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 in the Orbitz airline reservation system, but I don’t have the 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
programming languages and systems. I don’t know if
they were used in the above applications.

• The original idea was to combine 1956 list processing
by Newell, Simon and Shaw with ideas from John Back-
tran.

• Herbert Gelernter at IBM undertook to implement Ma-
sky’s idea for a plane geometry theorem prover, and
list processing in Fortran. Gelernter and Carl Gerber
developed FLPL.

• In 1958 Lisp was started at M.I.T. using recursion,
not feasible in Fortran. Lisp was intended for AI pro-
gramming.
Lisp was intended to be compiled at first. However, a universal Lisp function `eval` in 1959 to show that Lisp is a neater language for computability theory than Turing. Steve Russell pointed out that the universal function could be taken as an interpreter for pure Lisp, and hand-compiled for an 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

\[
\text{diff}(e, v) \leftarrow \text{if at } e \text{ then } \\
[\text{if } e = v \text{ then } 1 \text{ else } 0] \\
\text{else if car } e = \text{PLUS then PLUS . maplist(cdr } e, \lambda u. \text{diff}(u, v)) \\
\text{else if car } e = \text{TIMES then PLUS . maplist(cdr } e, \lambda u.\text{ maplist(cdr } e, \lambda w. \text{if } u \text{ eq } w \text{ then diff(car } u, v) \text{ else })}
\]
ASPECTS OF LISP

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

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

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

• Many important properties of the functions can be found 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)))))

- $u \ast v \leftarrow \text{if } n \ u \ \text{then } v \ \text{else } a \ u$. $[d \ u \ \ast v]$ is a functional program.

- $(\forall u \ v)(u \ast' v = \text{if } n \ u \ \text{then } v \ \text{else } a \ u$. $[d \ u \ \ast' v]$ equation for the function computed by the program. 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
their specifications. For example, it is easy to prove
duction that

\[ \forall u \ v. (u \ast v) \ast w = u \ast (v \ast w) \], i.e. that appending
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

(BUY item1 Item2 Item3)

<BUY> item1 item2 item3 </BUY>

• LISP programs having access to the abstract syntax program has not been imitated. This represents a lack of imitation, but I admit I don’t have convincing examples.
DERIVED FUNCTIONS

The computational cost of a Lisp functional recursive
is not determined by the extension of the function. We
\[f_{91}(x) \leftarrow \begin{cases} x > 100 \text{ then } x - 10 & \text{else } f_{91}(f_{91}(x + 1))\end{cases}\]
and \[f f_{91}(x) \leftarrow \begin{cases} x > 100 \text{ then } x - 10 & \text{else } 91.\end{cases}\]

We have \(\forall x. f_{91}'(x) = f f_{91}'(x))\), but clearly the function
grams are different computationally. Suppose we are
in how many times the \(+\) operation is executed in
\(f_{91}(x)\). This is given by \(f_{91}p'(x)\), where

\[f_{91}p(x) \leftarrow \begin{cases} x > 100 \text{ then } 0 \text{ else } 1 + f_{91}p(f_{91}(x + 11))\end{cases}\]
• **An elephant never forgets.** An Elephant program “A passenger has a reservation in a situation $s$ if he has reservation and not cancelled it. The Elephant program specify a data structure to remember reservations. This must provide the necessary data structures so that passenger has a reservation can be determined.

\[
\text{Has}(\text{passenger}, \text{reservation}, s) \equiv \\
(\exists s' < s) \text{Occurs}(\text{Makes}(\text{passenger}, \text{reservation}), s) \\
\land \neg (\exists s'')(s < s'' < s' \land \text{Occurs}(\text{Cancel}(\text{Passenger}, \text{reservation}), s''))
\]
• An elephant is faithful one hundred percent. A reservation is a promise to let the passenger on the airplane if he was the passenger 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 that commit to future commitments generalizing Floyd assertions, because commitments refer to the future, A correct Elephant program fulfills its commitments.

\[(\forall s > \text{Now})(\text{Value}(X, s) > \text{Value}(X, \text{Now}))\]

• Elephant i-o input output statements are speech acts, declarations, requests, acceptances of proposals, answers to questions. Answers to questions should be true and responsive.
If we introduce time explicitly as distinct from the counter, Algolic programs can be written as sets of
Here’s an Algol 60 program for computing the product of two natural numbers.

```
start:
  i := n;
  p := 0;
  loop: if i = 0 then go to done;
        i := i - 1;
        goto loop;
  done:
```
Here’s what mathematicians might have written in 1948, before programming languages existed.

\[
\begin{align*}
    pc(0) &= 0; \\
    i(t + 1) &= \text{if } pc(t) = 4 \text{ then } i(t) - 1 \text{ else } i(t); \\
    p(t + 1) &= \text{if } pc(t) = 5 \text{ then } p(t) + m \text{ else } p(t) \\
    pc(t + 1) &= \text{if } pc(t) = 5 \text{ then } 2 \text{ else } pc(t) + 1;
\end{align*}
\]
The proof that $\exists t. (t \geq 0 \land pc(t) = 6 \land p(t) = mn)$ follows the sentences expressing the program and the laws of a system, i.e. no theory of program correctness is needed. However, the proof ideas are essentially the same as those used to show an algolic program terminates and that the outputs have a correct relation to the inputs. Amir Pnueli and Nissim Hadzis had this idea before I did, but they mistakenly abandoned it using temporal logic.