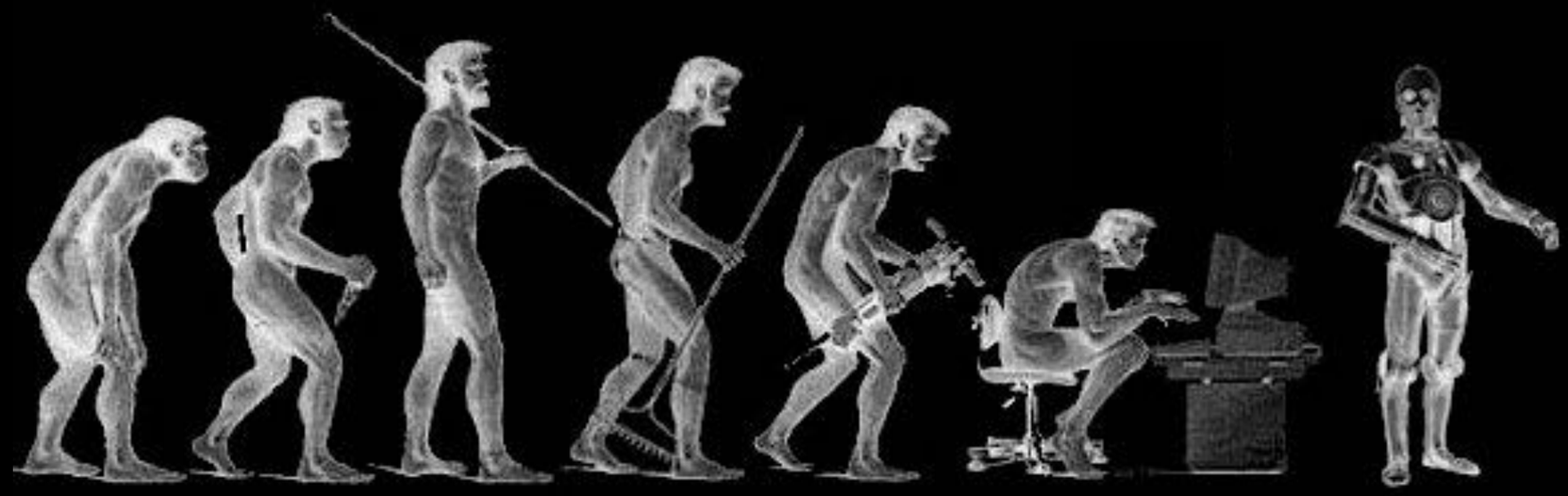
A TALE OF LISP



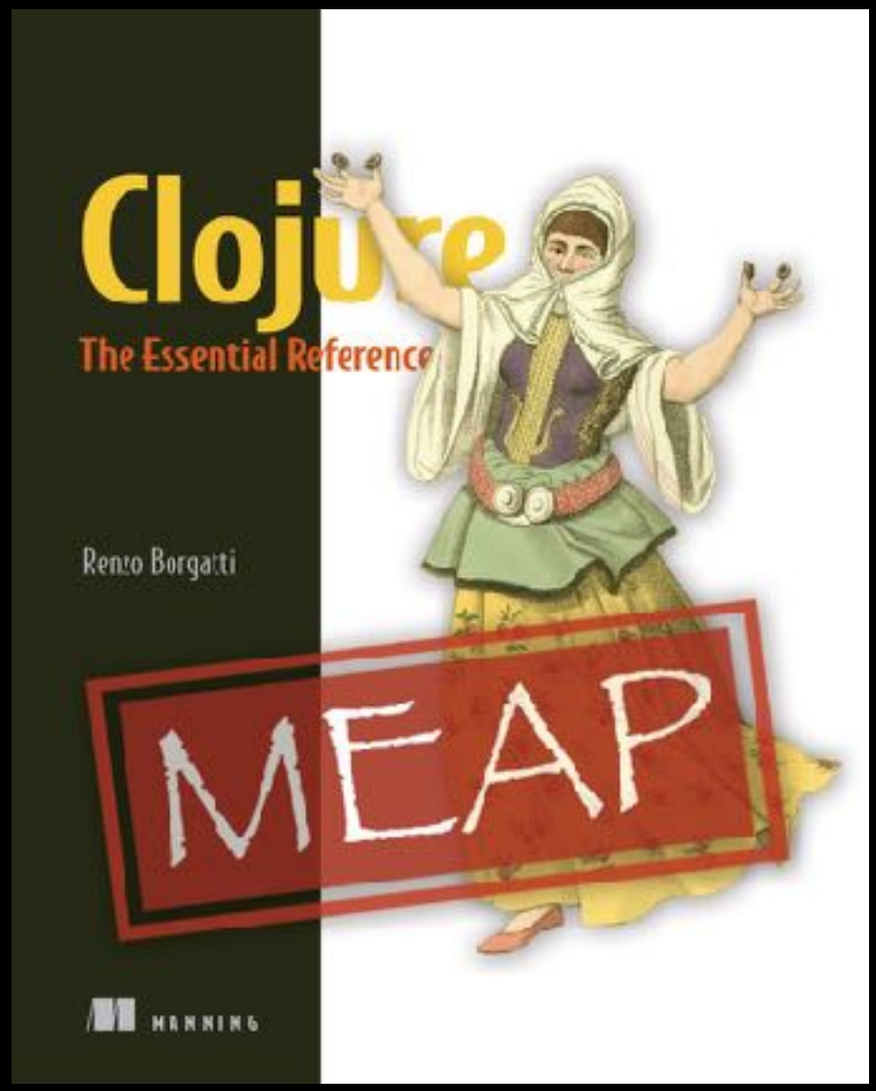
ClojureCircle Meetup

Funding Circle

Nov 2018

RENZO BORGATTI

- Software Engineer
- http://droit.tech
- github.net/reborg
- twitter.com/reborg
- Almost finished a book (!) –>
- Passionate about everything Clojure and Lisp



ONCE UPON A MIT

IN 1950 JOHN MCCARTHY, MATHEMATICIAN, WAS RESEARCHING WAYS TO TEACH COMPUTERS HOW TO LEARN

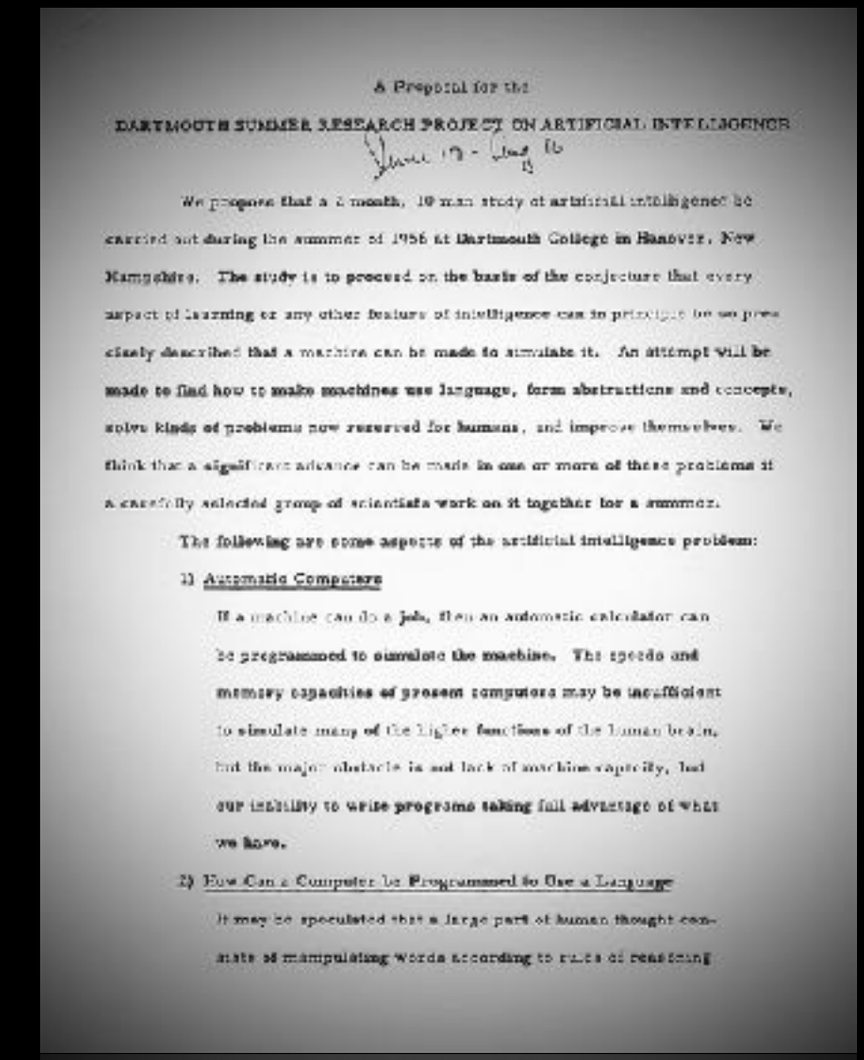
HE HAD NO IDEA HE WOULD INVENT A MILESTONE LANGUAGE

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

John McCarthy (ca 1931)

WHAT DOES IT MEAN TO TEACH MACHINES

- 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.

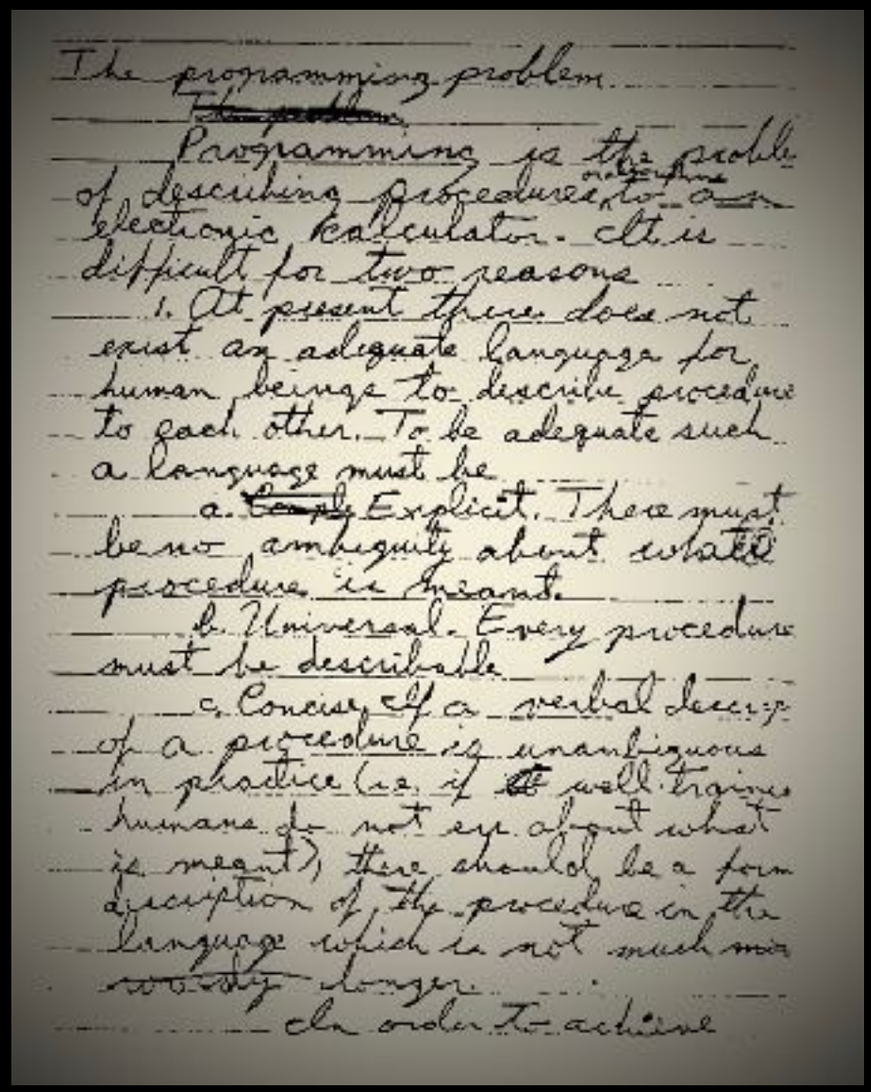
2) How Can a Computer be Programmed to Use a Language

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

Dartmouth Proposal 1955

A PROGRAMMING LANGUAGE HAS TO BE:

- Unambiguous
- Able to describe any computation (also recursive)
- Not much larger than the sentence it describes
- Abstracted away from hardware constraints
- Is there already such a language?

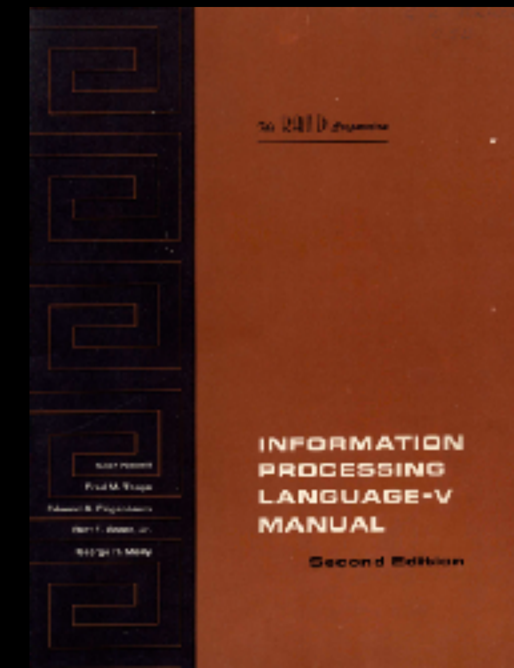


The programming Problem
Programming is the problem of describing procedures to an electronic calculator. It is difficult for two reasons
1. At present there does not exist an adequate language for human beings to describe procedure to each other. To be adequate such a language must be
a. ~~Comp~~ Explicit. 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 err 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

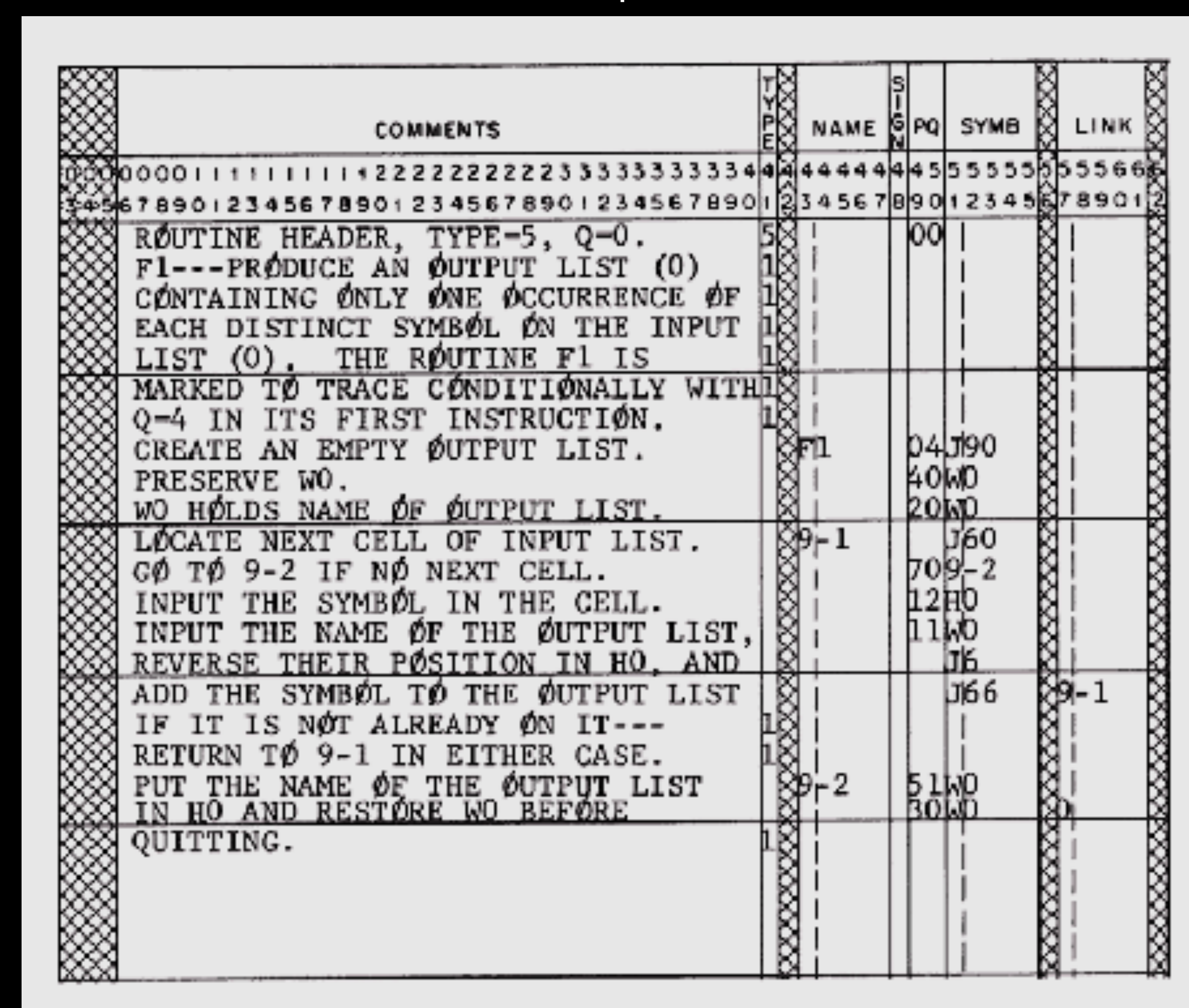
The programming Problem, manuscript, ~1955

LOOKING AROUND: IPL

- Development started in 1959
- List based assembly language
- Instructions on lists
- Extremely low level



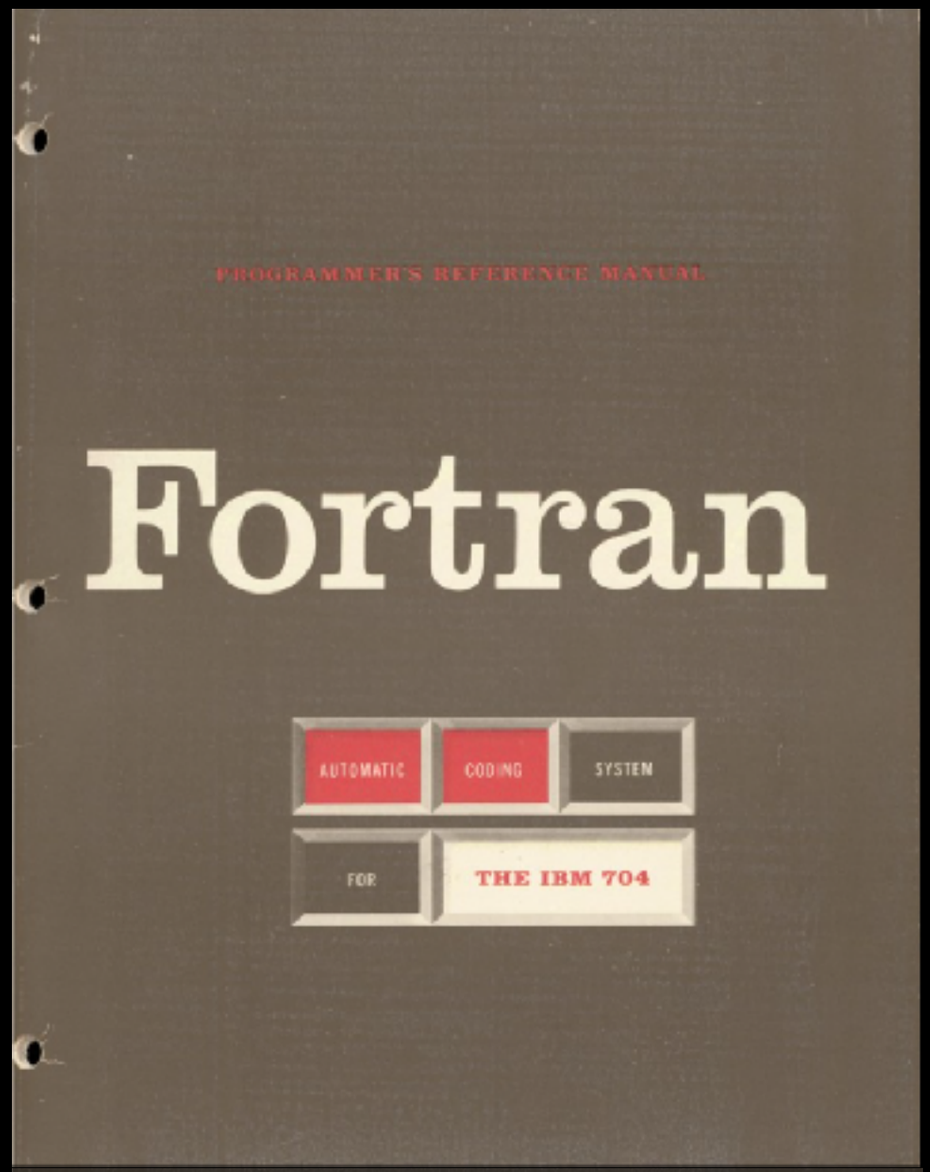
“distinct” in IPL. J90, J60 etc, are operators From the standard library.



COMMENTS	TYPE	NAME	SIGN	PQ	SYMB	LINK
RØUTINE HEADER, TYPE-5, Q-0.	5			00		
F1---PRØDUCE AN ØUTPUT LIST (0)	1					
CØNTAINING ØNLY ØNE ØCCURRENCE ØF	1					
EACH DISTINCT SYMBØL ØN THE INPUT	1					
LIST (0). THE RØUTINE F1 IS	1					
MARKED TØ TRACE CØNDITIØNALLY WITH	1					
Q-4 IN ITS FIRST INSTRUCTIØN.	1					
CREATE AN EMPTY ØUTPUT LIST.		F1		04	J90	
PRESERVE W0.				40	W0	
W0 HØLDS NAME ØF ØUTPUT LIST.				20	W0	
LØCATE NEXT CELL OF INPUT LIST.		9-1			J60	
GØ TØ 9-2 IF NØ NEXT CELL.				70	9-2	
INPUT THE SYMBØL IN THE CELL.				12	H0	
INPUT THE NAME ØF THE ØUTPUT LIST,				11	W0	
REVERSE THEIR PØSITION IN H0, AND					J6	
ADD THE SYMBØL TØ THE ØUTPUT LIST					J66	9-1
IF IT IS NØT ALREADY ØN IT---	1					
RETURN TØ 9-1 IN EITHER CASE.	1					
PUT THE NAME ØF THE ØUTPUT LIST		9-2		51	W0	
IN H0 AND RESTØRE W0 BEFØRE				30	W0	0
QUITTING.	1					

SOURCE OF INSPIRATION #2: FORTRAN

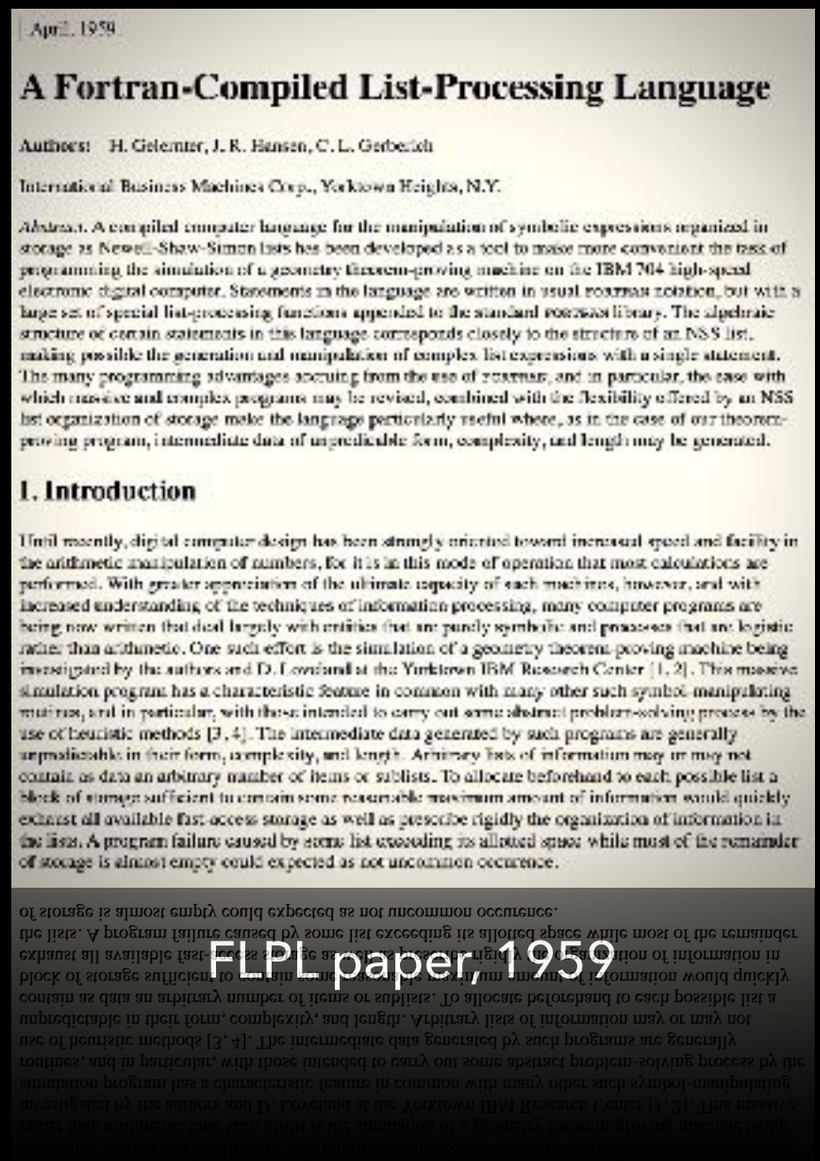
- Symbolic (allowing “a + a” after defining “a”)
- Algebraic expressions
- Arithmetic IF (expression) negative, zero, positive
- Composing sub-routines



Fortran coding system for the IBM 704, 1956

SOURCE OF INSPIRATION #3: F-LPL

- Fortran List Processing Language
- Replicate IPL concepts
- Extend Fortran with lists
- Work based on the IBM 704
- But everything had to be an integer (i.e. no symbolic domain)



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.

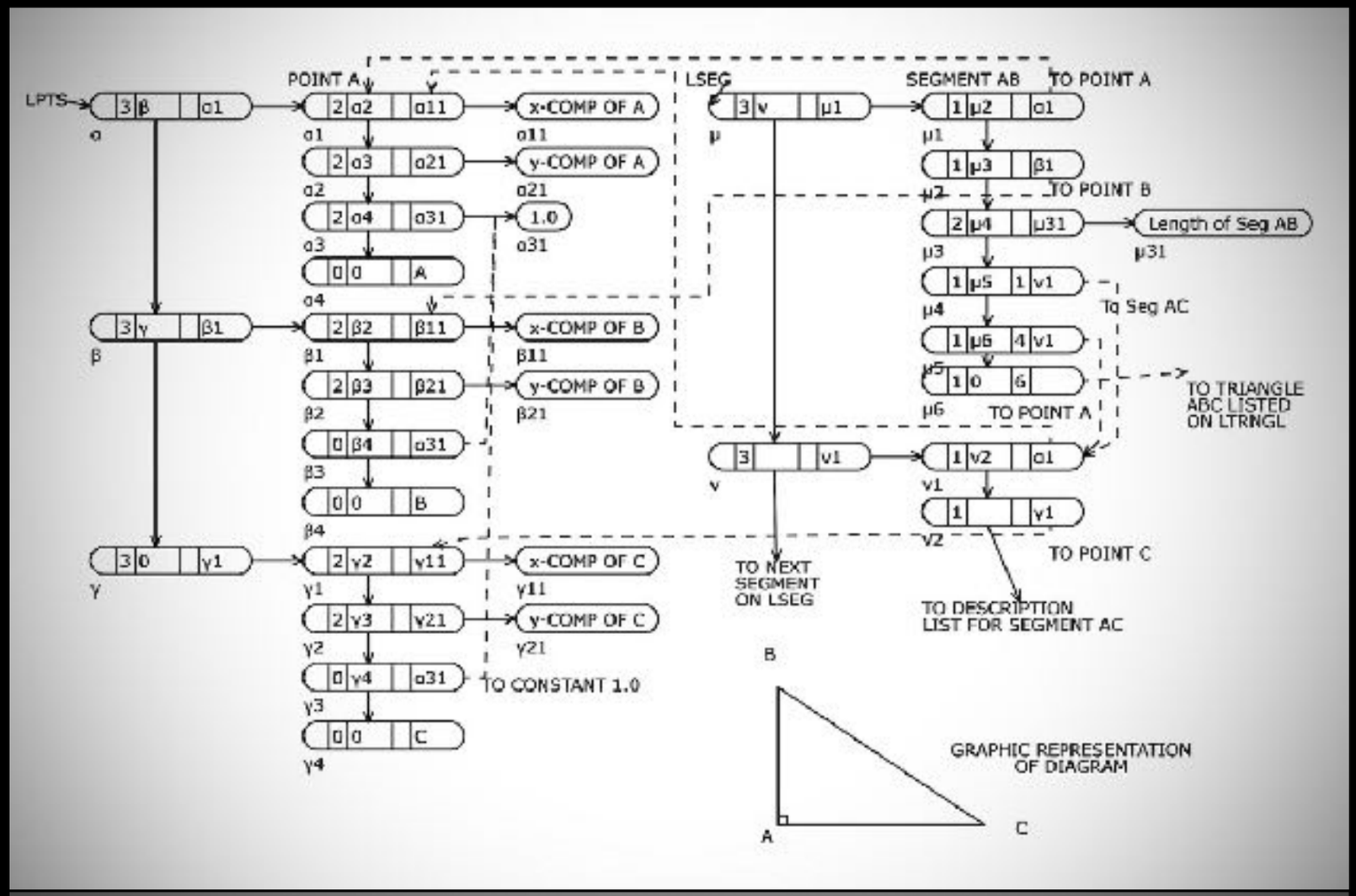
1. Introduction

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

F-LPL EXAMPLE

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

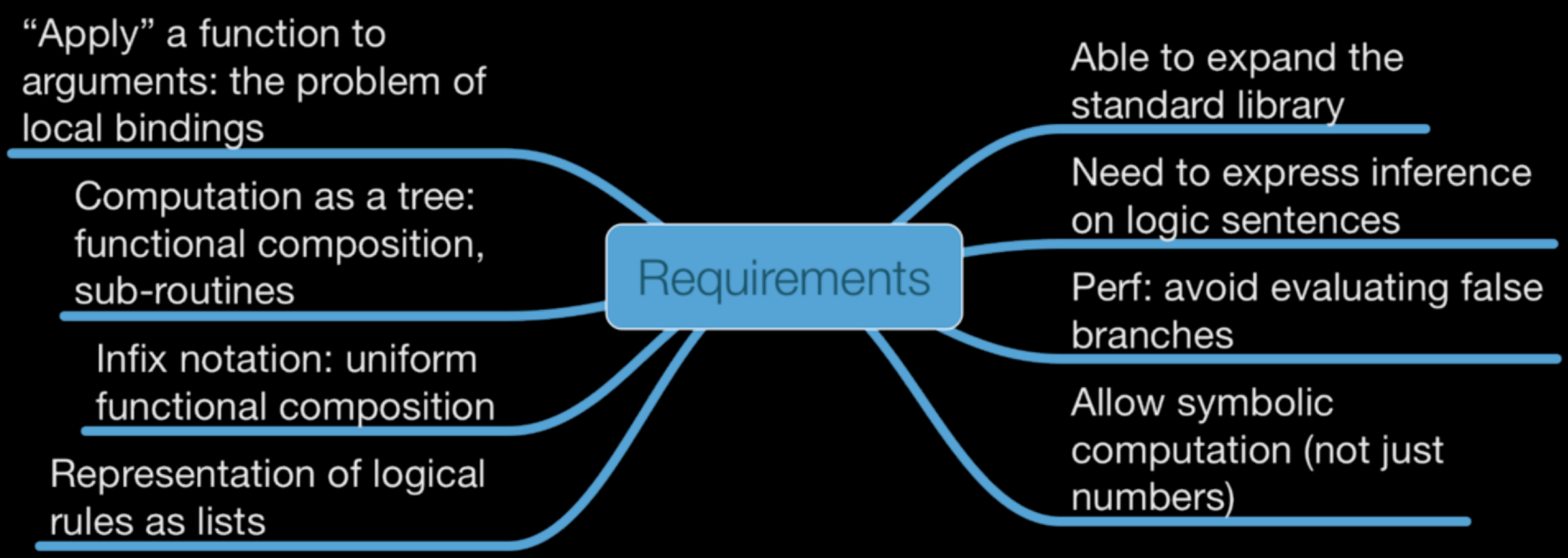



```
INDEX = LPTS. Initialize index pointer.
   10    IF(CWWF(XCAR2F(INDEX)) - 2.0) 15, 15, 20.  Go to statement 20
                if x-component is greater than 2.0; otherwise go to 15.
   15    INDEX = XCDRF(INDEX). Move index to next ``1``-word on ``LPTS``.
   20    NAMEPT = XCARF(INDEX). Retrieve machine name of point.
         GO TO (EXIT TO ROUTINE FOR PROCESSING NAMEPT).
   25    (EXIT WHEN LPTS CONTAINS NO POINT WITH X-COMP > 2.0).
```

FLPL MIGHT NOT THE BEST IDEA

- Difficult to see lists in Fortran
- Value of a function depending on register state
- Arithmetic IF with pass by reference: both branches need evaluation
- Unable to extend standard functions

GRADUALLY CONVERGING TO A SET OF REQUIREMENTS



ALGOL IS THE ANSWER!

- Proposal for: conditional statement
- Proposal for: prefix notation
- Proposal for: var assignment of functions
- Proposal for: Church lambda notation
- The compiler should translate “text” according to “rules”
- All rejected!
- **Go home McCarthy!**

Quiz time:

Algol Committee reunion (1974)

BUT RIGHT WHEN ALL HOPE SEEMS LOST...

THE PERFECT STORM

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

The basement at MIT in 1958 (not really :)

THE AVAILABLE HARDWARE

IBM 704

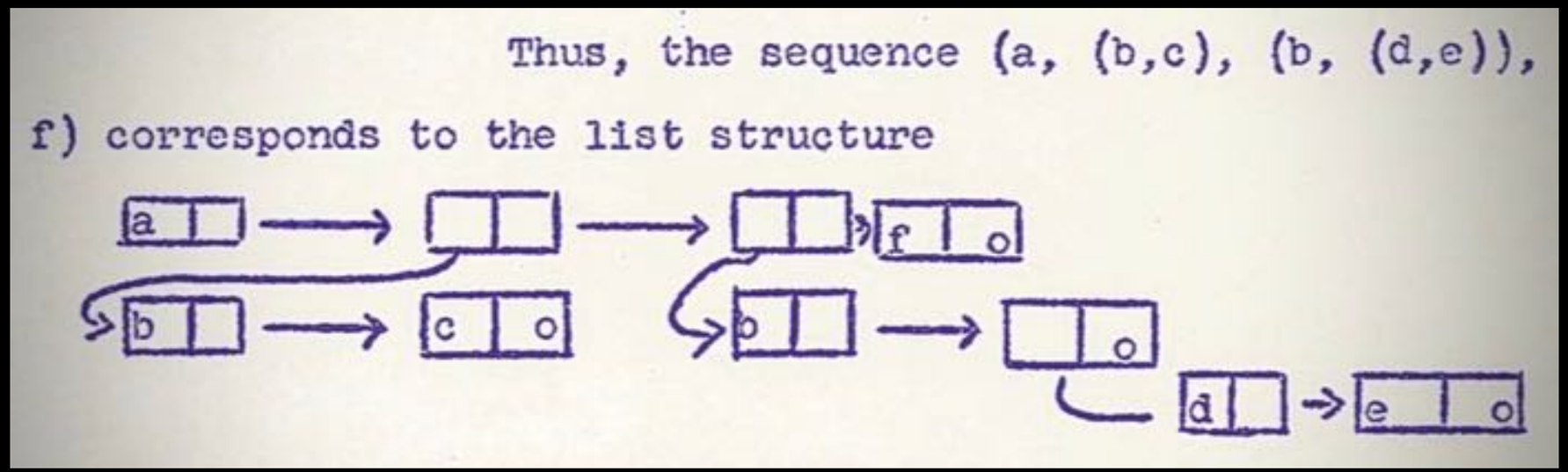
PDP-1

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

C (content of the) <name> **R** (egistry)

HOW TO MODEL LISTS ON 36 BITS WORDS?



- Here’s why we are still talking about “cons” (cells)
- **consel (e,ø)** puts "e" in the address and "ø" 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 (x)** returns the word at location "x" back to free storage.

VERY FIRST CUT

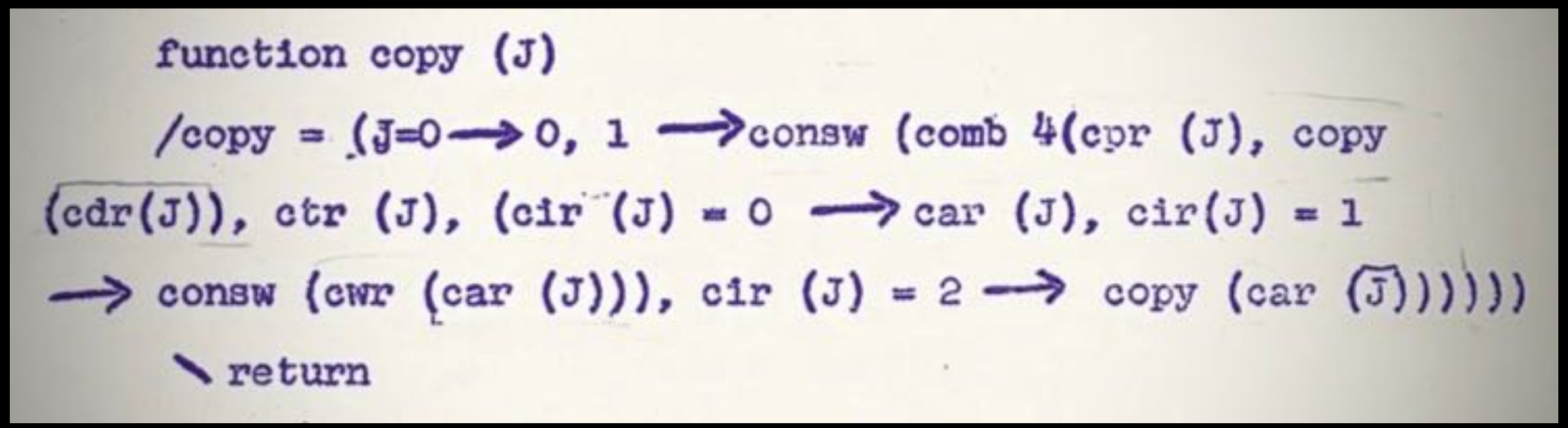
- Not exactly the Lisp you'd imagine...
- Strong Fortran influence
- Yet recursive
- "Goto" based conditionals
- Note: non-homoiconic "m-expressions".
- The List is just an implementation detail

```
subroutine eralis (J)
/ J = ø -> 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

LESS FORTRAN-Y: COPY

```
function copy (J)
  /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

DIFF AND THE FIRST "MAP"

```
      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
```

- **maplist (cdr(J),K,diff(K)))** is equivalent to **(map diff L)**
- But "K" is a dummy variable to range over the addresses
- **maplist** mutates **cdr**(J) in place.

MAPLIST: LAMBDA TO THE RESCUE

```
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)
```

diff version as appearing on the AI Memo #4

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

SUBSTITUTIONAL FUNCTIONS

```
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

- Common problem: invoke **f** on a list of **args**
- **subfun** is sublist of **f** with the rest of the list **(f, p1, p2, .. pN)**
- Then replace **car(f)** with the result of **subfun**
- **(subfun(x) (f x)) is indeed (lambda(x) (f x))**

END OF 1958 PROJECT STATUS

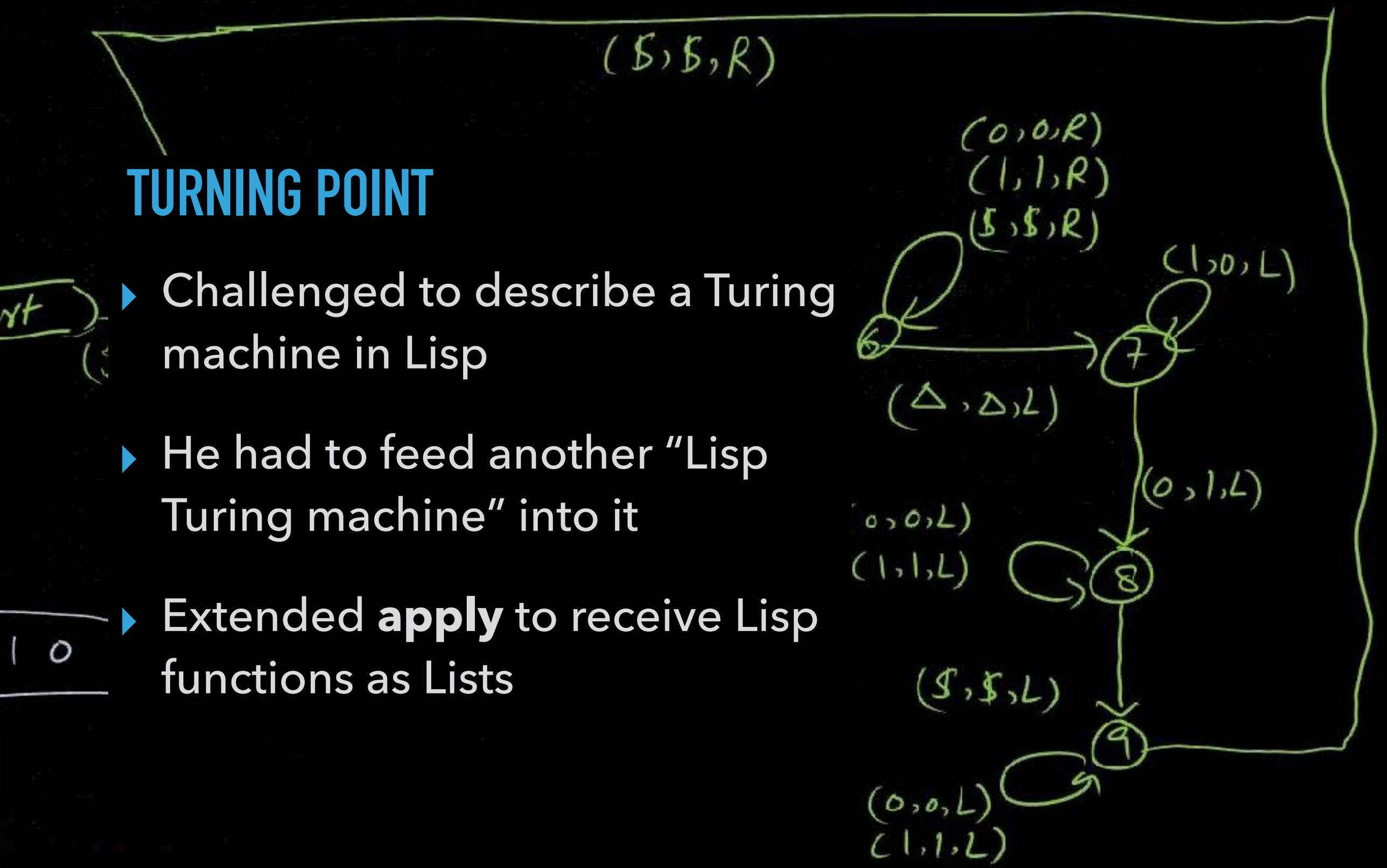
- Several functions **hand-written** in assembly for the IBM 704
- A **compiler** envisioned to automate some of the manual process
- Envisioned an "**external standard notation**" to feed **apply** with lists

```
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)

TURNING POINT

- Challenged to describe a Turing machine in Lisp
- He had to feed another “Lisp Turing machine” into it
- Extended **apply** to receive Lisp functions as Lists



THE EXTERNAL NOTATION (AKA LISP)

- An alternative syntax without arrows, square brackets, etc. called s-expressions.
- Round parenthesis () because the card puncher only had those!
- Prefix notation, giving a special meaning to the first item (function call)

Card punch recorder

EVAL

- A generic **apply** needs to understand Lisp: symbols, lambdas, lists or other apply.
- The **eval** part in apply is the condition to select the correct way to interpret a form.
- **apply** is called recursively to invoke a function on its arguments.
- Apply/Eval/Apply/Eval…

AND THEN STEVE RUSSEL CAME AROUND . . .

- Steve: “Why don’t we write this apply-eval down to assembly?”
- McCarthy: “ho, ho, you're confusing theory with practice...this EVAL is intended for reading not for computing."
- Russel compiled the eval part down to 704 assembly
- And the first **REPL** was born!

```
apply[f;args] = eval[combine[f;args]]
eval[e] = [first[e] = NULL → [null[eval[first[rest[e]]]] → T;
                              T → F];
           first[e] = ATOM → [atom[eval[first[rest[e]]]] → T;
                              T → F ;
           first[e] = EQ → [eval[first[rest[e]]] = eval[first[rest[rest[e]]]] → T;
                           T → 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]]] = T → eval[first[rest[first[e]]]];
            T → evcon[rest[c]]]
evlam[vars;exprargs] = [null[vars] → eval[exp];
                        T → evlam[rest[vars];
                                  subst[first[args];first[vars];exp];
                                  rest[args]]]
```

The first recorded APPLY-EVAL version

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

SPECIAL MENTION

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 prower 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 formalisms 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.

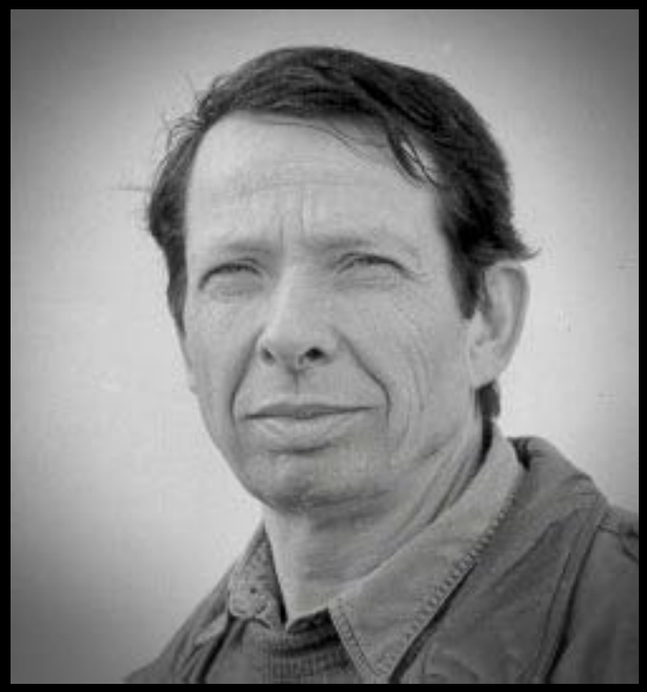
A programming language for artifical intelligence

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

J. McCarthy, Dartmouth College
M.L. Minsky, Harvard University
N. Rochester, I.B.M. Corporation
C.E. Shannon, Bell Telephone Laboratory

August 31, 1955



by Prof. i. R. Dr. Herbert Stoyan

@reborg

~ FIN ~