SOME LISP HISTORY AND SOME PROGRAMMING LANGUAGE IDEAS

John McCarthy, Stanford University

Berlin, 2002 June 19

• I suppose I'm here because of Lisp, although I haven't been involved in the Lisp community for a very long time.

• Lisp is actively used, e.g. on the Deep Space 1 spacecraft and in the Orbitz airline reservation system, but I don't know any details.

• Fritz Kunze's Franz Inc. makes quite a good common Lisp, called Allegro Common Lisp.

• Some important aspects of Lisp are not available in other programming languages and systems. I don't know if they are used in the above applications.

• The original idea was to combine 1956 list processing as done by Newell, Simon and Shaw with ideas from John Backus's Fortran.

• Herbert Gelernter at IBM undertook to implement Marvin Minsky's idea for a plane geometry theorem prover, and I proposed list processing in Fortran. Gelernter and Carl Gerberich developed FLPL.

• In 1958 Lisp was started at M.I.T. using recursion, which was not feasible in Fortran. Lisp was intended for AI programming.

• Lisp was intended to be compiled at first. However, I wrote a universal Lisp function $eval$ in 1959 to show that Lisp was a neater language for computability theory than Turing machines. Steve Russell pointed out that the universal function could be taken as an interpreter for pure Lisp, and hand-compiled it in IBM 704 machine language.

DIFFERENTIATION—the motivating example

The following example motivated

- recursion using conditional expressions
- lisp notation for algebraic expressions
- allowing functions as arguments with λ-expression syntax

diff$(e, v) \leftarrow$ if at e then
[if $e = v$ then 1 else 0]
else if car e $= PLUS$ then $PLUS$. $maplist(cdr\ e, \lambda u.diff(car\ u, v))$
else if car e $= TIMES$ then $PLUS$. $maplist(cdr\ e, \lambda u.TIMES$.
$maplist(cdr\ e, \lambda w.$ if u eq w then $diff(car\ u, v)$ else $car\ w)))$

ASPECTS OF LISP

• Lisp lists including lists of list are the appropriate representation of symbolic expressions for computation—better than strings—and better than XML.

• Lisp programs are Lisp data. Put abstractly, Lisp has functions for its own *abstract syntax.*

• Lisp programs, most conveniently pure Lisp functional programs, are are described extensionally by first order sentences.

• Many important properties of the functions can be proved by first order reasoning.

• Other important properties require *derived functions.*

EXAMPLES OF LISP FUNCTIONAL PROGRAMS

•

```
(defun append (u v)
(if
(null u)
v
(cons (car u) (append (cdr u) v))))
```

• $u \ * \ v \leftarrow \mathbf{if\ n}\ u\ \mathbf{then}\ v\ \mathbf{else\ a}\ u\ .\ [\mathbf{d}\ u\ *v]$ is a pure LISP functional program.

• $(\forall u\ v)(u\ *'\ v = \mathbf{if\ n}\ u\ \mathbf{then}\ v\ \mathbf{else\ a}\ u\ .\ [\mathbf{d}\ u\ *'\ v])$ is an equation for the function computed by the program. The close correspondence is very convenient but sometimes confusing.

• The pure Lisp functional program as an equation permits convenient proofs in a first order theory that Lisp programs meet their specifications. For example, it is easy to prove by list induction that

$\forall u\ v.(u * v) * w = u * (v * w))$, i.e. that appending lists is an associative operation.

LISP AND OTHER LANGUAGES

• Garbage collection, conditional expressions and recursive programs have been taken into other languages.

• LISP data structures have been imitated clumsily in XML.

(BUY item1 Item2 Item3)

<BUY> item1 item2 item3 </BUY>

• LISP programs having access to the abstract syntax of the program has not been imitated. This represents a lack of imagination, but I admit I don't have convincing examples of its use.

DERIVED FUNCTIONS

The computational cost of a Lisp functional recursive program is not determined by the extension of the function. We have $f91(x) \leftarrow$ if $x > 100$ then $x - 10$ else $f91(f91(x + 11))$ and $ff91(x) \leftarrow$ if $x > 100$ then $x - 10$ else 91.

We have $\forall x.f91'(x) = ff91'(x))$, but clearly the functional programs are different computationally. Suppose we are interested in how many times the $+$ operation is executed in evaluating $f91(x)$. This is given by $f91p'(x)$, where

$f91p(x) \leftarrow$ if $x > 100$ then 0 else $1 + f91p(f91(x + 11)) + f91p(x + 11)$.

ELEPHANT 2000: a programming language for the year 2010

www.formal.stanford.edu/jmc/elephant.html

• An elephant never forgets. An Elephant program can say, "A passenger has a reservation in a situation s if he has made a reservation and not cancelled it. The Elephant program need not specify a data structure to remember reservations. The compiler must provide the necessary data structures so that whether a passenger has a reservation can be determined.

$$Has(passenger, reservation, s) \equiv$$
$$(\exists s' < s) Occurs(Makes(passenger, reservation), s)$$
$$\wedge \neg(\exists s'')(s < s'' < s' \wedge Occurs(Cancel(Passenger, reservation), s'') \quad (1)$$

• An elephant is faithful one hundred percent. A reservation is a promise to let the passenger on the airplane if he has one when he shows up. One kind of Elephant output statement is a promise, and correct Elephant programs fulfill their promises.

• The Elephant language includes program statements called commitments generalizing Floyd assertions, because they can refer to the future, A correct Elephant program fulfills its commitments.

$$(\forall s > Now)(Value(X, s) > Value(X, Now)) \tag{2}$$

• Elephant i-o input output statments are speech acts, e.g. questions, requests, acceptances of proposals, answers to questions. Answers to questions should be true and responsive.

ALGOL 48

If we introduce time explicitly as distinct from the program counter, Algolic programs can be written as sets of equations. Here's an Algol 60 program for computing the product mn of two natural numbers.

$$
\begin{array}{rcl}
start: & & \\
i & := & n; \\
p & := & 0; \\
loop: & & if\ i = 0\ then\ go\ to\ done; \\
i & := & i - 1; \\
goto & & loop; \\
done: & &
\end{array}
$$

Here's what mathematicians might have written in 1948, before programming languages existed.

$$
\begin{aligned}
pc(0) &= 0; \\
i(t+1) &= \text{if } pc(t) = 0 \text{ then } n \\
&\text{else if } pc(t) = 4 \text{ then } i(t) - 1 \text{ else } i(t); \\
p(t+1) &= \text{if } pc(t) = 1 \text{ then } 0 \\
&\text{else if } pc(t) = 5 \text{ then } p(t) + m \text{ else } p(t) \\
pc(t+1) &= \text{if } pc(t) = 2 \wedge i(t) = 0 \text{ t} \\
&\text{else if } pc(t) = 5 \text{ then } 2 \text{ else } pc(t) + 1;
\end{aligned}
$$

The proof that $\exists t.(t \geq 0 \wedge pc(t) = 6 \wedge p(t) = mn)$ follows from the sentences expressing the program and the laws of arithmetic, i.e. no theory of program correctness is needed. However, the proof ideas are essentially the same as those used to prove that an algolic program terminates and that the outputs have the correct relation to the inputs. Amir Pnueli and Nissim Francez had this idea before I did, but they mistakenly abandoned it for temporal logic.