JOHN MCCARTHY:
the uncommon logician of common sense

Excerpt from *Out of their Minds: the lives and discoveries of 15 great computer scientists*

by Dennis Shasha and Cathy Lazere,

Copernicus Press

August 23, 2004

If you want the computer to have general intelligence, the outer structure has to be common sense knowledge and reasoning. — John McCarthy

When a five-year old receives a plastic toy car, she soon pushes it and beeps the horn. She realizes that she shouldn't roll it on the dining room table or bounce it on the floor or land it on her little brother's head. When she returns from school, she expects to find her car in more or less the same place she last put it, because she put it outside her baby brother's reach.

The reasoning is so simple that any five-year old child can understand it, yet most computers can't. Part of the computer's problem has to do with its lack of knowledge about day-to-day social conventions that the five-year old has learned from her parents, such as don't scratch the furniture and don't injure little brothers. Another part of the problem has to do with a computer's inability to reason as we do daily, a type of reasoning that's foreign to conventional logic and therefore to the thinking of the average computer programmer. Conventional logic uses a form of reasoning known as deduction. Deduction permits us to conclude from statements such as "All unemployed actors are waiters, " and " Sebastian is an unemployed actor," the new statement that "Sebastian is a waiter." The main virtue of deduction is that it is "sound" — if the premises hold, then so will the conclusions. It is also "monotonic" (a mathematical term whose root meaning is "unvarying"). If you learn new facts that do not contradict the premises, then the conclusions will still hold.

But while most of us learn deduction in school, we rarely use it in practice. The five-year-old believes her toy car will remain in its place, because she put it outside her brother's reach, but she might not feel so sure if she had seen him climb on a chair as she left for school that day. The common sense reasoning of a five-year-old relies on educated guesses that may have to be revised "non-monotonically" as new facts surface. Not that five-year-olds are special in this regard either.

Even that popular master of "deduction," Sherlock Holmes, didn't use much of it. In one of his most ingenious insights, during the adventure about

an injured racehorse, *Silver Blaze*, Holmes concluded that the watchdog never barked because he knew the culprit. This is clever, plausible, and holds true in the story, but it is not deduction — the dog could have been drugged, muffled, or off hunting for rabbits.

Programmers know how to get a computer to perform deduction, because the mathematics involved is well understood. But if you want a computer to perform the conjectural — but usually correct — common sense reasoning upon which human survival depends, you must invent a whole new kind of mathematical logic. Before doing so, you should study the work of John McCarthy.

There are other reasons to know about McCarthy. He invented LISP (LISt Processing), the principal language of Artificial Intelligence and since its inception a fertile source of ideas for language design. As a teacher and a poser of puzzles, he has inspired other computer scientists in a variety of subspecialties from cryptography to planarity testing as the chapters on Rabin and Tarjan will illustrate.

Born in Boston in 1927 to Communist party activists, McCarthy lived in a family on the move. While his Irish Catholic father worked as a carpenter, fisherman and union organizer, the family kept migrating, from Boston to New York and then to Los Angeles. His Lithuanian Jewish mother worked as a journalist for The Federated Press wire service then for a Communist newspaper and finally as a social worker. McCarthy connects his early interest in science with the political views of his family.

There was a general confidence in technology as being simply good for humanity. I remember when I was a child reading a book called 100,000 Whys — a Soviet popular technology book written by M. Ilin from the early 1930's. I don't recall seeing any American books of that character. I was very interested to read ten or fifteen years ago about some extremely precocious Chinese kid and it was remarked that he read 100,000 Whys.

...

In 1949, now in the graduate mathematics program at Princeton, McCarthy

made his first attempt to model human intelligence on a machine.

I considered an intelligent thing as a finite automaton connected to an environment that was a finite automaton. I made an appointment to see John von Neumann. He was encouraging. He said, "Write it up, write it up." But I didn't write it up because I didn't feel it was really good.

An automaton models a machine that moves from one state to another over time. For example, a standard transmission car moves from the "off-state" to the "in-neutral-but-on" state when the driver engages the ignition. It then moves to the "in-first-and-on" state when the driver switches gears to drive. An interacting automaton moves from state to state depending on its own state and on what it observes regarding the states of other machines. Some automata are intelligent (presumably, the driver is) whereas others need not be. The interacting automata model attempts to establish a continuum between the two kinds.

In 1951, the 23-year-old McCarthy completed his Ph.D. in mathematics, specializing in differential equations. He stayed on at Princeton for two years. Throughout this time, McCarthy's interest in making a machine as intelligent as a human persisted.

....

So when I started to organize the Dartmouth project in 1955, I wanted to nail the flag to the mast and used the term **Artificial Intelligence** to make clear to the participants what we were talking about.

The Dartmouth 1956 Summer Research Project on Artificial Intelligence proved to be a landmark event in the history of computer science. The ambitious goal for the two-month ten-person study was to (in the words of the proposal) "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."

The four organizers, McCarthy, Marvin Minsky (then at Harvard), Nat Rochester (a prominent IBM computer designer), and Shannon, asked the Rockefeller Foundation to provide what in today's terms looks like a rather

quaint financial request: $1200 for each faculty level participant and "railway fare for participants coming from a distance"— for a total of $7,500.

In his part of the proposal, McCarthy wrote that he would study the relation of language to intelligence with the hope of programming a computer to "Play games well and do other tasks." Looking back at the conference almost 40 years later, McCarthy's applies his notoriously blunt style to his hopes at the time.

While the conference came nowhere close to resolving the enormously difficult problem of creating a truly intelligent machine, it established goals and techniques that led to the recognition of Artificial Intelligence as a separate, and ultimately, a driving field of study within computer science. Although many of the conference attendees never pursued further research in the field, a few presented work of lasting impact.

Allen Newell, J. C. Shaw, and Herbert Simon of Carnegie Mellon University described a second version of their Information Processing Language (IPL 2). IPL2 resulted from the three scientists' effort to build a program that could prove theorems in elementary logic called the Logic Theory Machine. To do this, they needed a programming language that could manipulate symbols for objects such as chess pieces and truth values of logical variables. Because this was very different from performing arithmetic on numbers, they suggested using "list structure."

To illustrate the use of lists for symbolic processing, let us take some liberties with *Alice in Wonderland*. Suppose a mutant Cheshire Cat tells Alice "Either I am mad or the Hatter is mad." Suppose we let C, H, and A represent the respective assertions that the Cheshire Cat, Hatter, or Alice is mad. We might render the cat's statement in list format as (or C H). The Cat then tells Alice, "Either you or the Hatter is mad." A clever Alice might put this statement together with the other as (and (or C H) (or A H)). Finally, the cat says, " Only one of the three of us is mad." That is, at least two are NOT mad. Alice might represent this as (and (or C H) (or A H) (or (and (not A) (not C)) (and (not A) (not H)) (and (not C) (not H)))).

After putting these statements in list format, we can then form rules for list manipulation such as (and (or X Y) (or Z Y)) = (or (and X Z) Y). That is, if either X or Y and either Z or Y hold, then either X and Z hold or Y holds. Applying that rule and others will allow us to conclude (and H (not C) (not A)). According to the Cat, then, only the Hatter is mad.

The beauty of using lists for logical reasoning is that they can grow and shrink and reform themselves as the inferences proceed. Further, one can represent both rules and data in the same form. To most participants at the conference, list manipulation looked to be a clear winner. The other achievement of the Dartmouth Conference was Marvin Minsky's proposal to build a geometry theorem prover. Minsky had tried out some examples on paper and proposed that proving theorems in geometry might be a good application of the rule-based approach that Newell and Simon had advocated. Nathaniel Rochester at IBM decided to implement the program and hired Herbert Gelernter to do the work. Gelernter would later develop a tool to help organic chemists synthesize new chemicals. His son David is a renowned researcher and designer in parallel programming and in medical Artificial Intelligence. McCarthy acted as a consultant to the theorem proving project, giving him the chance to program intelligent behavior.

Gelernter and his assistant Carl Gerberich took my suggestion to start with FORTRAN and made something which they called FLPL — FORTRAN LIST PROCESSING LANGUAGE. And they added some ideas of their own.

In 1956, John Backus and his IBM team had introduced FORTRAN, the first higher level programming language we discussed in the last chapter. FORTRAN freed programmers working on numerical computations from the difficulty of writing assembly language specific to each computer. To this day, it remains a lingua franca for scientific and engineering computing. FLPL was a first attempt to extend FORTRAN's capabilities to symbolic manipulation. In the summer of 1958, while working at IBM, McCarthy tried to use FLPL on an application he knew extremely well from his high school reading.

I wrote list programs for differentiation [an operation from calculus] of

algebraic expressions. The idea immediately demanded recursive conditional expressions. [For example, the derivative of e^{x^2} results in $e^{x^2} \times$ derivative of (x^2); this is recursive because the derivative of an expression is defined in terms of the derivative of a component of the expression.]

If FORTRAN had allowed recursion, I would have gone ahead using FLPL. I even explored the question of how one might add recursion to FORTRAN. But it was much too kludgy.[1]

Instead, McCarthy invented LISP. Whereas Newell, Shaw, and Simon later described IPL as a language that grew complex over time, McCarthy describes LISP as a language that became simpler over time. From the beginning, McCarthy had a team of eager, if captive, collaborators.

When I returned to M.I.T. in the fall of 1958, Minsky and I had a big work room, a key punch, a secretary, two programmers and six mathematics graduate students. We had asked Jerry Wiesner for these things the preceding spring in order to form an Artificial Intelligence Project.

We got them even though we hadn't prepared a written proposal. Fortunately, the Research Laboratory of Electronics at M.I.T. had just been given an open-ended Joint Services Contract by the U.S. Armed Forces and hadn't yet committed all the resources. I think that such flexibility was one of the reasons the U.S. started in AI ahead of other countries. The Newell-Simon work was also possible because of the flexible support the U.S. Air Force provided the Rand Corporation.

As the effort progressed, McCarthy tried to improve the language's expressive power. In 1959, while trying to show that the language could formulate any computable function, he added a feature that became known as "eval."

Eval permits a program to define a new function or procedure and then execute it as part of that program. Most languages would force the program

[1]IBM soon lost interest in Artificial Intelligence anyway. Some customers thought their jobs might be threatened by intelligent machines, so by the early 1960s IBM's marketing message was that computers were just dumb number-crunchers who would do what they were told — no more, no less.

to stop and "recompile" before executing the new function. Since the **eval** function can take any function and execute it, **eval** plays the role of a "Universal Turing Machine," a universal simulator of other computers.

The **eval** notion serves very practical purposes. For example, a brokerage house runs its computing services 24 hours a day, 7 days per week because of international financial market activity. Suppose someone writes a program to analyze Reuters stock data in a new way. Brokers would want to use the program immediately, but only if they could do so without ever losing the use of their machines. **Eval** makes this possible.

...

Basic Operations in LISP

Typical LISP functions and procedures either take a list and break it apart or take a few lists and then form a new list. For example, the function "append," takes two lists and creates a new one by attaching one to the end of the other. This might be useful if you were to build a sentence out of noun and verb phrases. Let us look at the relationship between two characters, Bob and Alice: (append (Bob kissed) (Alice)) creates the list (Bob kissed Alice).

Another useful function, "reverse," might order the elements in a list from back to front. In that case (reverse (append (Bob kissed) (Alice))) would create (reverse (Bob kissed Alice)) which would then create (Alice kissed Bob). On the other hand (append (reverse (Bob kissed)) (Alice)) would create (append (kissed Bob) (Alice)) which would then create (kissed Bob Alice).

However Bob and Alice work out their relationship, we can see that a LISP program can be composed from functions that take lists and produce new lists.

Recursion and Eval in LISP

McCarthy made recursion a centerpiece of LISP's processing strategy. Recursion is a technique for defining an operation in terms of itself. Programmers get around the taboo of circular definitions by making the defining instance

refer to a simpler problem.

For example, if the list contains one element, you could define the reverse of a list as the list itself. Or if the list has several elements, you could define the reverse of of the list as: append the reverse of all elements of the list except the first with the list containing the first element.

If that seemed like a mouthful, try this. Call the first element of list L the head of L, denoted (head L), and the rest of the elements the tail of L or (tail L). We then could write out this program in the spirit (though not the syntax) of LISP as:

define (reverse L) as

if L has one element then L

else (append (reverse (tail L)) (list (head L)))

Let's try this on a concrete example — this time with a menage a trois, Alice, Bob and Carol: (reverse (Alice Bob Carol)) = (append ((reverse (Bob Carol)) (Alice))) = (append ((Carol Bob) (Alice))) = (Carol Bob Alice).

Because functions and procedures are themselves defined as lists, they can be constructed through other functions and procedures. The eval function can then take such a function or procedure and execute it.

McCarthy's pioneering use of recursion in LISP was not lost on the international committee charged with the design of ALGOL (the language for which Backus and Naur invented Backus-Naur form). In the Paris committee meeting on the language in 1960, McCarthy proposed both recursion and conditional expressions. The committee liked the ideas, but rejected the syntax of conditional expressions.

...

LISP has reigned as the standard language of Artificial Intellgence for almost 40 years. McCarthy did not anticipate its longevity and even proposed changes to make it similar to ALGOL. AI programmers preferred LISP's original syntax, however, and McCarthy, like Backus before him and Kay after, lost control over the language.

...