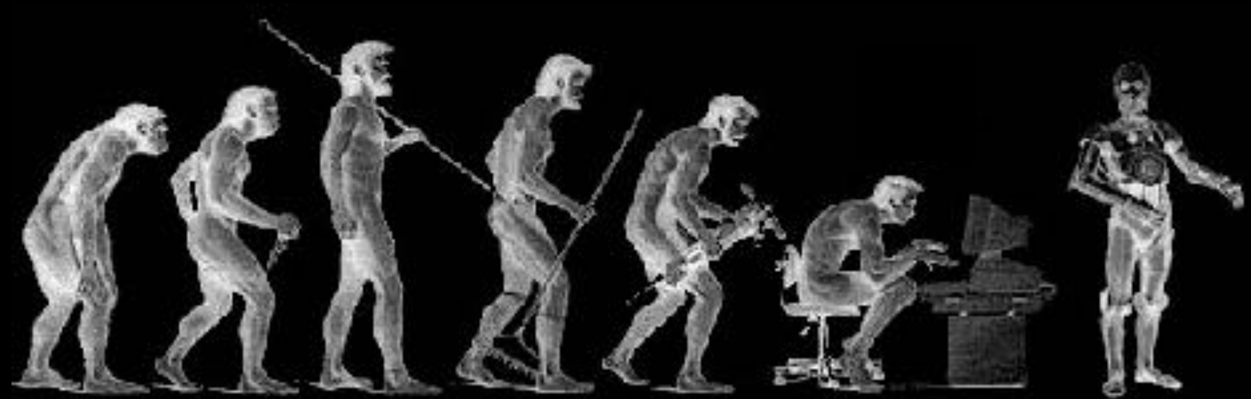
Papers We Love

$f(x) = x$

London Chapter



EARLY LISP HISTORY

1956 – 1959

A paper by Herbert Stoyan - Illustrated by Renzo Borgatti

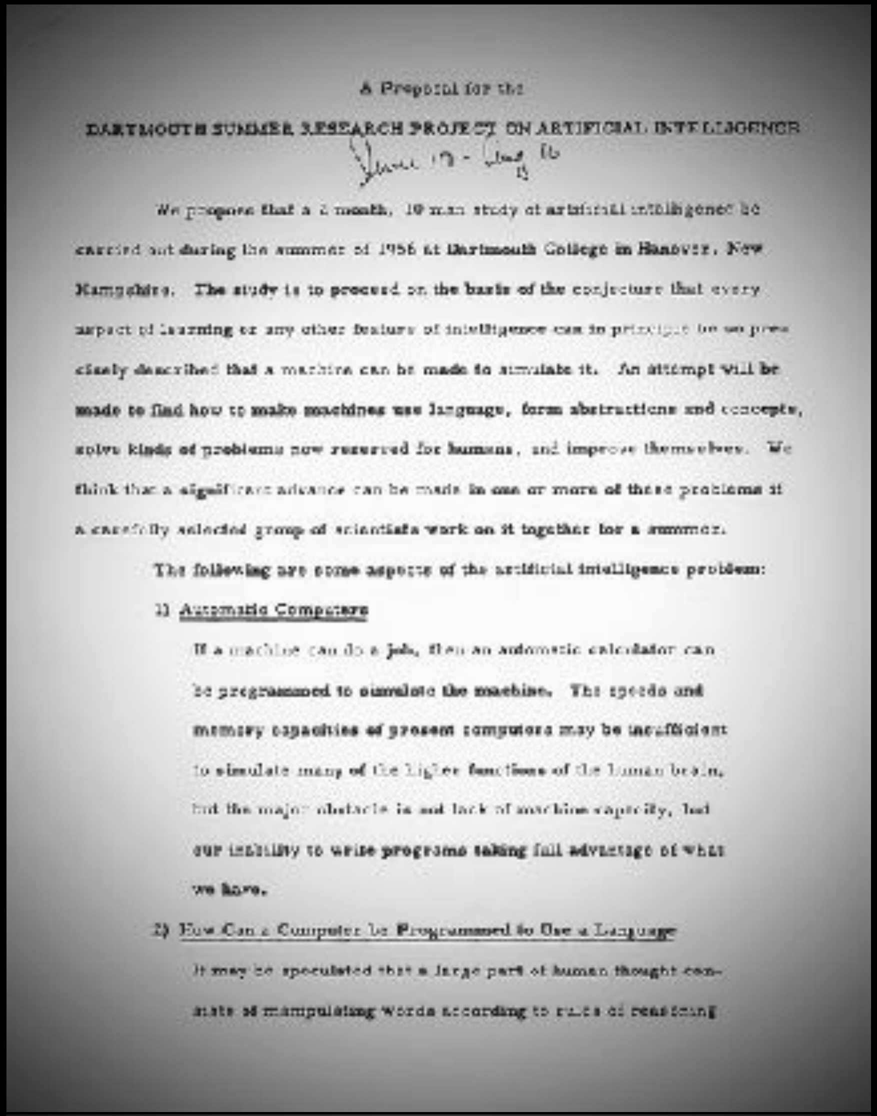
TRIAL AND ERROR

- Gradual discovery process
- From imperative to functional in baby steps
- The goal: a specific problem to solve
- The myth of the "pure" Lisp

John McCarthy (ca 1931)

TEACHING MACHINES TO LEARN

- Instruct "reasoning devices"
- Logic inference from declarative sentences
- Programming with short statements
- Relationship between intelligence and language



A Proposal for the

DARTMOUTH SUMMER RESEARCH PROJECT ON ARTIFICIAL INTELLIGENCE

We propose that a 2 month, 10 man study of artificial intelligence be carried out during the summer of 1956 at Dartmouth College in Hanover, New Hampshire. The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves. We think that a significant advance can be made in one or more of these problems if a carefully selected group of scientists work on it together for a summer.

The following are some aspects of the artificial intelligence problem:

1) Automatic Computers

If a machine can do a job, then an automatic calculator can be programmed to simulate the machine. The speeds and memory capacities of present computers may be insufficient to simulate many of the higher functions of the human brain, but the major obstacle is not lack of machine capacity, but our inability to write programs taking full advantage of what we have.

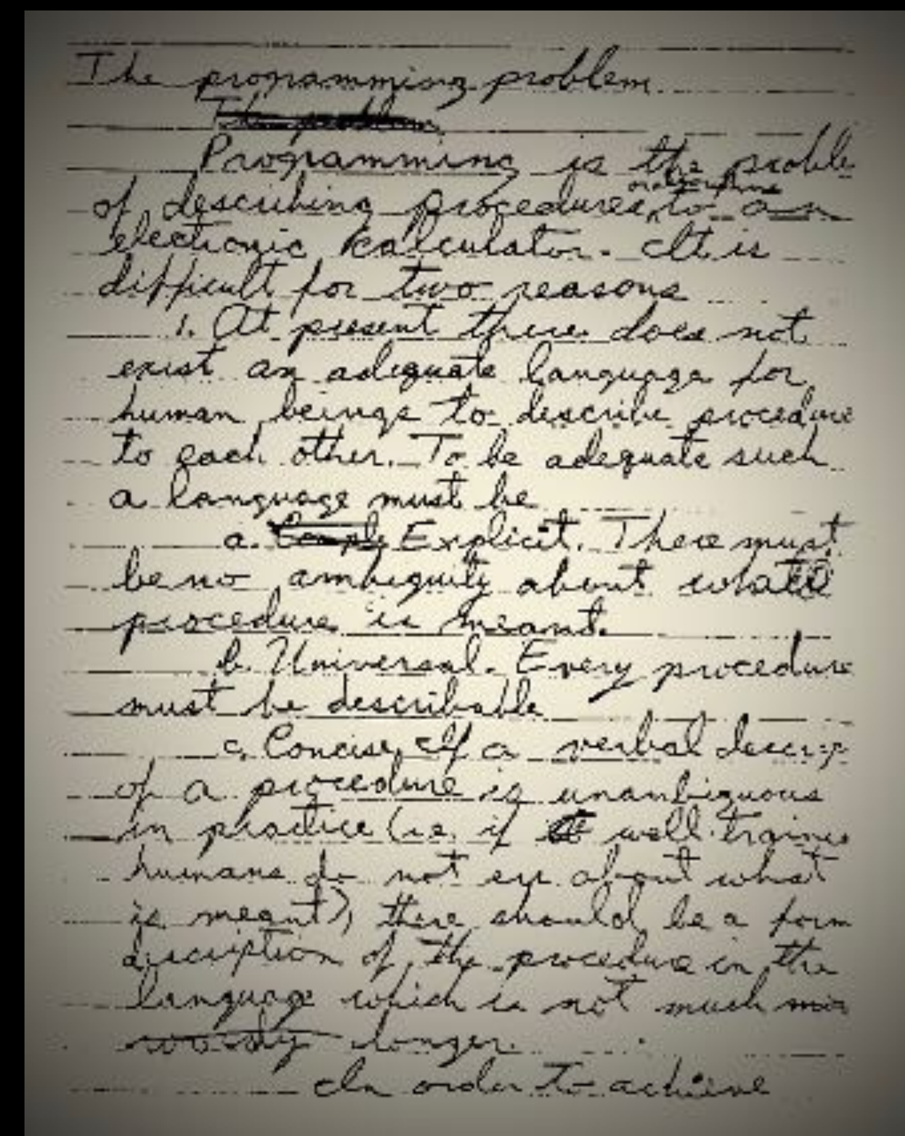
2) How Can a Computer be Programmed to Use a Language

It may be speculated that a large part of human thought consists of manipulating words according to rules of reasoning

Dartmouth Proposal 1955

THE PROGRAMMING PROBLEM

- Unambiguous
- Able to describe any computation (also recursive)
- Not much larger than the sentence it describes
- Abstracted away from hardware constraints



The programming problem

Programming is the problem of describing procedures to a machine an electronic calculator. It is difficult for two reasons

1. At present there does not exist an adequate language for human beings to describe procedures to each other. To be adequate such a language must be

a. ~~Comple~~ Explicit. There must be no ambiguity about what procedure is meant.

b. Universal. Every procedure must be describable

c. Concise. If a verbal descrip of a procedure is unambiguous in practice (i.e. if well trained humans do not argue about what is meant), there should be a form description of the procedure in the language which is not much more ~~wordy~~ longer.

In order to achieve

The programming Problem, manuscript, ~1955

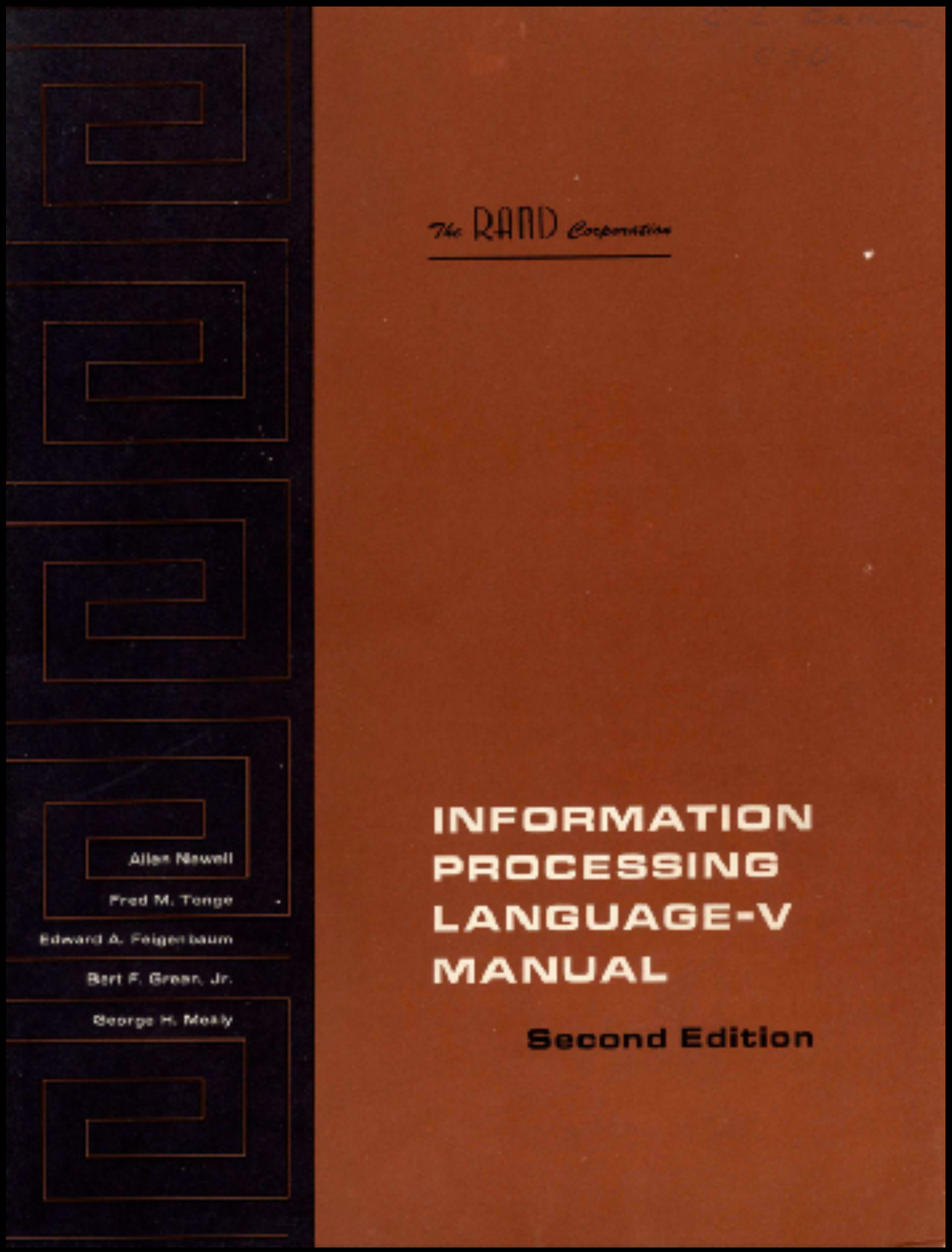
HARDWARE

PDP-1

IBM 704

IPL

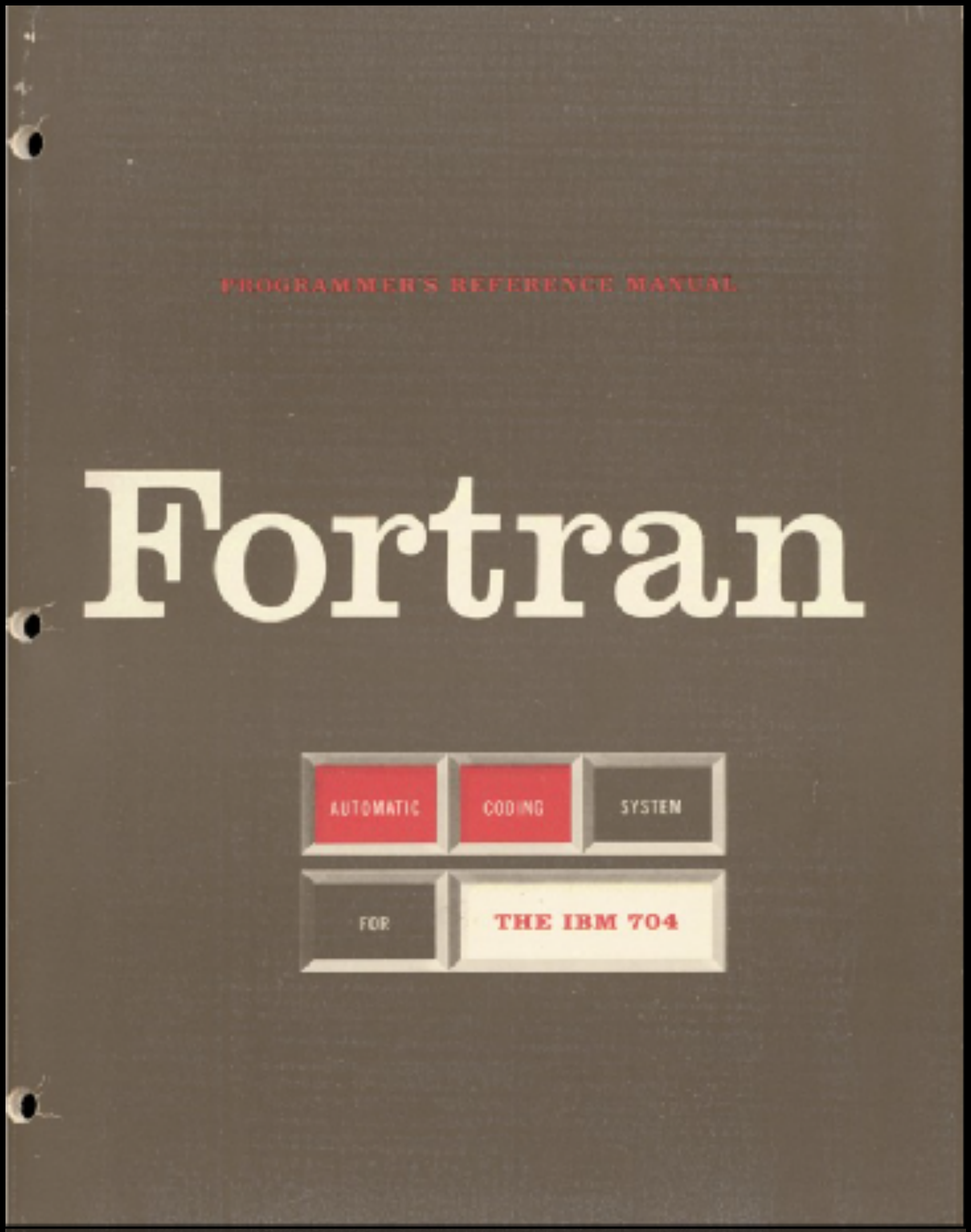
- List based assembly language
- Instructions are also lists
- Symbolic computation
- High order functions



IPL manual, 2nd ed, 1964

FORTRAN

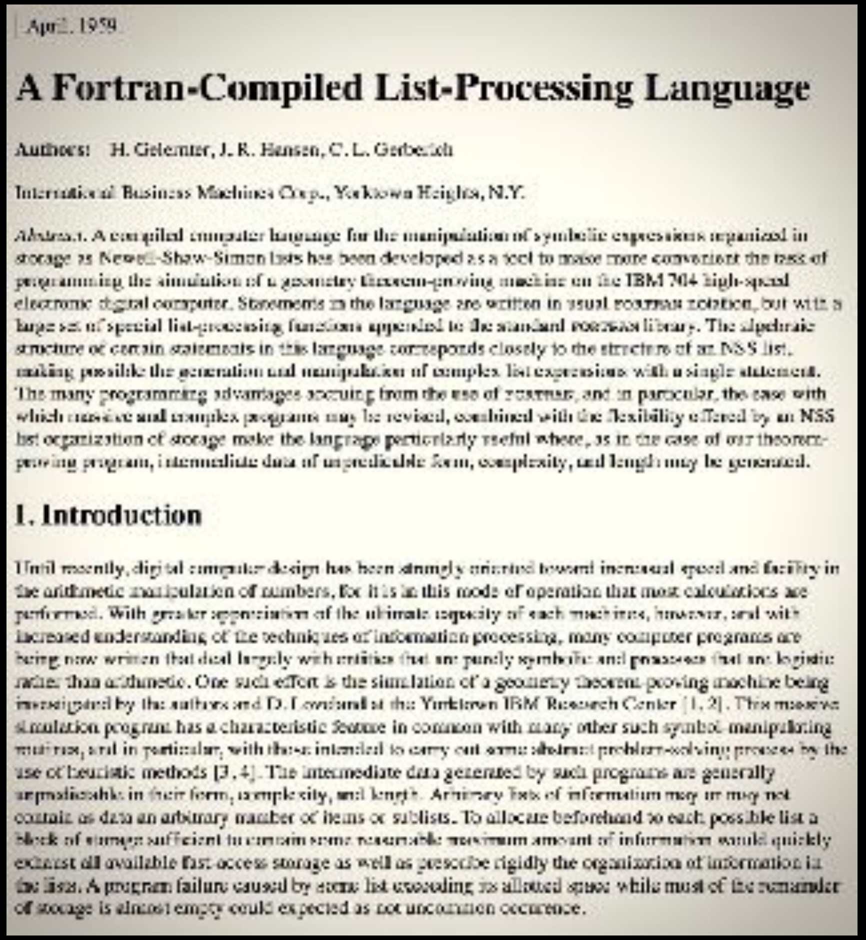
- Symbolic notation
- Arithmetic IF (expression) negative, zero, positive
- Sub-routines



Fortran coding system for the IBM 704, 1956

FLPL (FORTRAN LIST PROCESSING LANGUAGE)

- Goal: proving theorems
- Extend Fortran with lists
- Replicate IPL concepts
- Work based on the IBM 704



April, 1959

A Fortran-Compiled List-Processing Language

Authors: H. Gelernter, J. R. Hansen, C. L. Gerberich

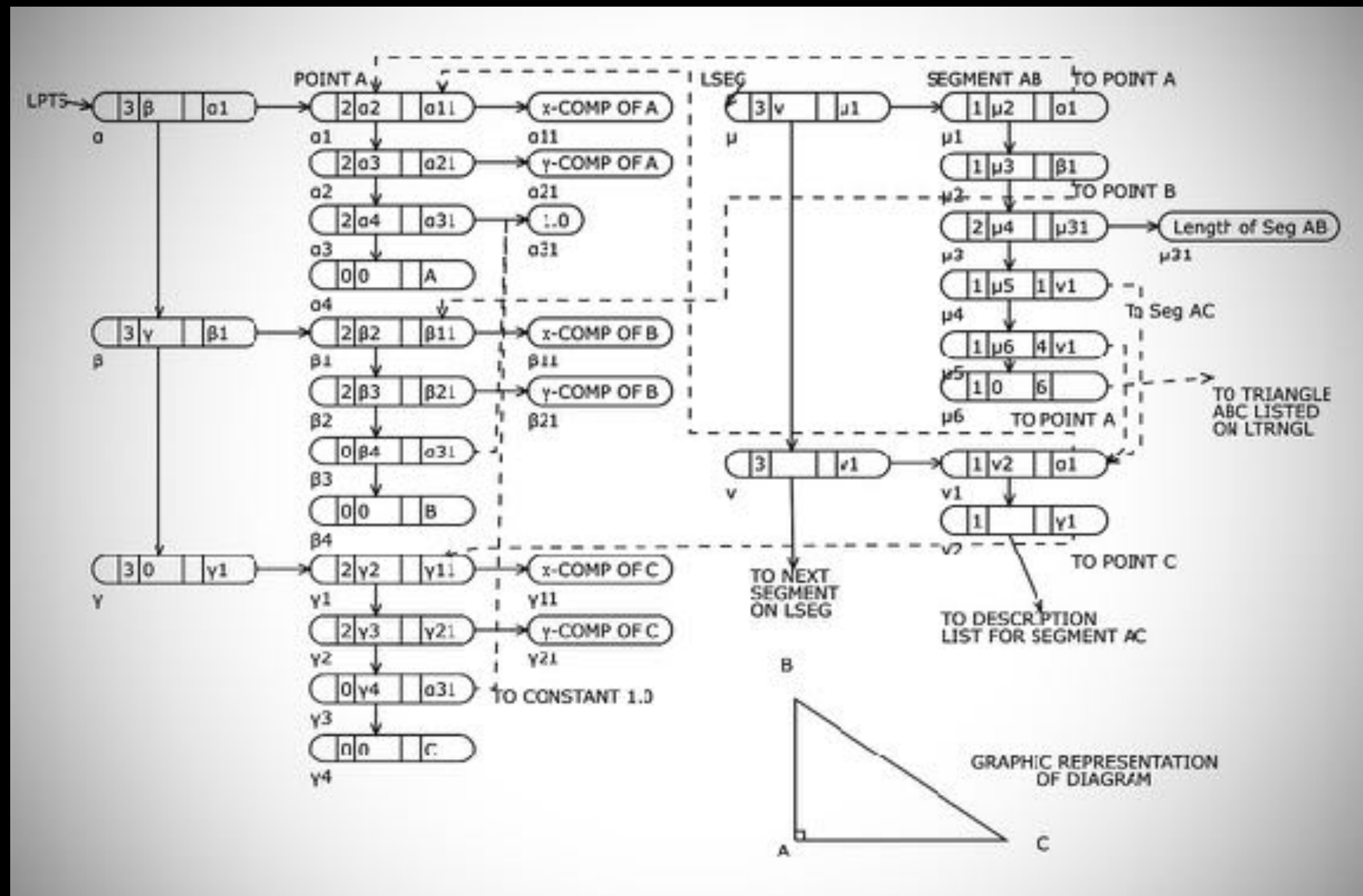
International Business Machines Corp., Yorktown Heights, N.Y.

Abstract. A compiled computer language for the manipulation of symbolic expressions organized in storage as Newell-Shaw-Simon lists has been developed as a tool to make more convenient the task of programming the simulation of a geometry theorem-proving machine on the IBM 704 high-speed electronic digital computer. Statements in the language are written in usual FORTRAN notation, but with a large set of special list-processing functions appended to the standard FORTRAN library. The algebraic structure of certain statements in this language corresponds closely to the structure of an NSS list, making possible the generation and manipulation of complex list expressions with a single statement. The many programming advantages accruing from the use of FORTRAN, and in particular, the ease with which massive and complex programs may be revised, combined with the flexibility offered by an NSS list organization of storage make the language particularly useful where, as in the case of our theorem-proving program, intermediate data of unpredictable form, complexity, and length may be generated.

1. Introduction

Until recently, digital computer design has been strongly oriented toward increased speed and facility in the arithmetic manipulation of numbers, for it is in this mode of operation that most calculations are performed. With greater appreciation of the ultimate capacity of such machines, however, and with increased understanding of the techniques of information processing, many computer programs are being now written that deal largely with entities that are purely symbolic and processes that are logistic rather than arithmetic. One such effort is the simulation of a geometry theorem-proving machine being investigated by the authors and D. Loveland at the Yorktown IBM Research Center [1, 2]. This massive simulation program has a characteristic feature in common with many other such symbol-manipulating routines, and in particular, with those intended to carry out some abstract problem-solving process by the use of heuristic methods [3, 4]. The intermediate data generated by such programs are generally unpredictable in their form, complexity, and length. Arbitrary lists of information may or may not contain as data an arbitrary number of items or sublists. To allocate beforehand to each possible list a block of storage sufficient to contain some reasonable maximum amount of information would quickly exhaust all available fast-access storage as well as prescribe rigidly the organization of information in the lists. A program failure caused by some list exceeding its allotted space while most of the remainder of storage is almost empty could expected as not uncommon occurrence.

FLPL paper, 1959



"Given a triangle in list notation, find a point with x-component greater than two."

```
INDEX = LPTS. Initialize index pointer.
   10   IF(CWWF(XCAR2F(INDEX)) - 2.0) 15, 15, 20.  Go to statement 20
               if x-component is greater than 2.0; otherwise go to 15.
   15   INDEX = XCDRF(INDEX). Move index to next ``l``-word on ``LPTS``.
   20   NAMEPT = XCARF(INDEX). Retrieve machine name of point.
        GO TO (EXIT TO ROUTINE FOR PROCESSING NAMEPT).
   25   (EXIT WHEN LPTS CONTAINS NO POINT WITH X-COMP > 2.0).
```

MAYBE NOT THE BEST IDEA

- Difficult to see lists in Fortran
- Value of a function depending on register state
- Arithmetic IF clunkiness
- Restrictions to extend the standard functions (beyond the library tape)

ALGOL FAILED COUP

- Proposal for conditional statement
- Infix notation
- Var assignment of functions
- Function composition
- Church lambda notation
- Compiler as a program to translate text according to rules
- Way too many innovative ideas!

Algol Committee reunion (1974 ACM conference on programming languages)

THE PERFECT STORM

- September 1958
- New MIT AI Lab
- 1 Basement room
- 2 programmers, 6 students
- 1 secretary, 1 typewriter

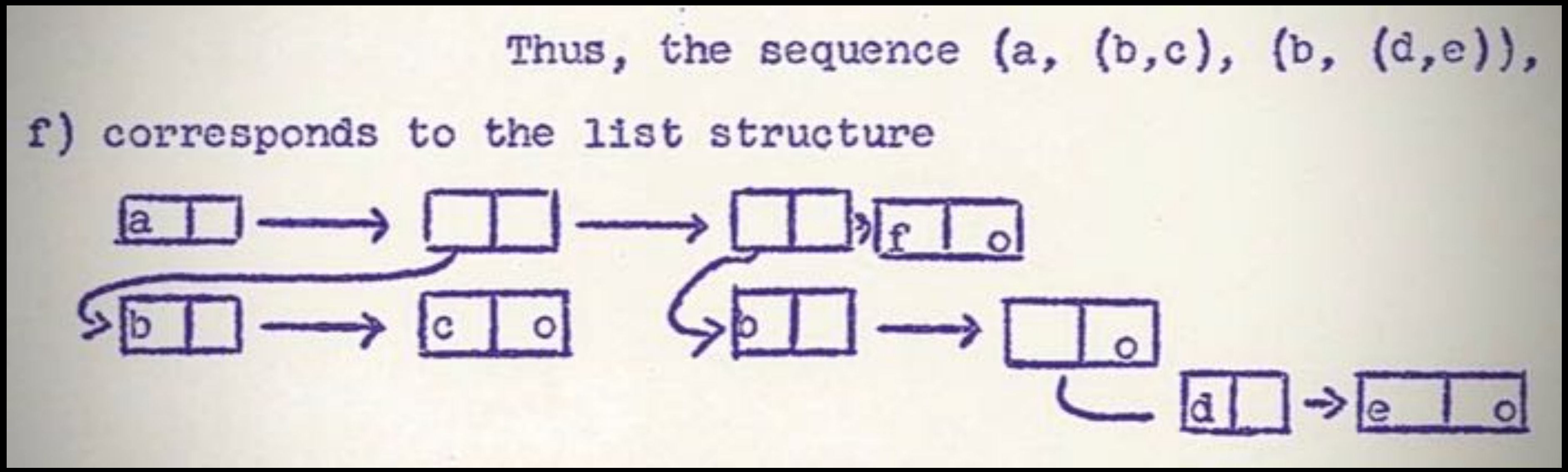
Perhaps not the same basement at MIT in 1958

- Some limited IBM 704 CPU time
- 1 Initial goal: symbolic differential equations

IBM 704 36-BITS WORD STRUCTURE

Part	Name	Bits	Fn
w	whole word	0-35	cwr
p	prefix	0-1-2	cpr
i	indicator	1-2	cir
s	sign bit	0	csr
d	decrement	3-17	cdr
t	tag	18-20	ctr
a	address	21-35	car

HOW TO STICK LISTS ON 36 BITS WORDS?


Thus, the sequence (a, (b,c), (b, (d,e)), f) corresponds to the list structure

- **consel (a,d)** puts "a" in the address and "d" in the decrement part of a word from "free storage"
- **consf1 (w,d)** takes 2 words from free storage. Puts "w" in one, puts the address of that in the address field of the other.
- **erase (j)** returns the word at location "j" back to free storage.

BUILDING ON TOP

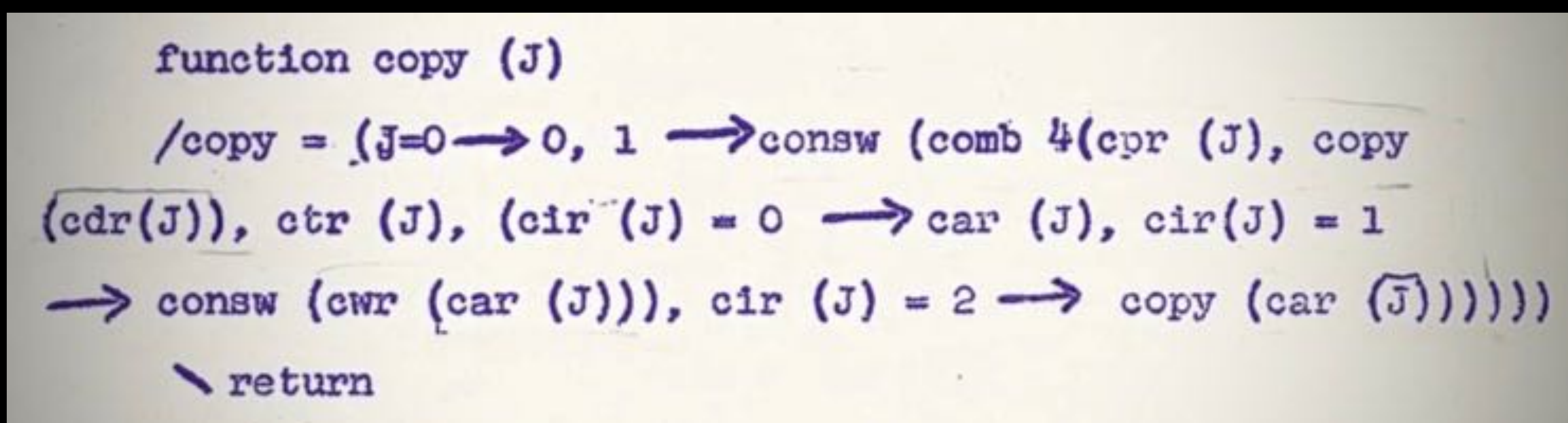
- Not exactly the Lisp you would expect…
- Strong Fortran influence
- Yet recursive
- GO (to) based conditionals

```
subroutine eralis (J)
/ J = 0 -> return
  go (a (cir(J))
a(1) jnk = erase (car (J))
a(0) eralis (dec (erase(J)))
          return
a(2) eralis (car (J))
\ go (a(o))
```

"eralis" (erase-list) walk a list to return items to free storage

COPY

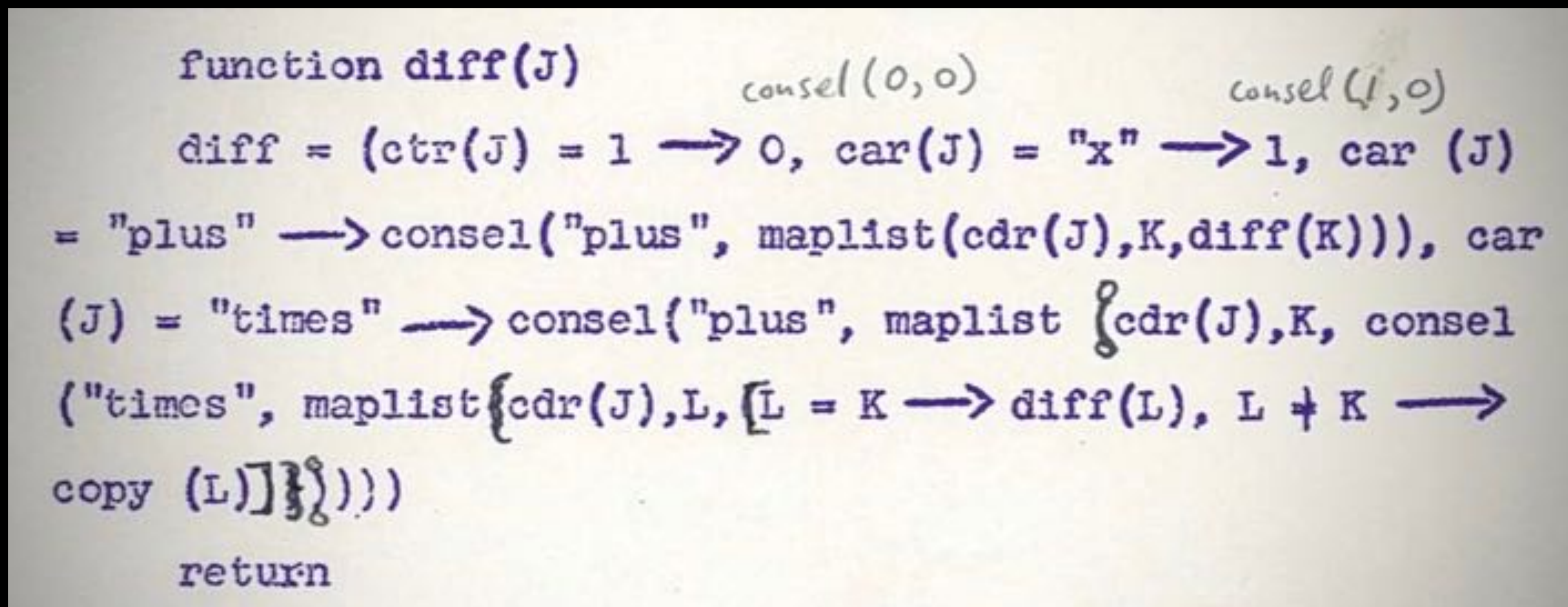
```
/copy = (J = 0 -> 0, J = 1 -> consw (comb 4 (cpr (J),
copy (cdr (J)), ctr (J), (cir (J) = 0 -> car (J), cir (J) = 1
-> consw (cwr (car (J))), cir (J) = 2 -> copy (car (J))))))
\return
```



```
    function copy (J)
    /copy = (J=0 ⟶ 0, 1 ⟶consw (comb 4(cpr (J), copy
(cdr(J)), ctr (J), (cir (J) = 0 ⟶ car (J), cir(J) = 1
⟶ consw (cwr (car (J))), cir (J) = 2 ⟶ copy (car (J))))))
     \ return
```

The copy function from the AI Memo #1

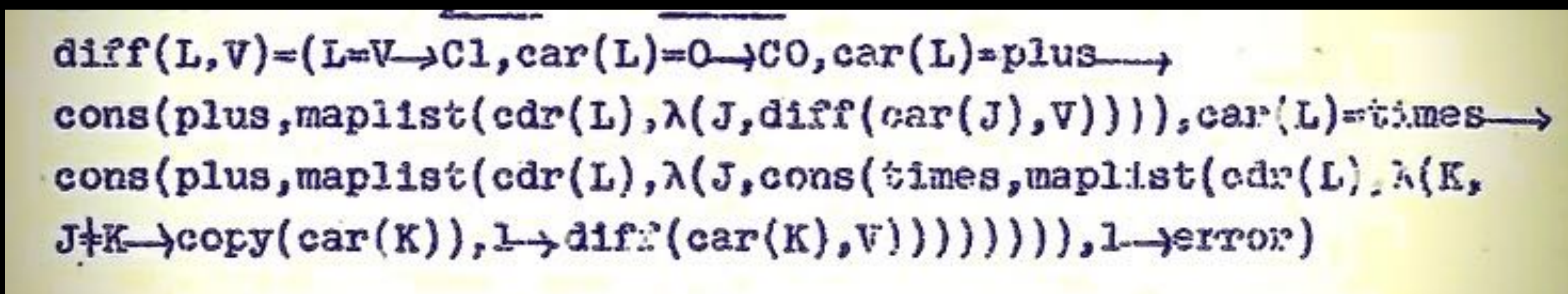
MAPLIST



```
    function diff(J)          consel(0,0)                 consel(1,0)
    diff = (ctr(J) = 1 → 0, car(J) = "x" → 1, car (J)
= "plus" → consel("plus", maplist(cdr(J),K,diff(K))), car
(J) = "times" → consel("plus", maplist {cdr(J),K, consel
("times", maplist{cdr(J),L,[L = K → diff(L), L ≠ K →
copy (L)]}})))
    return
```

- McCarthy is experimenting with syntax one-liners
- That **maplist (cdr(J),K,diff(K)))** is equivalent to **(map diff L)**
- But "K" is a dummy variable to range over the addresses
- maplist appears to mutate cdr(J) in place.

CHURCH NOTATION TO THE RESQUE



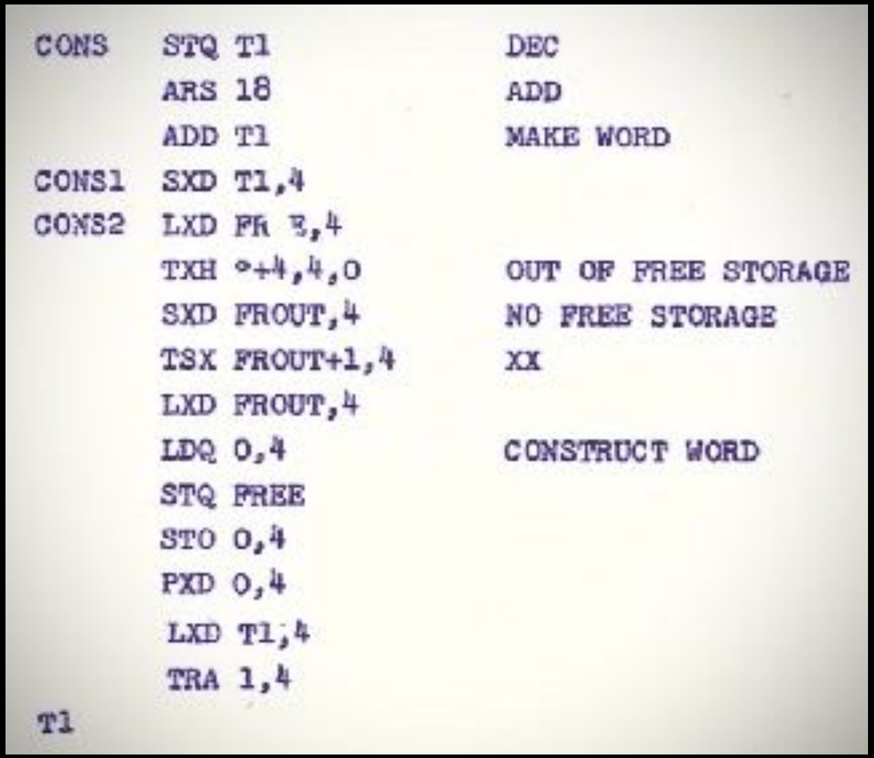
```
diff(L,V)=(L=V→C1,car(L)=0→C0,car(L)=plus→
cons(plus,maplist(cdr(L),λ(J,diff(car(J),V)))),car(L)=times→
cons(plus,maplist(cdr(L),λ(J,cons(times,maplist(cdr(L),λ(K,
J≠K→copy(car(K)),1→diff(car(K),V)))))))),1→error)
```

New diff version as appearing on the AI Memo #4

- The λ now allow maplist to declare the meaning of "J"
- Maplist returns a new list from free storage

END OF 1958 STATUS

- Several functions hand-written in assembly for the IBM 704
- Compiler envisioned to automate the process
- Described a restricted "external standard notation" to feed the compiler



```
CONS    STQ T1          DEC
        ARS 18          ADD
        ADD T1          MAKE WORD
CONS1   SXD T1,4
CONS2   LXD FR 3,4
        TXH *+4,4,0     OUT OF FREE STORAGE
        SXD FROUT,4     NO FREE STORAGE
        TSX FROUT+1,4   XX
        LXD FROUT,4
        LDQ 0,4         CONSTRUCT WORD
        STQ FREE
        STO 0,4
        PXD 0,4
        LXD T1,4
        TRA 1,4
T1
```

cons in assembly IBM 704 (called SAP)

THE EXTERNAL NOTATION (AKA LISP)

- A "colloquial" Lisp without lambdas, arrows, square brackets etc.
- Round parenthesis () because the keyboard only had those!
- Prefix notation preferred, so it could be parsed easy as a list
- "linear" fashion (no indentation or tabs)

Card punch recorder

APPLY

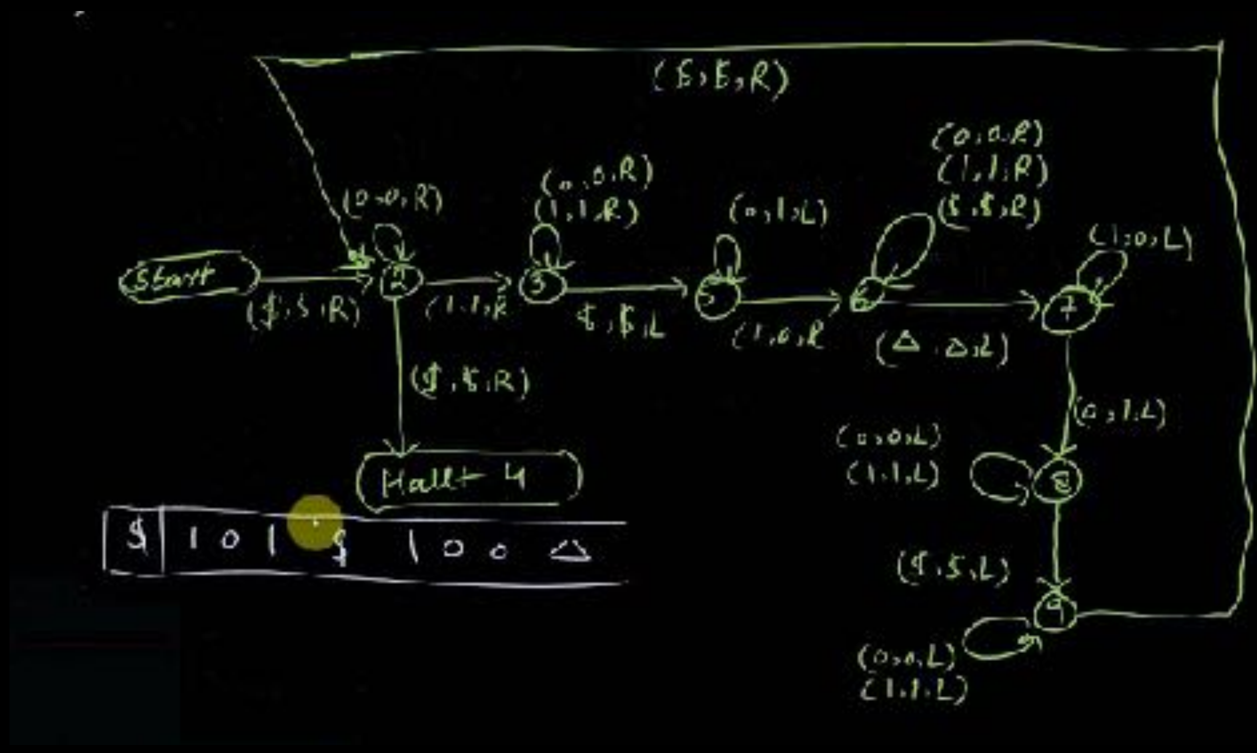
```
apply(L,f)=(car(f)=subfun→sublis(pair(car(cdr(f)),
L),car(cdr(cdr(f)))),1→error)
```

Appeared first time on the AI Memo #4

- In the meanwhile, additional ideas and extensions like **apply**
- Replace car(f) with the result of subfun
- subfun is the sublist obtained combining f with the rest of the list, (f l1, l2, .. l-N)
- Evaluate the new (f l1 l2, .., l-N)

QUOTING

- McCarthy wanted to demonstrate the universality of Lisp through "apply"
- He thought it should result in a very concise Turing machine



- Apply had to be extended to understand Lisp
- Symbols and list constants problematic
- Quoting introduced to distinguish between function calls and symbols

APPLY EVAL LOOP

- A generic apply needs to understand Lisp: symbols, lambdas, lists or other apply
- The eval part in apply was the conditional to select the correct way to interpret a form
- As part of interpreting a form, apply is used to invoke a function on its arguments
- The loop continues…

"HO, HO, YOU'RE CONFUSING THEORY WITH PRACTICE..."

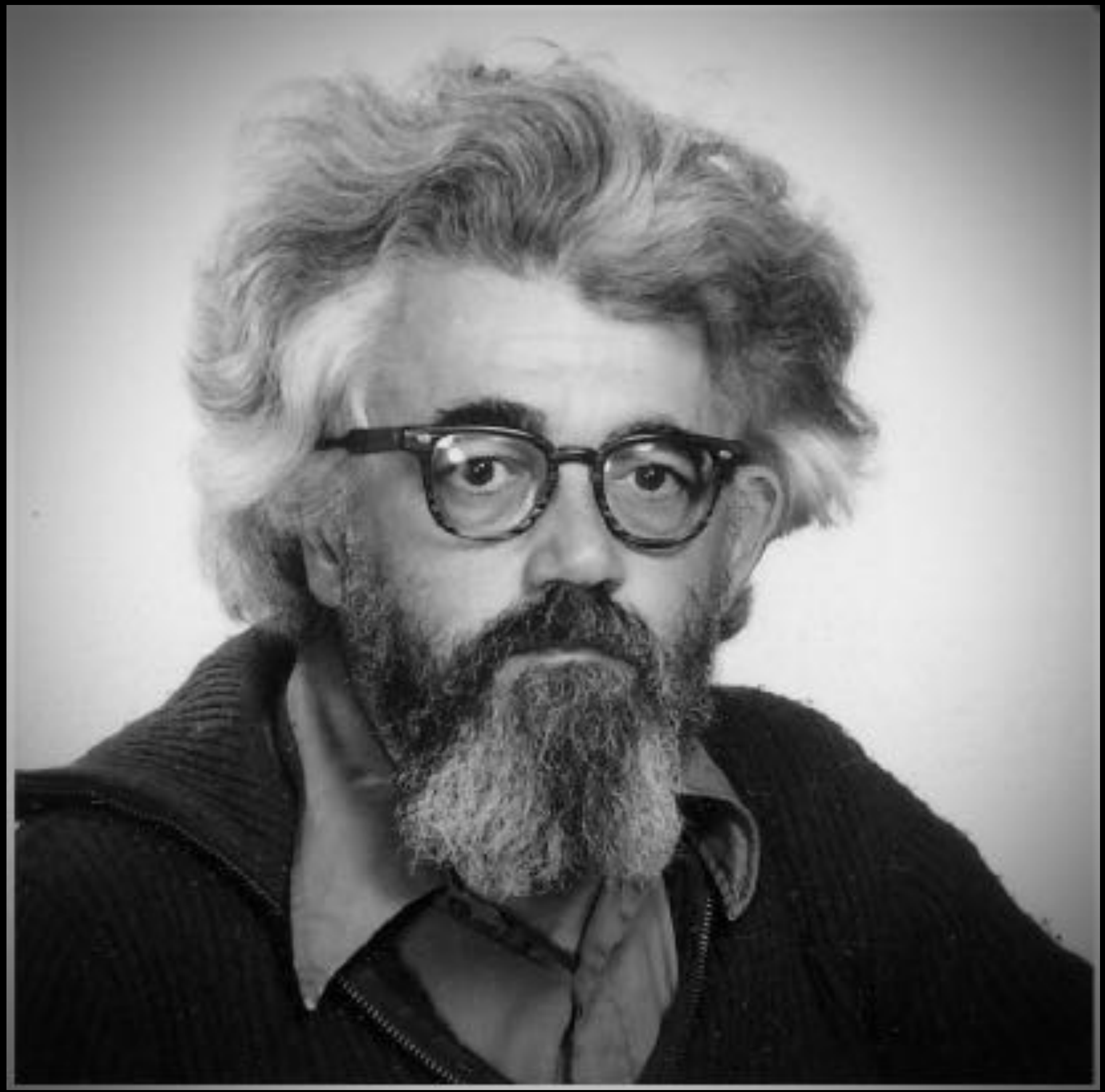
- ..."this EVAL is intended for reading not for computing." (McCarthy to Russel)
- After some (small) resistance, Russel compiled the eval part down to 704 assembly
- The first Lisp interpreter was born!

```
apply[f.args] = eval[combine[f;args]]
eval[e] = [first[e] = NULL →[null[eval[first[rest[e]]]]→ T;
                              1→ F];
          first[e] = ATOM →[atom[eval[first[rest[e]]]]→ T;
                              1→F ;
          first[e] = EQ →  [eval[first[rest[e]]] = eval[first[rest[rest[e]]]]→ T;
                              1→ F];
          first[e] = QUOTE→ first[rest[e]];
          first[e] = FIRST→ first[eval[first[rest[e]]]];
          first[e] = REST→ rest[eval[first[rest[e]]]];
          first[e] = COMBINE →combine[eval[first[rest[e]]];
                                      eval[first[rest[rest[e]]]]];
          first[e] = COND →evcon[rest[e]];
          first[first[e]] = LAMBDA→evlam[first[rest[first e]]];
                                         first[rest[rest[first[e]]]];
                                         rest[e]]
          first[first[e]] = LABEL → eval[combine[subst[first[e];
                                                       first[rest[first[e]]];
                                                       first[rest[rest[first[e]]]]]
                                                 rest[e]]]]
evcon[c] = [eval[first[first[c]]] = 1 →eval[first[rest[first[e]]]];
             1 →evcon[rest[c]]]
evlam[vars;exp;args] = [null[vars] → eval[exp];
                        1→ evlam[rest[vars];
                                 subst[first[args];first[vars];exp];
                                 rest[args]]]
```

The first recorded APPLY-EVAL version

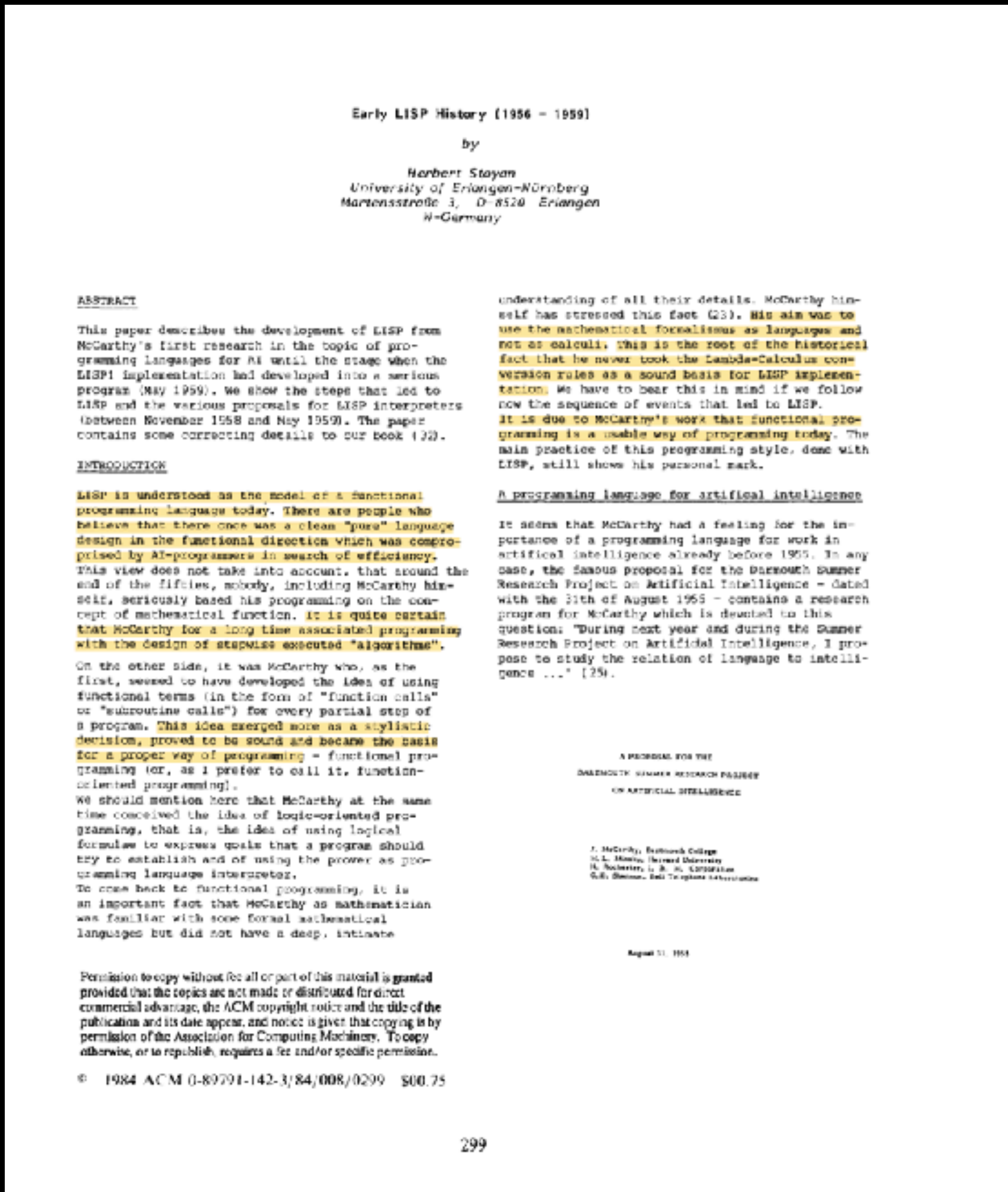
LISP: AN INSPIRING LESSON

- Well defined goals
- Baby steps, small increments, refined versions
- If the alternative doesn't work, make your own
- Constrained resources = more time to think
- Team interplay
- Ultimately, no fear!



John McCarthy 1927-2011

EARLY LISP HISTORY (1956 – 1959)



Early LISP History (1956 - 1959)

by

Herbert Stoyan
University of Erlangen-Nürnberg
Martensstraße 3, D-8520 Erlangen
W-Germany

ABSTRACT

This paper describes the development of LISP from McCarthy's first research in the topic of programming languages for AI until the stage when the LISP1 implementation had developed into a serious program (May 1959). We show the steps that led to LISP and the various proposals for LISP interpreters (between November 1958 and May 1959). The paper contains some correcting details to our book [32].

INTRODUCTION

LISP is understood as the model of a functional programming language today. There are people who believe that there once was a clean "pure" language design in the functional direction which was compromised by AI-programmers in search of efficiency. This view does not take into account, that around the end of the fifties, nobody, including McCarthy himself, seriously based his programming on the concept of mathematical function. It is quite certain that McCarthy for a long time associated programming with the design of stepwise executed "algorithms".

On the other side, it was McCarthy who, as the first, seemed to have developed the idea of using functional terms (in the form of "function calls" or "subroutine calls") for every partial step of a program. This idea emerged more as a stylistic decision, proved to be sound and became the basis for a proper way of programming - functional programming (or, as I prefer to call it, function-oriented programming).
We should mention here that McCarthy at the same time conceived the idea of logic-oriented programming, that is, the idea of using logical formulae to express goals that a program should try to establish and of using the prover as programming language interpreter.
To come back to functional programming, it is an important fact that McCarthy as mathematician was familiar with some formal mathematical languages but did not have a deep, intimate understanding of all their details. McCarthy himself has stressed this fact [23]. His aim was to use the mathematical formalismus as languages and not as calculi. This is the root of the historical fact that he never took the Lambda-Calculus conversion rules as a sound basis for LISP implementation. We have to bear this in mind if we follow now the sequence of events that led to LISP.
It is due to McCarthy's work that functional programming is a usable way of programming today. The main practice of this programming style, done with LISP, still shows his personal mark.

A programming language for artifical intelligence

It seems that McCarthy had a feeling for the importance of a programming language for work in artifical intelligence already before 1955. In any case, the famous proposal for the Dartmouth Summer Research Project on Artificial Intelligence - dated with the 31th of August 1955 - contains a research program for McCarthy which is devoted to this question: "During next year and during the Summer Research Project on Artificial Intelligence, I propose to study the relation of language to intelligence ..." [25].

A PROPOSAL FOR THE
DARTMOUTH SUMMER RESEARCH PROJECT
ON ARTIFICIAL INTELLIGENCE

by Prof. i. R. Dr. Herbert Stoyan

~ FIN ~