Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

JOHN MCCARTHY, *Massachusetts Institute of Technology, Cambridge, Mass.*

presented by Ryan Doenges
January 29th, 2019

Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I of 1

JOHN MCCARTHY, *Massachusetts Institute of Technology, Cambridge, Mass.*

presented by Ryan Doenges
January 29th, 2019

Plan

- Historical context
- S-expressions, S-functions, M-expressions
- `eval`
- Legacy of the paper
- **Discussion!**

Computing in 1960

- Computers like the IBM 704 use vacuum tubes, reels of magnetic tape, punch cards, and cost a lot of money

- "the field variously called artificial intelligence, heuristic programming, automata theory, etc."

- Algorithms defined in flowcharts (Section 6)

The IBM 704

The IBM 704

PL in 1960

- FORTRAN turns 7

- ALGOL 60 standardized by Backus, Naur, Perlis, McCarthy, *et al.*

- Lambda calculus approaching 30 years of age, considered a subject of purely mathematical interest

S-expressions

Abstract syntax:

```
atom ::= A | B | C | ... | AA | AB | ...
s-exp  ::= Atom atom
         |  Cons s-exp s-exp
```

Let's use modern syntactic sugar instead of the paper's notation. Some examples:

```
()     = NIL
(A)    = (Cons (Atom A) NIL)
(A B) = (Cons (Atom A) (Cons (Atom B) NIL))
```

M-expressions

Meta-expressions are the host language or metalanguage and are already equipped with an evaluator, unlike S-expressions.

McCarthy writes `f[x; y]` for M-expression function calls; let's just write `(f x y)` and distinguish S-expressions by quoting them `'(like this)`. More on this in a minute.

Primitive S-functions

- Predicates: `atom`, `eq`
- Constructor: `cons`
- Projections: `car`, `cdr`, `caar`, `cadr`, ...
  - `car` is the first thing in the cons cell
  - c**dr** is **d**a **r**est

Conditional expressions

There are distinguished atoms T and F which serve as truth values.

McCarthy writes

```
[p₁ → e₁; p₂ → e₂; ...]
```

for conditionals, but let's write

```
(cond (p₁ e₁) (p₂ e₂) ...).
```

The predicates p_i are evaluated in order, and if p_i evaluates to T then the conditional short-circuits to e_i.

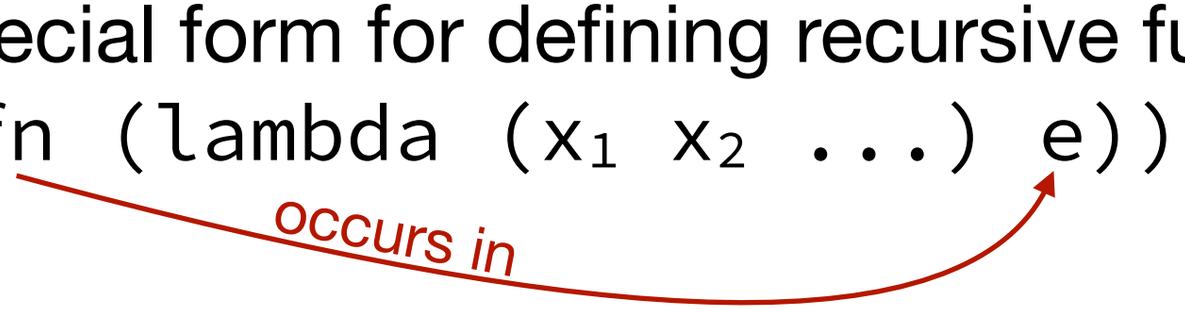
Lambdas and recursion

Our friend the anonymous function:

```
(lambda (x1 x2 ...) e)
```

There's also a special form for defining recursive functions:

```
(label fn (lambda (x1 x2 ...) e))
```



Is `label` necessary? Why or why not?

Quotation

The quotation operator **(page 189, left)** takes an M-expression and produces an S-expression which represents it.

```
'x                          := X
'(f x y ...)                := ('f 'x 'y ...)
'(cond (p₁ e₁) ...) := (COND ('p₁ 'e₁)
'(lambda (x...) e) := (LAMBDA ('x...) 'e)
```

apply **and** eval

apply

The function `apply` takes an S-expression representing a function and then a list of arguments.

```
(apply f args) := (eval (f (appq args)))
```

where `appq` quotes each element of the list `args`.

eval

The function `eval` takes a quoted s-expression along with an environment (an association list) and evaluates it. Full definitions are on **page 189**.

There's some fishy stuff going on here. For example, **what will this evaluate to?**

```
((lambda (x)
   ((lambda (g x) (g nil))
    (lambda (y) x)
    2)
1)
```

Ron Garcia
@rg9119

Gul Agha to John McCarthy: Why did LISP have dynamic scope?

McCarthy to Agha: To be honest, I didn't understand lambda calculus.
(AGERE17)

1:46 PM · Oct 26, 2017 · TweetDeck

6. Finally, the evaluation of ((LAMBDA, (x_1 , $\cdots$, x_n), &), e_1 , $\cdots$, e_n) is accomplished by evaluating & with the list of pairs ((x_1 , e_1), $\cdots$, ((x_n , e_n)) put on the front of the previous list *a*.

The list *a* could be eliminated, and LAMBDA and LABEL expressions evaluated by substituting the arguments for the variables in the expressions &. Unfortunately, difficulties involving collisions of bound variables arise, but they are avoided by using the list *a*.

(page 190, left column)

Legacy

The first functional programming language, even if it got scope wrong

S-expressions were necessary for the development of rich macro systems and fancy metaprogramming features present in modern Lisps (Racket/Scheme, Clojure, ...)

Automatic memory management

Computer algebra and other forms of "symbolic computing"

Domain-specific langauges **(page 191)**

Some discussion questions

Were there any concepts or techniques in the paper that felt modern? Any that felt strange or dated?

Was LISP a scripting language?

How would you describe McCarthy's approach to semantics?

What became of M-expressions?

This notation is writable and somewhat readable. It can be made easier to read and write at the cost of making its structure less regular. If more characters were available on the computer, it could be improved considerably.

(page 189, left column)