Programming Languages (CS302 2007S)

John McCarthy - History of LISP

Comments on:

McCarthy, J. (1978). History of LISP. *ACM SIGPLAN Notices* **13**(8), pp. 217-223. DOI= http://doi.acm.org/10.1145/960118.808387

When discussing about conditional expressions (in pg. 218), McCarthy says that XIF(M,N1,N2) had to be used sparingly since all three arguments had to be evluated even if M was either zero or not. He says that it led to the invention of the true conditional expression which evaluated only one of N1 and N2 depending on whether M was true of false. What is the difference between representing the value of M as either 0 or 1, or True and False? How are True and False implemented in LISP? C does not have explicit True and False in which case we define True and False to be 0 and 1 if we ever need to use it.

It's mostly an order-of-evaluation issue. For a traditional procedure, you evaluate all the arguments before applying the procedure. In the revised version, you only evaluate some of the arguments.

I was wondering whether or not data structures were prevalent during the time Lisp was developed. It is evident why he makes lists the main data structure (functions themselves are simply lists) but why did he not include a way to make a struct or a similar data type?

No, generic mechanisms for making compound data types had not really been invented at the time Lisp was first developed.

Were macros added after this paper was written? He doesn't seem to mentioned them much, but the other readings seem to emphasize them a lot.

Macros were not part of the original Lisp. I'm not sure when they were added, but they were a central part of Common Lisp.

Also, what exactly is pornographic programming?

I'll get back to you on that.

When McCarthy discusses the creation of a "LISP environment" does he mean creating the interpreter for the language or is it something else? Assuming that is what he means, what reasons were there for LISP to be an interpreted language instead of a compiled language? McCarthy states that the concern was that producing the compiler would take too long, but were there any other advantages or disadvantages for favoring an interpreted language over a compiled language?

Yes, he generally means the interpreter. The big advantage is that it encourages experimentation - type a little, see the result, type a little more, see the next result, and so on and so forth. There's also a bit of encouragement in such a system to write small procedures, which many of us think are good.

McCarthy says that a large number of computer scientists and programmers regard the fact that LISP programs are lists to be a disadvantage. Why is that so? And why does he go on to say that this same feature of the language may have contributed to its survival?

Let me get back to you on that.

The writer of the History of Lisp says that one could conjecture that LISP owes its survival specifically to the fact its programs are lists. He says that everyone including him have regarded as a disadvantage. If it is a disadvantage, why isn't it changed especially considering its so central to Lisp?

Because it's also an advantage, and he's exaggerating. As the previous writer mentions, McCarthy also says that lists were key to Lisp's survival.

McCarthy states, "The first succesful LISP compiler was programmed by Timothy Hart and Michael Levin. It was written in LISP and was claimed to be the first compiler written in the language to be compiled" How is it possible to write a compiler in the same language as the language to be compiled?

There are a number of ways, we'll actually cover this issue on Friday. In terms of the interpreter, seven key operations (plus or minus) are implemented in assembler. Everything else can be implemented with those key operations. In addition, if there's an interpreter, then you can interpret the compiler and have that compile itself to get an executable compiler.

LISP puts the operator first in a mathematical expression because "any other notation necessarily requires special programming". When an operation is translated into machine code, why would it require special programming? It seems like it would be a very simple change. Is it there for other reasons besides making the programming easier?

It's easier to parse things in which the operator comes first. When you see the operator, you know its arity, you know the expected types, you know almost everything.

It's also there because it makes the form of s-expressions a bit simpler, but I suppose that's incorporated in the previous answer.

What are the possible downsides to building a programming language that revolves around lists?

You need to do garbage collection, you need to chase pointers (which means that things that are related may not be nearby in memory), you sometimes succumb to "If the only tool you have is a hammer, everything looks like a nail.".

When the paper compares the LISP with Turing Machine, it doesn't provide any example. Generally, in this paper, I am not familiar with many things such as IBM 704, SDS.... It's hard for students, who are from current programming world, to understand this paper.

Well, if understanding the thing seems central to the paper, then you probably need to a bit of extra research. However, knowing the details of the computer on which LISP ran is probably not essential. What kind of example would you like in the comparison of Lisp with a Turing machine? Church's theorem shows them to be equivalent. You must also realize the audience for whom he was writing.

Note that one of the things I want you to gain from this course is a better understanding of history, and, because I respect you as intelligent learners, I have no intention of spoon-feeding you that history.

John McCarthy credits his work with AI as very influential in his creation of LISP. He explains that he chose to make a list-based language because of his work with AI . This brought to mind one of our past readings about how software engineering has influenced the design of programming languages. It seemed to me that McCarthy's article also shows how the available hardware influences the design of a programming language (e.g car and cdr).

*It is natural to make a language reflect the hardware, at least a bit, particularly if you care about efficient code.*a

The 15 bit sections for address ond decrement make sense. What are the prefix and tag sections of the word for?

I'll get back to you that.