Programming for Hedgehogs

Péter Salvi

Budapest University of Technology and Economics

2021

Abstract

In this paper I argue that in spite of the constant evolution of programming languages and the emergence of new paradigms, homoiconic metaprogramming *is the only feature that is needed in a language to cater for the changing needs of the programming community. This is also a good excuse for writing a Lisp tutorial.*

Introduction

'The fox knows many things, but the hedgehog knows one big thing.'[†] Lisp knows one big thing: *homoiconicity*, i.e., the fact that program code is valid data, through symbolic programming. This enables the use of advanced metaprogramming techniques, where the language can be extended through ordinary libraries.

In other words, the expressive power of Lisp is virtually unbounded. Here I do not mean the theoretical expressivity of the language, which is the same as for lambda calculus or combinatorical logic,[7] or even some simple cellular automata.[14] That definition would just lead us to the "Turing tar-pit".[12]

There is a practical sense of expressiveness concerned with whether a wide range of problems can be solved in a concise, idiomatic way. In the last few decades various programming paradigms were proposed to improve the programming experience, some of them—like object-oriented programming—immensely popular, and mainstream languages had to be constantly revised to keep up with the times. Thus the transition from C to C++ and Objective-C, or the inclusion of functional programming features in Java 8.

In contrast, extending Lisp or creating domain-specific languages in it does not require amendments to the language specification, and is considered a basic exercise. So much in fact that most advanced textbooks include an example, such as object-oriented programming,[5,11] declarative/logic (Prolog) programming,[1,5,11] concatenative (Forth) programming,[8] lazy streams,[1] or nondeterministic computing.[1,5]

The paper is organized as follows. After building up a minimal subset of Common Lisp, we look at a basic object-oriented programming suite with some variations, followed by a discussion, and concluded by further references for the interested reader.

[†] 'πόλλ' οἶδ' ἀλώπηξ, ἀλλ' ἐχῖνος ἓν μέγα.'—a fragment from the Greek poet Archilochus, made famous by the classic essay of Isaiah Berlin.[3] The title of this paper echoes 'Justice for Hedgehogs', a wonderful book on the unity of value by Ronald Dworkin.[4]

A Lisp Tutorial

In the following I am going to describe a very small subset of Common Lisp—just enough to be able to show the power of metaprogramming. While the fundamentals of Lisp can be shown with only 7 primitives,[6] here we will need a little more than that to also handle side effects.

Basics

Lisp programs are composed of *S-expressions* or *sexp*s:

$$\begin{aligned}
\langle sexp \rangle &::= \langle atom \rangle \mid \langle list \rangle \\
\langle atom \rangle &::= \langle symbol \rangle \mid \langle number \rangle \\
\langle list \rangle &::= \texttt{()} \mid \texttt{(}\langle elems \rangle\texttt{)} \mid \\
&\qquad \texttt{(}\langle elems \rangle\texttt{␣.␣}\langle sexp \rangle\texttt{)} \\
\langle elems \rangle &::= \langle sexp \rangle \mid \langle sexp \rangle\texttt{␣}\langle elems \rangle
\end{aligned}$$

where symbols are alphanumeric identifiers, and numbers use the conventional notation. So

```
(foo (bar 42) () . baz)
```

would be a valid example. A sexp can be evaluated, yielding another sexp, using the following rules:

- *Numbers* are self-evaluating,
- *Symbols* evaluate to the associated value in the current environment,
- *Lists* evaluate to the result of applying the function associated with the first element to the evaluation result of the rest of the elements.

For example, with suitably defined arithmetic functions, `(+ 1 (* 2 3))` evaluates to `7`.

An arbitrary sexp can be quoted with the `quote` special form, e.g. `(quote foo)` evaluates to `foo`. As a syntactic sugar, single quotation marks can also be used, so the example above can be written as `'foo`. This is not restricted to symbols, e.g. `'(+ 2 3)` evaluates to `(+ 2 3)`.

The function `cons` creates a *dotted pair* of its two arguments; the two elements can be extracted with `car` and `cdr`. For example, `(cons 'a 'b)` evaluates to `(a . b)`, and `(car '(a . b))` to `a`. This is a special case of lists—actually lists are just nested pairs, so the following expressions are equivalent:

```
(a . (b . (c . d)))
(a b c . d)
```

When the right element of the innermost pair is the special symbol `nil`, it is normally not printed:

```
(a . (b . nil)) ≡ (a b . nil)
                ≡ (a b)
```

The `nil` atom is very versatile: it is self-evaluating, identical to the empty list `()`, and also doubles as the *false* Boolean value. Note also that `(car nil)` and `(cdr nil)` both evaluate to `nil`.

Any non-`nil` value is considered *true*, but predicates usually return the canonical (self-evaluating) truth symbol `t`. Examples of such functions include `atom`, which is true for atoms and false for lists (but note that the empty list is `nil`, so it is considered an atom), and `eq`, which is true when its two arguments represent the same atom. For example,

```
(eq (car '(a b c))
    (cdr '(b . a)))
```

evaluates to `t`. (The language is white-space-insensitive; expressions can be freely reformatted.)

Another special form is the conditional expression

```
(if pred sexp1 sexp2)
```

which returns `sexp2` when `pred` evaluates to `nil`, and `sexp1` otherwise. For example the following returns `yes`:

```
(if (atom 42) 'yes 'no)
```

Functions

The last of the building blocks we need for a working Lisp is a way to define functions. A function is represented by the sexp

```
(lambda (p1 p2 ...) sexp)
```

where `(p1 p2 ...)` is a list of zero or more symbols, the *parameters* of this lambda-expression. Evaluating this with actual arguments `(a1 a2 ...)` creates an environment where the symbol `p1` is bound to the value `a1`, `p2` to `a2` and so forth, and returns the value of `sexp` when evaluated in this environment. For example, the function application

```
((lambda (x y)
   (cons x
         (cons (car y) nil)))
 'a '(b c))
```

returns `(a b)`.

With this, the language is already Turing-complete,[7] but for convenience we will also use

```
(defun name (p1 p2 ...) sexp)
```

to bind the corresponding lambda expression to the symbol `name` in the global environment, also allowing recursive invocations inside `sexp`. Figure 1 shows two examples.

A few notes are in order. The `funcall` in the first definition applies its first argument, a function, to the rest of its arguments. This is needed because Common Lisp is a *Lisp-2* where there are different namespaces for values and functions, so a symbol has a "slot" for both a value and a function. Lambda expressions bind the value slot, but function application needs the function slot; `funcall` converts between the two. Other Lisps, such as Scheme, have a common namespace and function-valued expressions can be called directly. A relative of `funcall` is `apply`, which treats its last argument as a list of arguments, so e.g. the following expressions are equivalent:

```
(apply f 1 2 '(3 4))
(funcall f 1 2 3 4)
```

The function `caar` in the second definition is the `car` of `car`, i.e., it can be defined as

```
(defun caar (x)
  (car (car x)))
```

There are many similar convenience functions `cadr`, `cdar`, `cddr`, `caaar`, etc.

Side effects

At this point our language is still purely functional. Let us add some functions that modify the environment: `setq` changes the binding of its first argument; `rplaca` and `rplacd` change the *car*- and *cdr*-part of a *cons* cell, respectively.

Since now we have side effects, it makes sense to connect function calls sequentially. The special form `progn` takes any number of arguments, calls them in the given order, and returns the value of the last expression. We also modify `lambda` and `defun` to have an implicit `progn` in their body. The following program evaluates to `((a . d) . e)`:

```
((lambda (x y)
   (rplacd x 'd)
   (setq y 'e)
   (cons x y))
 '(a . b) 'c)
```

```
(defun mapcar (fun list)
  (if (atom list)
      nil
      (cons (funcall fun (car list))
            (mapcar fun (cdr list)))))
```

Ex. `(mapcar (lambda (x) (* x x)) '(1 2 3))` ⇒ `(1 4 9)`

```
(defun assoc (item alist)
  (if (atom alist)
      nil
      (if (eq item (caar alist))
          (car alist)
          (assoc item (cdr alist)))))
```

Ex. `(assoc 'b '((a . 1) (b . 2) (c . 3)))` ⇒ `(b . 2)`

Figure 1: Definition of `mapcar` and `assoc`.

Finally, with `defvar` we can declare global variables. These are special in the sense that they have dynamic lookup, so their values are consulted at the time of evaluation, not at the time of function definition.

Metaprogramming

We already have a core language, but it is somewhat clumsy. Local variables, for example, can only be declared through lambda-expressions. We introduce our last tool, which is to define *macros*—functions that run in compile time and generate source code. A `defmacro` definition has the same syntax as a `defun`, but on evaluation it should return a sexp that will be treated as source code.

Since this requires building lots of lists, there is a convenient syntactic sugar called *quasiquoting*, which is like quoting, but with value substitution. It is written as a backquote (`` ` ``), and "interpolation" inside quasiquotation is done with a comma (`,`). Lists can also be spliced into the sexp with comma-at (`,@`). The expression below evaluates to `(42 (a b) a b)`:

```
((lambda (x)
   `(42 ,x ,@x))
 '(a b))
```

Figure 2 shows an extended example using `defmacro`, creating a more comfortable way of defining new local variables. The `let` form extends the environment in a single step, while the `let*` variant creates new bindings one by one, so each can depend on previous ones.

Some notes on the implementation. The list of arguments in `let` is reinterpreted as a dotted pair: the *car* is the first argument, and the *cdr* is a list containing the rest, so this is a way to define macros with variable arity. A more common—and explicit—way to do this in Common Lisp, which also works for functions, is to add `&rest` (or also `&body` in the case of macros) before

```
(defmacro let (bindings . body)
  `((lambda ,(mapcar 'car bindings)
      ,@body)
    ,@(mapcar 'cadr bindings)))

(defun let*-helper (bindings body)
  (if (atom bindings)
      `(progn ,@body)
      `((lambda (,(caar bindings))
          ,(let*-helper (cdr bindings) body))
        ,(cadar bindings))))

(defmacro let* (bindings &body body)
  (let*-helper bindings body))
```

Examples (both evaluate to `5`):

```
(let ((a 1)                (let* ((a 1)
      (b 2))                      (b (+ a 1)))
  (setq a 3)                 (setq a 3)
  (+ a b))                   (+ a b))
```

Figure 2: Definition of `let` and `let*`.

the variable associated with the rest of the arguments, as in the definition of `let*`.

Utilities

Before we go on, let us define some very useful utilities, shown in Figure 3.

- `nth` and `set-nth` query and set the *n*-th element in a list, respectively. The latter is not part of Common Lisp, because there is a very comfortable, general framework for in-place value modification (`setf`) instead, but the details are unfortunately outside the scope of this tutorial.

- `or` returns the first non-nil value of its arguments. It is short-circuiting in the sense that further arguments are not evaluated, so e.g. the following evaluates to `1` without error:

  ```
  (or nil 1 (/ 2 0))
  ```

  In the implementation, `(gensym)` returns a fresh symbol that cannot clash with other variables. This is needed because `f` may already exist in the environment.

- `cond` is a multi-way conditional with implicit `progn`s in the *then*-clauses.

- `member` returns the first tail of the original list that starts with the given item, or `nil` if it is not contained in the list.

```
(defun nth (n list)
  (if (= n 0)
      (car list)
      (nth (- n 1) (cdr list))))

(defun set-nth (n list value)
  (if (= n 0)
      (rplaca list value)
      (set-nth (- n 1) (cdr list) value)))

(defun or-helper (forms)
  (if (null forms)
      nil
      (let ((f (gensym)))
        `(let ((,f ,(car forms)))
           (if ,f
               ,f
               ,(or-helper (cdr forms)))))))

(defmacro or (&rest forms)
  (or-helper forms))

(defun cond-helper (clauses)
  (if (null clauses)
      nil
      `(if ,(caar clauses)
           (progn ,@(cdar clauses))
           ,(cond-helper (cdr clauses)))))

(defmacro cond (&rest clauses)
  (cond-helper clauses))

(defun member (item list)
  (cond ((null list) nil)
        ((eq (car list) item) list)
        (t (member item (cdr list)))))

(defun fold (f init list)
  (if (null list)
      init
      (fold f (funcall f (car list) init) (cdr list))))
```

Figure 3: Definition of some useful functions and macros.

- `fold` reduces a list using a binary operation and an initial element. (The corresponding Common Lisp function is actually called `reduce`, but it would need the introduction of keyword arguments.)

In Figure 4 we also define a simple dictionary data structure, represented as an association list with a `nil` sentinel node.

A SIMPLE OOP SUITE

Common Lisp has an amazing object system,[9] which has been imitated in several other languages. Here we will not make anything near as ambitious as that, only a proof-of-concept for object-oriented programming.

We are going to build a class-based system with single inheritance. A class is represented by a list containing (i) its parent, (ii) the names of its instance variables, (ii) its constructor method, (iii) its instance methods, and (iv) its class variables. All classes are inserted in a special (i.e., global- and dynamic-scoped) dictionary variable. A few accessor functions are also defined for convenience, see Figure 5.

Class definition is handled by a macro that stores variable names and creates dispatch functions for the methods, see Figure 6; some example class declarations are shown in Figure 7.

An instance of a class is represented by a list of its class and a dictionary of values for its variables. Querying and modifying a variable is internally done by `get-var` and `set-var`, going up the class hierarchy until the variable is found, see Figure 8. The macros `value` and `set-value` are just for getting rid of the quoting character.

When creating a method, we need to capture some "keywords":

- `this` refers to the current instance.
- `my` and `set-my` read and write variables through the current instance, respectively.
- `super` calls a method of the parent class.

Calling a method of the parent class needs to know the class in advance, since in that case the dispatch is static, not dynamic. This capturing is performed by `interpret-sexp`, which is called by both `defconstructor` and `defmethod`, see Figure 9. A few method definitions are shown in Figure 10.

We are almost done, only a few functions are missing. The dynamic dispatch calls generated by `defmacro` call `call-method`; the `interpret-sexp` macro also generates code calling `call-constructor`; and we also need a way to construct objects, which will be done by `instance`. The definitions of these functions are shown in Figure 11.

This implementation has some obvious flaws, such as lack of error handling, or efficiency issues (dictionaries would be better implemented as hash tables, for example, which are a part of Common Lisp).

Still, it is a usable system, where we can do something like

```
(let ((s (instance
           square 3 'red)))
  (area-ratio s 36))
```

evaluating to `4`.

Class variables present an interesting challenge. The example in Figure 7 declares the number of vertices as a class variable in `polygon`, with the intent that

```
(defun make-dict () (cons nil nil))
(defun dict-keys (dict) (mapcar 'car (cdr dict)))
(defun dict-values (dict) (mapcar 'cdr (cdr dict)))
(defun dict-get (key dict) (cdr (assoc key dict)))
(defun dict-set (key dict val)
  (let ((entry (assoc key dict)))
    (if entry
        (rplacd entry val)
        (rplacd dict (cons (cons key val)
                           (cdr dict))))))
```

Figure 4: Definition of a dictionary data structure.

```
(defvar *classes* (make-dict))
(defun parent (class)
  (nth 0 (dict-get class *classes*)))
(defun instance-var-names (class)
  (nth 1 (dict-get class *classes*)))
(defun constructor (class)
  (nth 2 (dict-get class *classes*)))
(defun set-constructor (class f)
  (set-nth 2 (dict-get class *classes*) f))
(defun method (class name)
  (dict-get name (nth 3 (dict-get class *classes*))))
(defun set-method (class name value)
  (dict-set name (nth 3 (dict-get class *classes*)) value))
(defun class-var-names (class)
  (dict-keys (nth 4 (dict-get class *classes*))))
(defun class-var (class name)
  (dict-get name (nth 4 (dict-get class *classes*))))
(defun set-class-var (class name value)
  (dict-set name (nth 4 (dict-get class *classes*)) value))
```

Figure 5: Accessor functions for the class data structure.

it would be set to `3` in the `triangle` class, and to `4` in the `square` class. However, in the current implementation, this variable has only one value (residing in `polygon`), and after the calls

```
(set-value
  'triangle vertices 3)
(set-value
  'square vertices 4)
```

it will be set to `4`, and even the query

```
(defmacro defclass (name parent &rest properties)
  `(progn
     (dict-set ',name *classes*
       (list ',parent
             ',(append
                 (instance-var-names parent)
                 (cdr (assoc 'instance-var properties)))
             nil (make-dict)
             (fold (lambda (var dict)
                     (dict-set var dict nil))
                   (make-dict)
                   ',(cdr (assoc 'class-var properties)))))
     ,@(mapcar (lambda (method)
                 `(defun ,method (instance &rest args)
                    (call-method instance
                                 (instance-class instance)
                                 ',method args)))
               (cdr (assoc 'method properties)))))
```

Figure 6: The `defclass` macro.

```
(value 'triangle vertices)
```

would evaluate to `4`. This is how static member variables operate in C++, but we can reinterpret them differently, in a way that each class has its own instances of its ancestors' class variables. We only need to modify `defclass` such that instead of the second call to `make-dict` it deep-copies the dictionary containing the class variables of the parent class.

As a further modification, we can eliminate method declarations—another remnant of conventional OOP frameworks. This is easily accomplished by moving the dispatch template code from `defclass` into `defmethod`. Doing this also has the upside that now we know the exact arguments of the method, so the dispatch function can have meaningful variable names instead of a single `args`. The modified `defmethod` is shown in Figure 12.

The reader is advised to try other modifications, such as moving to a prototype-based framework as in JavaScript or Self (should be even simpler than the current one), or adding abstract classes, information hiding, multiple dispatch, etc.

Discussion

Even through such a trivial example we can see how easy it is to extend the language, and how seamlessly the new contructions fit into the original syntax. While OOP may be a particularly low-hanging fruit, other paradigms can also be implemented in a similar fashion. In addition to the examples enumerated in the introduction, there are libraries for aspect- or context-oriented programming, reac-

```
(defclass polygon nil
  (class-var vertices)
  (instance-var color)
  (method area circumference size))

(defclass triangle polygon
  (class-var angle-tolerance)
  (instance-var a b c)
  (method right-angled-p))

(defclass equilateral-triangle triangle)

(defclass square polygon
  (instance-var a))
```

The definition of `triangle`, for example, is expanded to

```
(progn
 (dict-set 'triangle *classes*
           (list 'polygon '(color a b c) nil (make-dict)
                 (fold (lambda (var dict)
                         (dict-set var dict nil))
                       (make-dict)
                       '(angle-tolerance))))
 (defun right-angled-p (instance &rest args)
   (call-method instance (instance-class instance)
                'right-angled-p args)))
```

(In Common Lisp it is customary for the name of predicates to end with `p`.)

Figure 7: Example class declarations.

tive programming, inductive programming, and so on.

One concern is that such freedom has a price: it can easily be misused, and may harm 3rd-party code readability. This is true, but 'with great power comes great responsibility'; a judicious use of metaprogramming does no harm.

The "comfort" of the programming experience also depends on the environment, such as a fast feedback cycle using a read-eval-print loop, a nice IDE with on-line documentation, a large and logically constructed standard library, and probably some other factors, as well. Homoiconicity does not help here, but fortunately Common Lisp (and some other Lisp variants) do very well in these respects.

Finally, a word about types. Lisp, as outlined above, is basically typeless. Common

```
(defun instance-class (instance)
  (nth 0 instance))
(defun instance-var (instance name)
  (dict-get name (nth 1 instance)))
(defun set-instance-var (instance name value)
  (dict-set name (nth 1 instance) value))

(defun get-var (instance-or-class name)
  (let ((class (if (atom instance-or-class)
                   instance-or-class
                   (instance-class instance-or-class))))
    (cond ((member name (instance-var-names class))
           (instance-var instance-or-class name))
          ((member name (class-var-names class))
           (class-var class name))
          (t (get-var (parent class) name)))))

(defun set-var (instance-or-class name value)
  (let ((class (if (atom instance-or-class)
                   instance-or-class
                   (instance-class instance-or-class))))
    (cond ((member name (instance-var-names class))
           (set-instance-var instance-or-class name value))
          ((member name (class-var-names class))
           (set-class-var class name value))
          (t (set-var (parent class) name value)))))

(defmacro value (instance-or-class name)
  `(get-var ,instance-or-class ',name))

(defmacro set-value (instance-or-class name value)
  `(set-var ,instance-or-class ',name ,value))
```

Figure 8: Handling instances and variables.

Lisp has strong dynamic typing (with optional explicit type signatures), but Lisps normally do not have the kind of strong static typing we see in the ML or Haskell languages. This is in line with the "more freedom to the programmer" philosophy that leads to shorter development times, in exchange for less confidence in program correctness.

```
(defun interpret-sexp (instance sexp super-class)
  (cond ((atom sexp)
         (if (eq sexp 'this) instance sexp))
        ((eq (car sexp) 'my)
         `(get-var ,instance ',(nth 1 sexp)))
        ((eq (car sexp) 'set-my)
         `(set-var ,instance ',(nth 1 sexp) ,(nth 2 sexp)))
        ((eq (car sexp) 'super)
         (if (eq (cadr sexp) 'constructor)
             `(call-constructor ',super-class ,instance
                                ,@(cddr sexp))
             `(call-method ,instance ',super-class
                           ',(cadr sexp) ,@(cddr sexp))))
        (t (cons
            (interpret-sexp instance (car sexp) super-class)
            (interpret-sexp instance (cdr sexp)
                            super-class)))))

(defmacro defconstructor (class args &body body)
  (let ((instance (gensym)))
    `(set-constructor
       ',class
       (lambda (,instance ,@args)
         ,@(mapcar (lambda (sexp)
                     (interpret-sexp instance sexp
                                     (parent class)))
                   body)
         ,instance))))

(defmacro defmethod (class name args &body body)
  (let ((instance (gensym)))
    `(set-method
       ',class ',name
       (lambda (,instance ,@args)
         ,@(mapcar (lambda (sexp)
                     (interpret-sexp instance sexp nil))
                   body)))))
```

Figure 9: Creating methods.

```
(defconstructor polygon (color)
   (set-my color color))

(defconstructor triangle (a b c color)
  (super constructor color)
  (set-my a a)
  (set-my b b)
  (set-my c c))

(defconstructor equilateral-triangle (n color)
  (super constructor n n n color))

(defconstructor square (side color)
  (super constructor color)
  (set-my a side))

(defmethod polygon area-ratio (whole-area)
  (/ whole-area (area this)))

(defmethod square area ()
  (* (my a) (my a)))
```

The constructor of `equilateral-triangle`, for example, is expanded to

```
(set-constructor
  'equilateral-triangle
  (lambda (#:g655 n color)
    (call-constructor 'triangle #:g655 n n n color)
    #:g655))
```

(The symbol `#:g`*n* here stands for the *n*-th symbol generated with `gensym`.)

Figure 10: Example method definitions.

Conclusion and further reading

Programming languages evolve incessantly; new languages are born, and new paradigms emerge. In spite of this, a *hedgehog*—the Lisp family of languages—has needed relatively few changes since its inception more than 60 years ago to keep up with these developments. This is due to its "big thing": its powerful homoiconic metaprogramming capability. It persevered, and I believe it will continue to assimilate all new methodologies it encounters.[10]

I hope that through this little introduction I could convey some of the timeless beauty that pervades this deceptively sim-

```
(defun call-constructor (class instance &rest args)
  (apply (constructor class) instance args))

(defun call-method (instance class name &rest args)
  (apply (search-method class name) instance args))

(defun search-method (class name)
  (or (method class name)
      (search-method (parent class) name)))

(defmacro instance (class &rest args)
  `(funcall (constructor ',class)
            (list ',class (make-dict))
            ,@args))
```

Figure 11: Tying up loose ends.

```
(defmacro defmethod (class name args &body body)
  (let ((instance (gensym)))
    `(progn
       (defun ,name (instance ,@args)
         (funcall (search-method (instance-class instance)
                                 ',name)
                  instance ,@args))
       (set-method
        ',class ',name
        (lambda (,instance ,@args)
          ,@(mapcar (lambda (sexp)
                      (interpret-sexp instance sexp nil))
                    body))))))
```

Figure 12: Moving the dispatch function into `defmethod`.

ple concept which makes an almost minimal language comfortably high-level and modern.

For those who would like to learn more about Lisp, I recommend the Wizard Book,[1] which teaches Scheme, and Peter Seibel's Practical Common Lisp,[13] followed by On Lisp,[5] Let Over Lambda,[8] and Anatomy of LISP.[2‡]

[‡]Note that while we have concentrated on Lisp as the paradigm case for symbolic programming, there are also other (more-or-less) homoiconic languages, such as Prolog or Julia.

References

[1] Harold Abelson, Gerald Jay Sussman, Julie Sussman (1996). *Structure and Interpretation of Computer Programs* (The MIT Press, 2nd edition).

[2] John Allen (1978). *Anatomy of LISP* (McGraw–Hill).

[3] Isaiah Berlin (1953). *The Hedgehog and the Fox: An Essay on Tolstoy's View of History* (Weidenfeld & Nicolson).

[4] Ronald Dworkin (2011). *Justice for Hedgehogs* (Harvard University Press).

[5] Paul Graham (1993). *On Lisp: Advanced Techniques for Common Lisp* (Prentice Hall).

[6] ———— (2002). *The Roots of Lisp* (Draft). http://www.paulgraham.com/

[7] Chris Hankin (1995). *Lambda Calculi: A Guide for Computer Scientists* (Oxford University Press).

[8] Doug Hoyte (2008). *Let Over Lambda: 50 Years of Lisp* (Lulu Press).

[9] Sonya E. Keene (1989). *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS* (Addison–Wesley).

[10] Randall Munroe (2007). *Lisp Cycles* (xkcd). http://xkcd.com/297/

[11] Peter Norvig (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp* (Morgan Kaufmann Publishers).

[12] Eric S. Raymond (1996). *The New Hacker's Dictionary* (The MIT Press, 3rd edition).

[13] Peter Seibel (2005). *Practical Common Lisp* (Apress).

[14] Stephen Wolfram (2002). *A New Kind of Science* (Wolfram Media).