The Roots of Lisp

PAUL GRAHAM

Draft, January 18, 2002.

In 1960, John McCarthy published a remarkable paper in which he did for programming something like what Euclid did for geometry.[1] He showed how, given a handful of simple operators and a notation for functions, you can build a whole programming language. He called this language Lisp, for "List Processing," because one of his key ideas was to use a simple data structure called a *list* for both code and data.

It's worth understanding what McCarthy discovered, not just as a landmark in the history of computers, but as a model for what programming is tending to become in our own time. It seems to me that there have been two really clean, consistent models of programming so far: the C model and the Lisp model. These two seem points of high ground, with swampy lowlands between them. As computers have grown more powerful, the new languages being developed have been moving steadily toward the Lisp model. A popular recipe for new programming languages in the past 20 years has been to take the C model of computing and add to it, piecemeal, parts taken from the Lisp model, like runtime typing and garbage collection.

In this article I'm going to try to explain in the simplest possible terms what McCarthy discovered. The point is not just to learn about an interesting theoretical result someone figured out forty years ago, but to show where languages are heading. The unusual thing about Lisp—in fact, the defining quality of Lisp—is that it can be written in itself. To understand what McCarthy meant by this, we're going to retrace his steps, with his mathematical notation translated into running Common Lisp code.

1 Seven Primitive Operators

To start with, we define an *expression.* An expression is either an *atom,* which is a sequence of letters (e.g. `foo`), or a *list* of zero or more expressions, separated by whitespace and enclosed by parentheses. Here are some expressions:

```
foo
()
(foo)
(foo bar)
(a b (c) d)
```

The last expression is a list of four elements, the third of which is itself a list of one element.

[1] "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I." *Communications of the ACM* 3:4, April 1960, pp. 184–195.

In arithmetic the expression 1 + 1 has the value 2. Valid Lisp expressions also have values. If an expression e yields a value v we say that e *returns* v. Our next step is to define what kinds of expressions there can be, and what value each kind returns.

If an expression is a list, we call the first element the *operator* and the remaining elements the *arguments*. We are going to define seven primitive (in the sense of axioms) operators: `quote`, `atom`, `eq`, `car`, `cdr`, `cons`, and `cond`.

1. `(quote` *x*`)` returns x. For readability we will abbreviate `(quote` *x*`)` as `'`*x*.

```
> (quote a)
a
> 'a
a
> (quote (a b c))
(a b c)
```

2. `(atom` *x*`)` returns the atom `t` if the value of x is an atom or the empty list. Otherwise it returns `()`. In Lisp we conventionally use the atom `t` to represent truth, and the empty list to represent falsity.

```
> (atom 'a)
t
> (atom '(a b c))
()
> (atom '())
t
```

Now that we have an operator whose argument is evaluated we can show what `quote` is for. By quoting a list we protect it from evaluation. An unquoted list given as an argument to an operator like `atom` is treated as code:

```
> (atom (atom 'a))
t
```

whereas a quoted list is treated as mere list, in this case a list of two elements:

```
> (atom '(atom 'a))
()
```

This corresponds to the way we use quotes in English. Cambridge is a town in Massachusetts that contains about 90,000 people. "Cambridge" is a word that contains nine letters.

Quote may seem a bit of a foreign concept, because few other languages have anything like it. It's closely tied to one of the most distinctive features of Lisp: code and data are made out of the same data structures, and the quote operator is the way we distinguish between them.

3. `(eq` x y`)` returns `t` if the values of x and y are the same atom or both the empty list, and `()` otherwise.

```
> (eq 'a 'a)
t
> (eq 'a 'b)
()
> (eq '() '())
t
```

4. `(car` x`)` expects the value of x to be a list, and returns its first element.

```
> (car '(a b c))
a
```

5. `(cdr` x`)` expects the value of x to be a list, and returns everything after the first element.

```
> (cdr '(a b c))
(b c)
```

6. `(cons` x y`)` expects the value of y to be a list, and returns a list containing the value of x followed by the elements of the value of y.

```
> (cons 'a '(b c))
(a b c)
> (cons 'a (cons 'b (cons 'c '())))
(a b c)
> (car (cons 'a '(b c)))
a
> (cdr (cons 'a '(b c)))
(b c)
```

7. `(cond` $(p_1\ e_1) \ldots (p_n\ e_n)$`)` is evaluated as follows. The p expressions are evaluated in order until one returns `t`. When one is found, the value of the corresponding e expression is returned as the value of the whole `cond` expression.

```
> (cond ((eq 'a 'b) 'first)
        ((atom 'a)  'second))
second
```

In five of our seven primitive operators, the arguments are always evaluated when an expression beginning with that operator is evaluated.[2] We will call an operator of that type a *function*.

2 Denoting Functions

Next we define a notation for describing functions. A function is expressed as `(lambda` $(p_1 \ldots p_n)$ e`)`, where $p_1 \ldots p_n$ are atoms (called *parameters*) and e is an expression. An expression whose first element is such an expression

`((lambda` $(p_1 \ldots p_n)$ e`)` $a_1 \ldots a_n$`)`

is called a *function call* and its value is computed as follows. Each expression a_i is evaluated. Then e is evaluated. During the evaluation of e, the value of any occurrence of one of the p_i is the value of the corresponding a_i in the most recent function call.

```
> ((lambda (x) (cons x '(b))) 'a)
(a b)
> ((lambda (x y) (cons x (cdr y)))
   'z
   '(a b c))
(z b c)
```

If an expression has as its first element an atom f that is not one of the primitive operators

`(`f $a_1 \ldots a_n$`)`

and the value of f is a function `(lambda` $(p_1 \ldots p_n)$ e`)` then the value of the expression is the value of

`((lambda` $(p_1 \ldots p_n)$ e`)` $a_1 \ldots a_n$`)`

In other words, parameters can be used as operators in expressions as well as arguments:

```
> ((lambda (f) (f '(b c)))
   '(lambda (x) (cons 'a x)))
(a b c)
```

There is another notation for functions that enables the function to refer to itself, thereby giving us a convenient way to define recursive functions.[3] The

[2] Expressions beginning with the other two operators, `quote` and `cond`, are evaluated differently. When a `quote` expression is evaluated, its argument is not evaluated, but is simply returned as the value of the whole `quote` expression. And in a valid `cond` expression, only an L-shaped path of subexpressions will be evaluated.

[3] Logically we don't need to define a new notation for this. We could define recursive functions in our existing notation using a function on functions called the Y combinator. It may be that McCarthy did not know about the Y combinator when he wrote his paper; in any case, `label` notation is more readable.

notation

(label f (lambda ($p_1 \ldots p_n$) e))

denotes a function that behaves like (lambda ($p_1 \ldots p_n$) e), with the additional property that an occurrence of f within e will evaluate to the label expression, as if f were a parameter of the function.

Suppose we want to define a function (subst x y z), which takes an expression x, an atom y, and a list z, and returns a list like z but with each instance of y (at any depth of nesting) in z replaced by x.

```
> (subst 'm 'b '(a b (a b c) d))
(a m (a m c) d)
```

We can denote this function as

```
(label subst (lambda (x y z)
               (cond ((atom z)
                      (cond ((eq z y) x)
                            ('t z)))
                     ('t (cons (subst x y (car z))
                               (subst x y (cdr z))))))))
```

We will abbreviate f = (label f (lambda ($p_1 \ldots p_n$) e)) as

(defun f ($p_1 \ldots p_n$) e)

so

```
(defun subst (x y z)
  (cond ((atom z)
         (cond ((eq z y) x)
               ('t z)))
        ('t (cons (subst x y (car z))
                  (subst x y (cdr z)))))))
```

Incidentally, we see here how to get a default clause in a cond expression. A clause whose first element is 't will always succeed. So

(cond (x y) ('t z))

is equivalent to what we might write in a language with syntax as

if x then y else z

3 Some Functions

Now that we have a way of expressing functions, we define some new ones in terms of our seven primitive operators. First it will be convenient to introduce some abbreviations for common patterns. We will use cxr, where x is a sequence of as or ds, as an abbreviation for the corresponding composition of car and cdr. So for example (cadr e) is an abbreviation for (car (cdr e)), which returns the second element of e.

```
> (cadr '((a b) (c d) e))
(c d)
> (caddr '((a b) (c d) e))
e
> (cdar '((a b) (c d) e))
(b)
```

Also, we will use (list $e_1 \ldots e_n$) for (cons e_1 ... (cons e_n '()) ...).

```
> (cons 'a (cons 'b (cons 'c '())))
(a b c)
> (list 'a 'b 'c)
(a b c)
```

Now we define some new functions. I've changed the names of these functions by adding periods at the end. This distinguishes primitive functions from those defined in terms of them, and also avoids clashes with existing Common Lisp functions.

1. (null. x) tests whether its argument is the empty list.

```
(defun null. (x)
  (eq x '()))

> (null. 'a)
()
> (null. '())
t
```

2. (and. x y) returns t if both its arguments do and () otherwise.

```
(defun and. (x y)
  (cond (x (cond (y 't) ('t '())))
        ('t '())))

> (and. (atom 'a) (eq 'a 'a))
t
> (and. (atom 'a) (eq 'a 'b))
()
```

3. (not. x) returns t if its argument returns (), and () if its argument returns t.

```
(defun not. (x)
  (cond (x '())
        ('t 't)))

> (not (eq 'a 'a))
()
> (not (eq 'a 'b))
t
```

4. (append. x y) takes two lists and returns their concatenation.

```
(defun append. (x y)
  (cond ((null. x) y)
        ('t (cons (car x) (append. (cdr x) y)))))

> (append. '(a b) '(c d))
(a b c d)
> (append. '() '(c d))
(c d)
```

5. (pair. x y) takes two lists of the same length and returns a list of two-element lists containing successive pairs of an element from each.

```
(defun pair. (x y)
  (cond ((and. (null. x) (null. y)) '())
        ((and. (not. (atom x)) (not. (atom y)))
         (cons (list (car x) (car y))
               (pair. (cdr x) (cdr y))))))

> (pair. '(x y z) '(a b c))
((x a) (y b) (z c))
```

6. (assoc. x y) takes an atom x and a list y of the form created by `pair.`, and returns the second element of the first list in y whose first element is x.

```
(defun assoc. (x y)
  (cond ((eq (caar y) x) (cadar y))
        ('t (assoc. x (cdr y)))))

> (assoc. 'x '((x a) (y b)))
a
> (assoc. 'x '((x new) (x a) (y b)))
new
```

4 The Surprise

So we can define functions that concatenate lists, substitute one expression for another, etc. An elegant notation, perhaps, but so what? Now comes the surprise. We can also, it turns out, write a function that acts as an interpreter for our language: a function that takes as an argument any Lisp expression, and returns its value. Here it is:

```
(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom)  (atom   (eval. (cadr e) a)))
       ((eq (car e) 'eq)    (eq     (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'car)   (car    (eval. (cadr e) a)))
       ((eq (car e) 'cdr)   (cdr    (eval. (cadr e) a)))
       ((eq (car e) 'cons)  (cons   (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'cond)  (evcon. (cdr e) a))
       ('t (eval. (cons (assoc. (car e) a)
                        (cdr e))
                  a))))
    ((eq (caar e) 'label)
     (eval. (cons (caddar e) (cdr e))
            (cons (list (cadar e) (car e)) a)))
    ((eq (caar e) 'lambda)
     (eval. (caddar e)
            (append. (pair. (cadar e) (evlis. (cdr e) a))
                     a)))))

(defun evcon. (c a)
  (cond ((eval. (caar c) a)
         (eval. (cadar c) a))
        ('t (evcon. (cdr c) a))))

(defun evlis. (m a)
  (cond ((null. m) '())
        ('t (cons (eval.  (car m) a)
                  (evlis. (cdr m) a)))))
```

The definition of `eval.` is longer than any of the others we've seen before. Let's consider how each part works.

The function takes two arguments: `e`, the expression to be evaluated, and `a`, a list representing the values that atoms have been given by appearing as parameters in function calls. This list is called the *environment*, and it is of the form created by `pair.`. It was in order to build and search these lists that we wrote `pair.` and `assoc.`.

The spine of `eval.` is a `cond` expression with four clauses. How we evaluate an expression depends on what kind it is. The first clause handles atoms. If `e` is an atom, we look up its value in the environment:

```
> (eval. 'x '((x a) (y b)))
a
```

The second clause of `eval.` is another `cond` for handling expressions of the form `(`*a* `...)`, where *a* is an atom. These include all the uses of the primitive operators, and there is a clause for each one.

```
> (eval. '(eq 'a 'a) '())
t
> (eval. '(cons x '(b c))
         '((x a) (y b)))
(a b c)
```

All of these (except `quote`) call `eval.` to find the value of the arguments.

The last two clauses are more complicated. To evaluate a `cond` expression we call a subsidiary function called `evcon.`, which works its way through the clauses recursively, looking for one in which the first element returns `t`. When it finds such a clause it returns the value of the second element.

```
> (eval. '(cond ((atom x) 'atom)
                ('t 'list))
         '((x '(a b))))
list
```

The final part of the second clause of `eval.` handles calls to functions that have been passed as parameters. It works by replacing the atom with its value (which ought to be a `lambda` or `label` expression) and evaluating the resulting expression. So

```
(eval. '(f '(b c))
       '((f (lambda (x) (cons 'a x)))))
```

turns into

```
(eval. '((lambda (x) (cons 'a x)) '(b c))
       '((f (lambda (x) (cons 'a x)))))
```

which returns `(a b c)`.

The last two clauses in `eval.` handle function calls in which the first element is an actual `lambda` or `label` expression. A `label` expression is evaluated by pushing a list of the function name and the function itself onto the environment, and then calling `eval.` on an expression with the inner `lambda` expression substituted for the `label` expression. That is,

```
(eval. '((label firstatom (lambda (x)
                           (cond ((atom x) x)
                                 ('t (firstatom (car x))))))
         y)
       '((y ((a b) (c d)))))
```

becomes

```
(eval. '((lambda (x)
           (cond ((atom x) x)
                 ('t (firstatom (car x)))))
         y)
       '((firstatom
          (label firstatom (lambda (x)
                            (cond ((atom x) x)
                                  ('t (firstatom (car x)))))))
         (y ((a b) (c d)))))
```

which eventually returns `a`.

Finally, an expression of the form `((lambda (`$p_1 \ldots p_n$`) `e`) `$a_1 \ldots a_n$`)` is evaluated by first calling `evlis.` to get a list of values `(`v_1 `...` v_n`)` of the arguments $a_1 \ldots a_n$, and then evaluating e with `(`p_1 v_1`)` ... `(`p_n v_n`)` appended to the front of the environment. So

```
(eval. '((lambda (x y) (cons x (cdr y)))
         'a
         '(b c d))
       '())
```

becomes

```
(eval. '(cons x (cdr y))
       '((x a) (y (b c d))))
```

which eventually returns `(a c d)`.

5 Aftermath

Now that we understand how `eval` works, let's step back and consider what it means. What we have here is a remarkably elegant model of computation. Using just `quote`, `atom`, `eq`, `car`, `cdr`, `cons`, and `cond`, we can define a function, `eval.`, that actually implements our language, and then using that we can define any additional function we want.

There were already models of computation, of course—most notably the Turing Machine. But Turing Machine programs are not very edifying to read. If you want a language for describing algorithms, you might want something more abstract, and that was one of McCarthy's aims in defining Lisp.

The language he defined in 1960 was missing a lot. It has no side-effects, no sequential execution (which is useful only with side effects anyway), no practical numbers,[4] and dynamic scope. But these limitations can be remedied with surprisingly little additional code. Steele and Sussman show how to do it in a famous paper called "The Art of the Interpreter."[5]

If you understand McCarthy's `eval`, you understand more than just a stage in the history of languages. These ideas are still the semantic core of Lisp today. So studying McCarthy's original paper shows us, in a sense, what Lisp really is. It's not something that McCarthy designed so much as something he discovered. It's not intrinsically a language for AI or for rapid prototyping, or any other task at that level. It's what you get (or one thing you get) when you try to axiomatize computation.

Over time, the median language, meaning the language used by the median programmer, has grown consistently closer to Lisp. So by understanding `eval` you're understanding what will probably be the main model of computation well into the future.

[4] It is possible to do arithmetic in McCarthy's 1960 Lisp by using e.g. a list of n atoms to represent the number n.

[5] Guy Lewis Steele, Jr. and Gerald Jay Sussman, "The Art of the Interpreter, or the Modularity Complex (Parts Zero, One, and Two)," MIT AI Lab Memo 453, May 1978.

Notes

In translating McCarthy's notation into running code I tried to change as little as possible. I was tempted to make the code easier to read, but I wanted to keep the flavor of the original.

In McCarthy's paper, falsity is represented by `f`, not the empty list. I used `()` to represent falsity so that the examples would work in Common Lisp. The code nowhere depends on falsity happening also to be the empty list; nothing is ever consed onto the result returned by a predicate.

I skipped building lists out of dotted pairs, because you don't need them to understand `eval`. I also skipped mentioning `apply`, though it was `apply` (a very early form of it, whose main purpose was to quote arguments) that McCarthy called the universal function in 1960; `eval` was then just a subroutine that `apply` called to do all the work.

I defined `list` and the `c`*x*`r`s as abbreviations because that's how McCarthy did it. In fact the `c`*x*`r`s could all have been defined as ordinary functions. So could `list` if we modified `eval`, as we easily could, to let functions take any number of arguments.

McCarthy's paper only had five primitive operators. He used `cond` and `quote` but may have thought of them as part of his metalanguage. He likewise didn't define the logical operators `and` and `not`, but this is less of a problem because adequate versions can be defined as functions.

In the definition of `eval.` we called other functions like `pair.` and `assoc.`, but any call to one of the functions we defined in terms of the primitive operators could be replaced by a call to `eval.`. That is,

```
(assoc. (car e) a)
```

could have been written as

```
(eval. '((label assoc.
                (lambda (x y)
                  (cond ((eq (caar y) x) (cadar y))
                        ('t (assoc. x (cdr y))))))
         (car e)
         a)
        (cons (list 'e e) (cons (list 'a a) a)))
```

There was a small bug in McCarthy's `eval`. Line 16 was (equivalent to) `(evlis. (cdr e) a)` instead of just `(cdr e)`, which caused the arguments in a call to a named function to be evaluated twice. This suggests that this description of `eval` had not yet been implemented in IBM 704 machine language when the paper was submitted. It also shows how hard it is to be sure of the correctness of any length of program without trying to run it.

I encountered one other problem in McCarthy's code. After giving the definition of `eval` he goes on to give some examples of higher-order functions—functions that take other functions as arguments. He defines `maplist`:

```
(label maplist
       (lambda (x f)
         (cond ((null x) '())
               ('t (cons (f x) (maplist (cdr x) f))))))
```

then uses it to write a simple function `diff` for symbolic differentiation. But `diff` passes `maplist` a function that uses `x` as a parameter, and the reference to it is captured by the parameter `x` within `maplist`.[6]

It's an eloquent testimony to the dangers of dynamic scope that even the very first example of higher-order Lisp functions was broken because of it. It may be that McCarthy was not fully aware of the implications of dynamic scope in 1960. Dynamic scope remained in Lisp implementations for a surprisingly long time—until Sussman and Steele developed Scheme in 1975. Lexical scope does not complicate the definition of `eval` very much, but it may make compilers harder to write.

[6]Present day Lisp programmers would use `mapcar` instead of `maplist` here. This example does clear up one mystery: why `maplist` is in Common Lisp at all. It was the original mapping function, and `mapcar` a later addition.