolog.

On Monday during class time I will host a finals review section. I will take questions from the students about any topic covered in class or in the reading assignments.

Where to Go from Here?

For those of you who are interested in continuing your programming language education, I suggest the following resources:

- Take CS 153 (Concepts of Compiler Design). Keep in mind that CS 154 (Formal Languages and Computability) is required and necessary.
- Take CS 252 (Advanced Programming Language Principles), which is essentially a follow-up to CS 152 and delves deeper into more theoretical and research-level concerns.
- Read journal and conference papers from programming language venues such as SPLASH, OOPSLA, ICFP, and PLDI.
- Read Y Combinator's Hacker News forum; quite often many users post articles about programming languages, historical, modern, and research.

Thank you for a great semester, and I wish all of you well in this course, the remainder of your SJSU career, and your future endeavors!