CSCI/ARTI 4540/6540 First Lecture on Symbolic Programming and LISP

Michael A. Covington[1]
The University of Georgia
August 23, 2010

Symbolic programming

Symbolic programming dates back to an insight that John McCarthy and his group had around 1958, which is expressed in their paper, "Recursive Functions of Symbolic Expressions," published in *Communications of the ACM,* 1960:

(1) People manipulate numbers

(2) People write formulas that say how to manipulate numbers

(3) People manipulate formulas (as in any algebra or calculus course)

(4) ?

The missing fourth item is of course "People write formulas that say how to manipulate formulas." That is the key idea McCarthy introduced – that we can define functions of expressions, not just functions of numbers or strings or bits or other data types.

Putting this another way, symbolic programming is programming in which **the program can treat itself, or material like itself, as data** – programs can write programs (not just as character strings or texts, but as the actual data structures that the program is made of).

Generally, in symbolic programming languages, **complex data structures are easy to create – you can usually create them by just writing them, and then extend them by performing simple operations.** If you've ever created a linked list in C, you know it takes a lot of function calls to do so. Not so in Lisp or Prolog.

In fact, the appeal of symbolic programming for AI is mostly the ease of creating and manipulating complex data structures (lists, trees, and other representations of search spaces).

---

[1] I want to thank Jennifer Rouan for giving me a copy of her notes from this lecture as given on August 19. This written version contains slightly more information.

Symbolic vs. functional vs. logic programming

I've told you what **symbolic** programming is.

Functional programming is programming by defining functions (e.g., "the recprocal of X is 1 divided by X") rather than by telling the computer what to do (load 1 into the accumulator, divide by X, store the result). A functional program usually has lots of short function definitions rather than long stretches of sequential code.

LISP is both a symbolic and a functional language.

Logic programming is programming controlled by a model of logical reasoning (e.g., "I can conclude that Y is the reciprocal of X if I can do such-and-such…").

Prolog, developed by Colmerauer around 1974, is both a logic programming language and a symbolic programming language.

LISP

McCarthy's original language, LISP, has a name that stands for "List Processor" (not "Lots of Individual Single Parentheses" as some people joke). Its key idea is that we write expressions as lists. Instead of **2+3** we write **(+ 2 3)**, which is a 3-element list with the symbol **+** as its first element and the numbers 2 and 3 as its other elements.

The reason for using lists is that they're easy to treat as data also.

A key idea of LISP (and also Prolog) is that **you don't evaluate an expression unless it's time to do so.** In most programming languages, anywhere you write 2+3 you can be sure that the number 5 will result, so you might as well have written 5 in the first place. In LISP, when you write **(+ 2 3)** you may be going to evaluate it and get 5, or you may be going to do something else with it, such as rearrange the elements of the list, or use them for some other purpose.

Let's evaluate some expressions…

I will write ➔ to mean "evaluates to." For example, when I write

(+ 2 3) ➔ 5

I mean "when you evaluate **(+ 2 3)** you get **5**.

Here are some examples of arithmetic in Lisp:

(+ 2 3) ➔ 5

(+ 2 1) ➔ 3
(+ 1 2 3 4) ➔ 10 the + function will add up all the numbers you give it
(+ 2) ➔ 2 not terribly useful, but consistent

(- 5 2) ➔ 3 subtraction
(- 5 1 2) ➔ 2 subtract all the numbers from the first one
(- 4) ➔ -4 find the opposite of a number
(- -4) ➔ 4

(* 10 32) ➔ 320 multiplication

(+ 2.4 5) ➔ 7.4 floating-point numbers

(/ 10 5) ➔ 2 division
(/ 2 3) ➔ 2/3

Here, **2/3** is called a **rational number** and is stored as a pair of integers. It gives you an exact representation of 2/3 with nothing lost to rounding:

(* 3 2/3) ➔ 2

You can make a rational into a floating-point number with **float** or by doing floating-point operations on it:

(float 2/3) ➔ 0.66667
(* 1.0 2/3) ➔ 0.66667

Incidentally, integers are also stored with as many digits as necessary (not just the bits of the machine's **int** type):

(* 12341234123412341234 1234123412341234) ➔
152306059688877163314808572566427 56

There are lots of arithmetic functions, such as the square root:

(sqrt 2) ➔ 1.414

Recursion

Recursion is what you have when **a computation or a data structure contains something of the same kind as itself.** If the recursion is finite, then the inner one will be **smaller** than the outer one.

An amusing graphical example of recursion is the University of Georgia arch. As you know, the 3-columned arch is on the Seal of the State of Georgia. In the top of the arch is supposed to be a medallion of – you guessed it – the Seal of the State of Georgia. And in the top of the arch depicted there…

That is endless recursion. Normally we deal with finite recursion.

A simple example of finite recursion is **expressions within expressions**. To evaluate the expression

(+ (+ 2 3) (+ 4 5))

the computer first evaluates the smaller expressions **(+ 2 3)** and **(+ 4 5)** and puts their values in place of them. Last, it evaluates **(+ 5 9)** and gets the result.

LISP evaluation rules

To describe LISP, all we need to do is describe how to evaluate all different kinds of expressions. Here's what we know so far:

Type of symbolic expression (s-expression)	*Examples*	*How evaluated*
Numbers:		
Integers Rationals Floating-point	23 -345 2/3 45.6778	The value of a number it itself. (Evaluation produces no change.)
Symbols	+ X Y SQRT	Usually cannot be evaluated. (We'll learn about them…)
Lists	(+ 2 3) (+ (- 10 5) (+ 5 6))	The first element of the list must be the name of a function. Evaluate the subsequent elements, and then, using their values as arguments, perform the function.

Take a moment to convince yourself that these rules describe everything we've seen so far.

The quote

You can prevent an expression from being evaluated. This is done by putting a single quotation mark (') in front of it.

You do not need a closing quotation mark. The quote applies to the whole symbolic expression right after it. (Not the rest of the line; just the expression. For example, if you write '(+ 2 3) (+ 4 5) only (+ 2 3) is quoted.)

'(+ 2 3) ➔ (+ 2 3)

You can also write the quote as **(QUOTE ...)**, like this:

(QUOTE (+ 2 3)) ➔ (+ 2 3)

In fact, that is how it is represented internally; the shorter notation with ' is translated into the longer notation with **QUOTE** by the LISP compiler.

The value of a list that starts with QUOTE is simply the expression following QUOTE. This expression is not evaluated. Thus:

(QUOTE (GO DOGS)) ➔ (GO DOGS)

and it doesn't matter that **(GO DOGS)** cannot be evaluated. Further,

(QUOTE (QUOTE (GO DOGS))) ➔ (QUOTE (GO DOGS))

because QUOTE prevents everything from being evaluated, including more QUOTEs.

QUOTE is not a function

Note that according to our evaluation rules, the arguments of a function are evaluated before the function is performed. It followed that QUOTE is not a function. QUOTE is what we call a **special form**, and we need to amend the evaluation rules as follows:

Lists	(+ 2 3) (+ (- 10 5) (+ 5 6))	If the first element of the list is the name of a function: Evaluate the subsequent elements, and then, using their values as arguments, perform the function.
	(QUOTE BLAH)	If it is the name of a special form: Do whatever the special form is defined to do.

In general, if you have something that looks like a function, but does not evaluate all its arguments before doing anything else, it is not a function but a special form.

We will use only a few special forms in this course. One of them is IF, which decides which arguments to evaluate.

EVAL

The opposite of QUOTE is EVAL. EVAL is a function, so its argument is evaluated. Then EVAL evaluates the result. For example:

(EVAL (QUOTE (+ 2 3))) ➔ 5

What happens is:
EVAL is a function, so its argument, (QUOTE (+ 2 3)), gets evaluated.
The value of (QUOTE (+ 2 3)) is (+ 2 3).
EVAL receives that and evaluates it, giving 5.

The reason we have EVAL is so that we can **construct expressions by computation and then evaluate them**. This is excessively hard or impossible in other programming languages – behold the power of symbolic programming.

Arguably, to implement LISP, all you have to do is implement EVAL. That, of course, is a big job.

Backquote

Lisp has another special form called backquote, written ` , which works like the quote except that you can use commas to designate parts of the expression for evaluation. Thus:

`(THE ANSWER IS ,(+ 2 3) WE THINK) ➔ (THE ANSWER IS 5 WE THINK)

We will not use backquote in this course. It comes in very handy when you start defining your own special forms.