Programming Languages (CS302 2007S)

John McCarthy - Recursive Functions of Symbolic Expressions

Comments on:

McCarthy, John. (1960). Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM* **3**(4), pp. 184-195.
DOI=http://doi.acm.org/10.1145/367177.367199

Very Good or Excellent Responses and Questions

The garbage collection algorithm mentioned in the paper seems rather inefficient in time critical applications and other situations. Even outside of time critical applications, using large amounts of memory that is not strictly needed for the program seems wasteful if those resources could be used on something else, perhaps another process. Is this the method of garbage collection still commonly used, or even used at all, or has it be replaced by different methods?

Acceptable Responses and Questions

In regards to linear LISP McCarthy said, "However, if the functions are to be represented by computer routines, LISP is essentially faster". Can you explain that?

This comment appears at the end of section 5. Remember that McCarthy is looking at two, somewhat related, issues. He is considering models of computation and he is considering an implementation of those models. I also expect that there are two aspects to his claim - First, he is noting that the indirection of "implement the plain Lisp functions and then do everything else with those" is going to be slower. Second, it is likely that linear LISP is more complicated to implement.

Why the extinction of M-expressions?

McCarthy Dealty with that issue a bit in the previous reading - most programmers found S-expressions clearer. The lack of ambiguity was also a big win (see, for example, the implies question below).

About circular expressions: "Such an expression could not exist in a world with our topology. [They] could have advantages in the machine..." If there are advantages, why not allow them? Is McCarthy referring to the branch in mathematics (topology)? If so is there a theory/concept that self-referential structures violate?

In the traditional concept of a list, when you step through the elements, you eventually reach the end and you never go back to a previous element. Circular lists lack both characteristics. And he refering to topology primarily primarily as a way of thinking geometrically. As to why they can't exist, see below.

Recursive Functions of Symbolic Expresssions and Their Computation by Machine was interesting in that it breaks down stuff that we already knew, like car and cdr. However, I am still unclear as to the difference between S-expressions and what we've come to know as a procedure. For example, in Phoenix we used the (define-movie) macro and, from that experience, I gathered that (define-syntax) was used to expand the language. However, I'm now unsure how that differs from us writing a map procedure without using (define-syntax).

Well, there are a lot of things here. The article doesn't really cover macros, but I'm happy to respond to that question. Macros deal with the source code*. Procedures like* `map` *deal with the translated version. And that's why you needed a macro for* `define-movie` *- you needed the name that was given to the movie, not just the value associated with that name.*

S-expressions are the source code of procedures and the values we see in Scheme (remember, the two are indistinguishable.

According to the reading a form can be converted into a function if we can determine the correspondance between the variables occuring in the form and the ordered list of arguments of the desired function. He says this is accomplished by Church's lambda notation. He gives us an example of such as y^2 + x, what happens for cases where the correspondance between the variables and the ordered list is more difficult to find. Do we just not have any forms in that case?

It should not be difficult to make a list of the variables in a form. Any order then suffices.

Short Notational Questions

Traditionally, these kinds of questions should be separate from your main questions or comments. If they are the only thing you submit, your grade will be a minus.

I take it then that what I've been reading is the less dense notation for Scheme then. so far one expression has thrown me off. Within the sqrt function I don't understand the significance of E->x and as a result the whole expression is kind of confusing. I don't really think that's the sort of thing that looking up will help me with.

I assume that you're reading the example on the bottom of the left column of p. 205. There is no "E->x" in that definition. There is, however, a conditional expression.

```
(|x^2-a| < epsilon -> x, T -> sqrt(a, 1/2(x+a/x), epsilon))
```

Remember, conditional expressions are a series of "test -> result" pairs that are separated by commas.

You can therefore read the read this as "If the the absolute value of the quantity x squared minus a end quantity is less than epsilon, then x, otherwise, if true then recurse using ...." Why is atom[(X.A)]=F? In other words, why isn't (X.A) an atomic symbol? The paper says (A.B) is an atomic symbol.

Okay, I'm game, where does the paper say that (A.B) is an atomic symbol?

(X.A) is a pair of two elements, X and A. Pairs are not atoms.

On page 208, McCarthy describes eval[e;a] as having two arguments, "and expression e to be evaluated, and a list of pairs a". He further says that the first item of a is an atomic symbol and the second an expression that a stands for. This convention is not followed in the expression "eval[((LABEL, FF, w), (QUOTE, (A.B))); NIL]" (i.e. the first step of the evaluation of "apply" demonstrated in the next page).

Um ... how does the example fail to follow the model?

The expression e: `((LABEL, FF, w ), (QUOTE, (A.B)))`

The list of pairs: `NIL`

In a previous reading, Paul Graham wrote about how the eval statement was used as a LISP interpreter. I am still not very sure about how eval works. Can you explain the eval statement in class?

McCarthy explains it a bit on pp. 208-209, but I'll certainly go over it.

On p211 McCarthy says the circular list structures have difficulties in printing and in certain other operations. What are these difficulties?

Well, the equal procedure on p. 207 would loop forever. The traditional print procedure would also loop forever. Here's one version of that procedure that only works for shallow lists, written in tradiitional Scheme.

```
(define printlst
  (lambda (lst)
     (if (null? lst)
         (display "()")
         (begin
           (display "(")
           (display (car lst))
           (let kernel ((remaining (cdr lst)))
             (if (null? remaining)
                 (display ")")
                 (begin
                   (display ", ")
                   (display (car remaining))
                   (kernel (cdr remaining)))))))))
```