JOHN BACKUS:
a restless inventor

Excerpt from *Out of their Minds: the lives and discoveries of 15 great computer scientists*
by Dennis Shasha and Cathy Lazere,
Copernicus Press

We didn't know what we wanted and how to do it. It just sort of grew. The first struggle was over what the language would look like. Then how to parse expressions — it was a big problem and what we did looks astonishingly clumsy now.... John Backus on the invention of FORTRAN

Necessity, says the adage, is the mother of invention. Yet some inventors are motivated less by necessity than by sheer irritation at the messiness or inefficiency of the way things are. John Backus is such an inventor. He played an inspirational role in three great creations: FORTRAN, the first high level programming language; Backus-Naur Form, which provides a way to describe grammatical rules for high level languages; and lastly a functional programming language called FP, which advocates a mathematical approach to programming. Today, each of his inventions drives research and commercial agendas throughout the planet. Yet Backus's own life is one of restless energy — from his youth through his retirement.

A distaste for inefficiency seems to run in the family. Before World War I, Backus's father had risen from a modest background to the post of chief chemist for the Atlas Powder Company, a manufacturer of nitroglycerine to be used in explosives. His promotion came for good reason.

Their plants kept blowing up or having very poor yields and they couldn't figure out why. The yield was very temperature sensitive. My father discovered that the very expensive German thermometers they had were incorrect. So, he went to Germany and studied thermometer making and got some good thermometers and their plants stopped blowing up so much.

During World War I, Backus senior served as a munitions officer. A promised postwar job at Dupont never materialized, so he became a stockbroker instead. By the time John Backus was born in Philadelphia in 1924, his father had grown rich in the postwar boom. Backus spent his early years in Wilmington, Delaware and attended the prestigious Hill School in Pottstown, Pennsylvania.

I flunked out every year. I never studied. I hated studying. I was just

goofing around. It had the delightful consequence that every year I went to summer school in New Hampshire where I spent the summer sailing and having a nice time.

....

I really didn't know what the hell I wanted to do with my life. I decided that what I wanted was a good hi fi set because I liked music. In those days, they didn't really exist so I went to a radio technicians' school. I had a very nice teacher — the first good teacher I ever had — and he asked me to cooperate with him and compute the characteristics of some circuits for a magazine.

I remember doing relatively simple calculations to get a few points on a curve for an amplifier. It was laborious and tedious and horrible, but it got me interested in math. The fact that it had an application — that interested me.

Backus enrolled at Columbia University's School of General Studies to take some math courses. He disliked calculus but enjoyed algebra. By the spring of 1949, the 25 year-old Backus was a few months from graduating with a B.S. in mathematics, still without any idea what to do with his life.

One day that spring, Backus visited the IBM Computer Center on Madison Avenue. He was taken on a tour of the Selective Sequence Electronic Calculator (SSEC), one of IBM's early electronic (vacuum tube) machines.

The SSEC occupied a large room, and the huge machine bulged with tubes and wires. While on the tour, Backus mentioned to the guide that he was looking for a job. She told Backus to talk to the director.

I said no, I couldn't. I looked sloppy and disheveled. But she insisted and so I did. I took a test and did ok.

Backus was hired to work on the SSEC. The machine was actually not a computer in the modern sense. It had no memory to store software, so programs had to be fed in on punched paper tape. With its thousands of electromechanical parts, the SSEC was not too reliable either.

It was fun to work on that machine. You had the whole thing to yourself.

You had to be there because the thing would stop running every three minutes and make errors. You had to figure out how to restart it.

The state of programming was just as primitive.

You just read the manual and got the list of instructions and that was all you knew about programming. Everybody had to figure out how to accomplish something and there were of course a zillion different ways of doing it and people would do it in a zillion different ways.

Backus worked on the SSEC for three years. His first major project was the calculation of a lunar ephemeris — the position of the moon at any given moment.

....

You had to know so much about the problem — it had all these scale factors — you had to keep the numbers from overflowing or setting big round off errors. So programming was very complicated because of the nature of the machine.

Working with Harlan Herrick at IBM, Backus created a program called Speedcoding to support computation with "floating point numbers." Floating point numbers carry their scaling factor around with them, thus relieving the programmer from the drudgery of that responsibility. Backus' experience with Speedcoding set the stage for a much greater challenge.

Everybody was seeing how expensive programming was. It cost millions to rent machines and yet the cost of programming was as big or bigger. Programming was expensive because you had to hire many programmers to write in "assembly" or second generation languages, which were only one step removed from the binary or machine code of 0s and 1s. Assembly language was time-consuming; it was an arduous task to write commands that were specific to a particular machine and then to have to debug the multitude of errors that resulted. As a consequence, the ultimate goal of the program was often lost in the shuffle.

FORTRAN — The First High Level Computer Language

In December of 1953, Backus wrote a memo to his boss at IBM, Cuthbert

Hurd, suggesting the design of a programming language for the IBM 704. That project became known as Formula Translation — FORTRAN. Its goals were straightforward.

It would just mean getting programming done a lot faster. I had no idea that it would be used on other machines. There were hardly any other machines.

But first, Backus had to overcome the persuasive arguments of von Neumann who was then a consultant to IBM.

He didn't see programming as a big problem. I think one of his major objections was you wouldn't know what you were getting with floating point calculations. You at least knew where trouble was with fixed point if there was trouble. But he wasn't sensitive to the issue of the cost of programming. He really felt that FORTRAN was a wasted effort.

Hurd approved the project anyway and von Neumann didn't fight it any further. Backus hired an eclectic team of experienced programmers and young mathematicians straight out of school. By the fall of 1954, the Programming Research Group had a clear vision: to create a language that would make programming easier for the IBM 704.

As they saw it, designing the language was the easy part. Translating it to the language that the machine understood directly was the real challenge. The program that does this translation is called a "compiler." Backus and his team lacked the algorithms that nowadays permit an undergraduate to design and implement a new language compiler within a semester. Specifically, they had no good way to design the heart of the compiler, the part known as the "parser" or grammatical analyzer.

Parsing is the computer equivalent of diagramming sentences, an activity most of us happily forget once we leave junior high school. To diagram a sentence, you identify the relationship between different parts of speech: article, noun, verb, adjective, adverb, and draw them in the form of a tree. [See figure 1.]

In the case of a language compiler, the parser draws the tree and then

translates a "high level" language into "machine language." Machine language consists of a sequence of instructions that are imprinted directly into the circuitry of a computer. (Different computer models often support different instructions — that's why a Mac program may not run on an IBM compatible.)

To make a machine language program efficient, one must be careful about using the very fast, short-term memory in the machine — the machine's registers. Most modern machines have at least 16 registers, although some have thousands. This is a tiny number compared to the millions of bytes of information in the slower RAM (random access memory). But all the computational action happens in the registers.

For example, an arithmetic expression such as (A + B)/C would be translated to the following sequence:

```
copy A to register 1
copy B to register 2
copy C to register 3
add register 1 and register 2 and put result in register 1
divide register 1 by register 3 and put the result in register 1.
```

You may notice that the arithmetic operations require the data to be in registers (in some machines the data need not be in registers, but then the operation is much slower). If the next arithmetic operation needs variables D and E, then the machine translation must decide whether to put D and E in registers 4 or higher or to reuse 1, 2, or 3. This decision in turn depends on when A, B, and C will next be needed and many other factors. It's a hard and still unsolved problem. Backus and his team were one of the first to have to reach some reasonable solution.

But the biggest challenge had to do with a new feature of FORTRAN: the DO statement. These statements programmers to perform repeated computations easily such as

```
DO 13  J = 1, 100
```

```
C[J] = A[J] + B[J]
13 CONTINUE
```

This sequence of instructions adds the first element of A to the first element of B to obtain the first element of C, then adds the second element of A to the second element of B to obtain the second element of C, and so on.

Compiling a DO statement efficiently requires using special registers known as "index" registers. On the IBM 704, for which FORTRAN was originally designed, there were only three index registers so they were a precious resource.

Harlan [Herrick] invented this DO statement and so we knew we would have problems there — how to do it. We had 3 index registers and all these subscripts [J in the example above, though a complex program can have 20 or more].

It was very very difficult to figure out which index registers to use given the information in the program. You would get terrible code if you tried to do it in a general way. We knew that we would have to analyze code for its frequency and all kinds of stuff.

Backus's concern for efficiency was shared by all members of the team. FORTRAN's designers knew that programmers would not use their new high level language if the machine language it generated was less efficient than what a good programmer could do by hand. For this reason, roughly half their work became directed toward generating fast code. As a result, FORTRAN has always been known for its good performance.

The IBM 704 had only about 80 users, including General Electric, United Aircraft, Lockheed and others in the airplane industry. By a fluke, Westinghouse became FORTRAN's first commercial user in April, 1957, when Backus's group sent them a punched card deck containing the language's compiler.

They just figured this was the FORTRAN deck and they got it to run without any instructions. They were doing hydrodynamics — calculating stresses of wing structure and stuff like that to design airplanes. Before that,

they would have used desk calculators and wind tunnels.

Although Backus had led his elite team for four years in a complicated endeavor that required considerable stamina, he is consistently modest about his participation in the project.

It seems very unfair to me that I get so much credit for these guys [Robert Nelson, Harlan Herrick, Lois Haibt , Roy Nutt, Irving Ziller, Sheldon Best, David Sayre, Richard Goldberg, Peter Sheridan] who invented a tremendous amount of stuff. It didn't take much to manage the team. Everybody was having such a good time that my main function was to break up chess games that began at lunch and would go on to 2 P.M.

...

In May 1958, an international committee of prominent computer scientists from business and academia convened in Zurich. Their goal was to improve FORTRAN and create a single, standardized computer language. Their creation, the International Algebraic Language, later known as ALGOL.

ALGOL had two significant advantages over FORTRAN. First, the new language introduced the notion of local variables. Every program names various data elements. In the first FORTRAN example, we named the elements A, B, and C. In general, there are so many data elements that you run the risk of using the same name twice. That is, you may introduce a name to mean something new, but that name may already exist, resulting in some very unpleasant errors. Anyone with a common name can recognize this kind of problem: bills go to the wrong address, credit is refused for the wrong reason, and late night phone calls are for the wrong person. In computer terms, names that keep their meaning over an entire program are called "global" and cases of mistaken identity are known as "name collisions."

One way to eliminate name collisions is to localize the context of names. For example, a person speaking of "the Duke" in the city of York would normally mean the Duke of York, whereas a person speaking of "the Duke" at the Newport Jazz Festival would probably mean Duke Ellington. Similarly, a local variable is a computer memory location whose name has meaning only

in a certain limited context. Outside that context, the same name may be used to designate a different memory location.

While FORTRAN allowed only global naming, the ALGOL was designed with local names. Besides being convenient, local naming permits a form of programming that John McCarthy had recently introduced to computer science known as "recursion."

Recursion permits a programmer to break a problem into smaller versions of itself and then glue together the solutions to each version. For example, to put a huge stack of papers in order, you might put half of the stack in order, then put the rest in order, then merge the two halves.

In a strictly theoretical sense, recursion and local naming didn't make the new language more powerful than FORTRAN, but they encouraged a new way of thinking that led to such later algorithmic techniques as depth-first search which is discussed in the chapter on Tarjan.

Backus liked the ideas embodied by ALGOL, but felt frustrated by the difficulty of expressing them clearly.

They would just describe stuff in English. Here's this statement — and here's an example. You were hassled in these committees [with unproductive debates over terminology] enough to realize that something needed to be done. You needed to learn how to be precise.

To address this problem, Backus applied a formalism called context-free languages that had just been invented by linguist Noam Chomsky. (See box on context-free languages.) Chomsky's work in turn had its roots in Emil Post's theoretical work on general rewriting grammars.

How Backus came to this synthesis promises to keep historians busy for some time.

There's a strange confusion here. I swore that the idea for studying syntax came from Emil Post because I had taken a course with Martin Davis at the Lamb Estate [an IBM think tank on the Hudson].... So I thought if you want to describe something, just do what Post did. Martin Davis tells me he did not teach the course until long afterward [1960-61 according to Davis's

records]. So I don't know how to account for it. I didn't know anything about Chomsky. I was a very ignorant person. [1]

Backus's invention eventually became famous as Backus-Naur Form due to a combination of fortuitous circumstances. It started with Backus trying to explain his ideas about precise grammars in a paper for the UNESCO meeting on ALGOL in Paris in June, 1959.

Of course, I had it done too late to be included in the proceedings. So I hand-carried this pile to the meeting. So it didn't get very good distribution. But Peter Naur read it and that made all the difference.

Naur was a Danish mathematician who improved Backus's notation and used it to describe all of ALGOL. When the programming language community started to experiment with ALGOL, Naur's manual proved to be the best reference available for describing the language syntax.

[1] Martin Davis speculates that Richard Goldberg, a Harvard-trained logician and part of the FORTRAN team, may have discussed Post's or Chomsky's work with Backus.

Context-Free Grammars and Backus-Naur Form

To gain an appreciation for Backus-Naur form, consider the following description of some English phrases. The terms noun_phrase and verb_phrase are borrowed from modern linguistics.

sentence ::= noun_phrase verb_phrase

noun_phrase ::= article adjective noun — article noun

verb_phrase ::= verb noun_phrase

adjective ::= red — blue — yellow — big— small — smart

noun ::= house — girl — boy

verb ::= likes — hits

article ::= the — a

The vertical bar symbol — represents an alternative. For example, an article can be either "the" or "a." This grammar allows us to say that the sentence "the girl likes the boy" is a good sentence, since "the girl" constitutes a noun_phrase, and "likes the boy" is a verb_phrase. The grammar would also tell us that "a smart house likes the boy" is grammatically acceptable, even though it doesn't mean much (its semantics are unclear).

In programming languages, Backus-Naur Form (or BNF, for short) also has this property. If you follow its rules, you will get a syntactically correct program. The compiler will translate it successfully to machine language, preserving the meaning in the high-level language program you started with. Of course, your original program may still produce nonsense. Compilers guarantee a correct translation only, not that the original program is correct.

...