Programming Languages (CS302 2007S)

Paul Graham - The Roots of LISP

Comments on:

Graham, Paul. (2002). The Roots of Lisp. Web essay at `http://lib.store.yahoo.net/lib/paulgraham/jmc.ps` (dated 18 January 20 02; visited 1 February 2006).

Graham states that "by understanding `eval` you're understanding what will probably be the main model of computation well into the future". If this statement holds, to what extent has the idea behind the `eval` statement been incorporated into more mordern or in Graham's view "median" languages?

I'd prefer that you try to answer this than to have me try to answer it.

Paul Graham argues that "there have been two really clean, consistent models of programming so far: the C model and the Lisp model" and that "have been moving steadily toward the Lisp model."

Is there some truth to this analysis under the current theory of what a "good" programming language is? Even if that is true, Graham's argument seems to rest on the assumption that there isn't a problem that both models have. It is much more likely that any programming language will be imperfect than that it will be perfect. Isn't putting any language, be it Lisp or C, on such a high pedestal run the risk of blinding its supporters to the language's flaws?

There is no single theory on what makes a "good" programming language. And it depends a bit on who puts the language on the pedestal. I find C and Lisp/Scheme cleaner (as in having a simpler, clearer model) than most languages.

What you have said in class on Fri. covered almost every point in Paul Graham's paper. However, in his paper, he also talks about some disadvantages of this kind of computation notation or theory. One thing pointed out by Graham makes me to reread or really carefully read the "eval" function. That is, he said, "understanding eval is more than just a stage in the history of languages", and I will also understand the how the "ideas" come up and how to use it in the future.

I'm not sure that I covered everything on Friday, but I agree that `eval` is an incredibly important idea.

I understand that using just quote, atom, eq, car, cdr, cons, and cond, we can define a function, eval., that implements the language, then using that we can define any additional functions we want- clearly an advantage because of its simplicity. However, Graham says that there were already models of computation- most notably the Turing Machine. Besides the disadvantage of the Turing Machine programs not being very edifying to read, was there any other need for the development of a language like LISP?

Well, you'd never write a real program in a Turing Machine, but people clearly write real programs in Lisp. In that sense, Lisp provides a nice bridge between abstract models of computation and actual implementation. (The von Neumann model does, too.).

The one thing that came to mind was where is the association list kept in Lisp? Can it be edited with something other than the define macro? Also, when going through the assoc list, how are duplicate associations handled? It seems as though assoc. will return the first occurence, but wouldn't we want the last, or most recent, one?

The most recent one is at the front, given how we update the association list. Different implementations of Lisp give you different access to that list (or, more generally, the environment).

It looks as though The Roots of Lisp by Paul Graham is a direct translation from Mcarthy's complicated M-notation to Lisp code.

Well, as Graham notes, he had to clean up at least one error.

On Pg.5, Graham shows two ways to define subst. My first question: Is "label" a keyword in Common LISP? If not, I can see his point on how to convert/abbreviate the notational concept of "label" to lisp code using the keyword "defun".

I'm not sure if `label` *was retained in Common Lisp, but it was ceratinly part of the original Lisp, as you should know from the McCarthy paper.*

I noticed that in Scheme the notation is slightly different. i.e Instead of

```
(define foo
   (lambda (a b c)
      ...
```

we can write,

```
(define (foo a b c) ...
```

Are they different in some way?

There is no real difference between the two. We teach the former rather than the latter because it makes it easier to transition to anonymous functions and because it's a bit clearer for variable-arity functions.

- It seems like "defun" defined on page 5 is unnecessary. Why did he choose to show it?

Because it's more familiar to most Lisp programmers than label + lambda.

In the same definition "defun", Graham uses "'t" as an else statement. Is that similar to how Scheme interprets else in a conditional expression?

Yes, and that should be familiar from the McCarthy reading.

When Graham introduces the concept of recursion in lisp, he introduces the notation: (label f (lambda (p1..pn) e)). Doesn't this implicate implicit goto statements? i.e., when an occurrence of f is found in e, goto f using the results of the last f evaluation as its parameters.

Well, as you can tell from `eval`*, the call of* `f` *does not involve a goto. In fact, since we can use f in a non-tail-recursive form, it can't be a goto.*