Programming in America in the 1950s—Some Personal Impressions

JOHN BACKUS

1. Introduction

The subject of software history is a complex one in which authoritative information is scarce. Furthermore, it is difficult for anyone who has been an active participant to give an unbiased assessment of his area of interest. Thus, one can find accounts of early software development that strive to appear objective, and yet the importance and priority claims of the author's own work emerge rather favorably while rival efforts fare less well.

Therefore, rather than do an injustice to much important work in an attempt to cover the whole field, I offer some definitely biased impressions and observations from my own experience in the 1950s.

2. Programmers versus "Automatic Calculators"

Programming in the early 1950s was really fun. Much of its pleasure resulted from the absurd difficulties that "automatic calculators" created for their would-be users and the challenge this presented. The programmer had to be a resourceful inventor to adapt his problem to the idiosyncrasies of the computer: He had to fit his program and data into a tiny store, and overcome bizarre difficulties in getting information in and out of it, all while using a limited and often peculiar set of instructions. He had to employ every trick

he could think of to make a program run at a speed that would justify the large cost of running it. And he had to do all of this by his own ingenuity, for the only information he had was a problem and a machine manual. Virtually the only knowledge about general techniques was the notion of a subroutine and its calling sequence[1].

Some idea of the machine difficulties facing early programmers can be had by a brief survey of a few of the bizzare characteristics of the Selective Sequence Electronic Calculator (SSEC). This vast machine (circa 1948–1952) had a store of 150 words; instructions, constants, and tables of data were read from punched tapes the width of a punched card; the ends of an instruction tape were glued together to form a paper loop, which was then placed on one of 66 tape-reading stations. The SSEC could also punch intermediate data into tapes that could subsequently be read by a tape-reading station. One early problem strained the SSEC's capacity to the limit. The computation was divided into three phases; in the first phase a tape of many yards of intermediate results was punched out; during the second phase this tape was glued into a loop and mounted on a tape-reading station so that in the third phase it could be read many times. The problem ran successfully through many cycles of these three phases, but then a mysterious error began to appear and disappear regularly in the third phase. For a long time no one could account for it. Finally, the large pile of intermediate data tape was pulled from the bin below its reading station and a careful inspection revealed that it had been glued to form a Möbius strip rather than a simple loop. The result was that on every second revolution of the tape each number would be read in reverse order.

Today a programmer is often under great pressure from superiors who know just how and how long he should take to write a program; his work is no longer regarded as a mysterious art, and much of his productive capacity depends on his ability to find what he needs in a 6-in.-thick manual of some baroque programming or operating system. In contrast, programming in the early 1950s was a black art, a private arcane matter involving only a programmer, a problem, a computer, and perhaps a small library of subroutines and a primitive assembly program. Existing programs for similar problems were unreadable and hence could not be adapted to new uses. General programming principles were largely nonexistent. Thus each problem required a unique beginning at square one, and the success of a program depended primarily on the programmer's private techniques and invention.

3. The Freewheeling Fifties

Programming in the America of the 1950s had a vital frontier enthusiasm virtually untainted by either the scholarship or the stuffiness of academia. The programmer–inventors of the early 1950s were too impatient to hoard an idea until it could be fully developed and a paper written. They wanted to

convince others. Action, progress, and outdoing one's rivals were more important than mere authorship of a paper. Recognition in the small programming fraternity was more likely to be accorded for a colorful personality, an extraordinary feat of coding, or the ability to hold a lot of liquor well than it was for an intellectual insight. Ideas flowed freely along with the liquor at innumerable meetings, as well as in sober private discussions and informally distributed papers. An idea was the property of anyone who could use it, and the scholarly practice of noting references to sources and related work was almost universally unknown or unpracticed. Thus, of 15 papers presented at an Office of Naval Research (ONR) symposium on automatic programming for digital computers in May 1954 [2], only two have separate acknowledgements and none refers to other papers.

As in any frontier group, the programming community had its purveyors of snake oil. Sometimes the snake oil worked and sometimes it did not. Thus some early programming concepts and systems enjoyed a chimerical fame as a result of the energy with which they were publicized. Numerous talks about some system might suggest it had mysterious, almost human abilities to understand the language and needs of the user; closer inspection was likely to reveal a complex, exception-ridden performer of tedious clerical tasks that substituted its own idiosyncrasies for those of the computer. Other systems achieved good reputations by providing clear and accurate descriptions to their users, since clear descriptions were even scarcer than elegant designs.

The success of some programming systems depended on the number of machines they would run on. Thus, an elegant system for a one-of-a-kind machine might remain obscure while a less-than-elegant one for a production computer achieved popularity. This point is illustrated by two papers at the 1954 ONR symposium [2]. One, by David E. Muller, describes a floating-point interpretive system for the ILLIAC designed by D. J. Wheeler. The other, by Harlan Herrick and myself, describes a similar kind of system for the IBM 701 called Speedcoding. Even today Wheeler's 1954 design looks spare, elegant, and powerful, whereas the design of Speedcoding now appears to be a curious jumble of compromises. Nevertheless, Wheeler's elegant system remained relatively obscure (since only ILLIAC users could use it) while Speedcoding provided enough conveniences, however clumsily, to achieve rather widespread use in many of the eighteen 701 installations.

4. The Priesthood

Just as freewheeling westeners developed a chauvinistic pride in their frontiersmanship and a corresponding conservatism, so many programmers of the freewheeling 1950s began to regard themselves as members of a priesthood guarding skills and mysteries far too complex for ordinary mortals.

This feeling is noted in an article by J. H. Brown and John W. Carr, III, in the 1954 ONR symposium: ". . . many 'professional' machine users strongly opposed the use of decimal numbers . . . to this group, the process of machine instruction was one that could not be turned over to the uninitiated." This attitude cooled the impetus for sophisticated programming aids. The priesthood wanted and got simple mechanical aids for the clerical drudgery which burdened them, but they regarded with hostility and derision more ambitious plans to make programming accessible to a larger population. To them, it was obviously a foolish and arrogant dream to imagine that any mechanical process could possibly perform the mysterious feats of invention required to write an efficient program. Only the priests could do that. They were thus unalterably opposed to those mad revolutionaries who wanted to make programming so easy that anyone could to it.

There was little awareness even as late as 1955 and 1956 that programming methods of that era were the most time-consuming and costly roadblock to the growth of computing. Thus, of 21 articles in the *Journal of the ACM* for all of 1955, only three concern general-purpose programming aids: one by H. Rutishauser discusses his astonishingly early plans for an algebraic compiler; another describes an interesting floatingpoint system at the University of Toronto; and the third gives a punched-card method for alleviating some programming drudgery.

It was a time in which recognition of a basic need was often the key insight leading to a significant development. But due to the resistance of the priesthood even the announcement of an important insight or invention was likely to be ignored unless it was accompanied by a widely distributed system that proved the practicality of the idea beyond a doubt. A good example of the resistance of the priesthood to revolutionary ideas is the reception it gave to the world's first algebraic compiler, produced by Laning and Zierler at MIT.

5. The Priesthood versus the Laning and Zierler Algebraic Compiler

Very early in the 1950s, J. Halcombe Laning, Jr., recognized that programming using algebraic expressions would be an important improvement. As a result of that insight he and Neal Zierler had the first algebraic compiler running on WHIRLWIND at MIT in January 1954 [3]. (A private communication from the Charles Stark Draper Laboratory indicates that they had demonstrated algebraic compiling sometime in 1952!) The priesthood ignored Laning's insight for a long time. A 1954 article by Charles W. Adams and Laning (presented by Adams at the ONR symposium) devotes less than 3 out of 28 pages to Laning's algebraic system; the rest are devoted to other MIT systems. The complete description of the system's method of operation as given there is the following [2, p. 64]:

> The system is mechanized by a compilation of closed subroutines entered from blocks of words, each block representing one equation. The sequence of equations is stored on the drum, and each is called in separately every time it is used. The compiled routine is then performed interpretively using the CS [MIT Comprehensive System] routines.

The article points out that the system yields a 10-to-1 reduction in speed, but that such a large reduction is due in part to the fact that the system was begun when the WHIRLWIND had a store of only 1024 registers. The elegant source code is described: single-letter variables with single subscripts, expressions and assignment statements involving variables, constants, functions, and arithmetic operators, plus some simple input, output, and control commands.

After the 1954 ONR article the Laning and Zierler system seems to be virtually unmentioned in the literature until much later when its historical significance was recognized. The extensively reported uses of the WHIRLWIND in the *ONR Digital Computer Newsletter* from October 1954 through January 1956 do not once mention the Laning and Zierler system or its use. The only early references to it I have found are (1) the Adams and Laning 1954 ONR article [2]; (2) the *ONR Digital Computer Newsletter* for April 1954, which mentions a December 1953 seminar by Laning: An Interpretive Program for Mathematical Equations; and (3) an Instrumentation Laboratory Report of 1954, E-364, by Laning and Zierler [3]. It boggles the mind to realize that this obscurity and neglect was the reaction of the priesthood to an elegant concept elegantly realized.

6. An Historical Footnote

The purpose of recounting the following historical detail is to point out that one cannot rely even on participants' accounts of an event given not too long after the event. In this case I am the culprit. I have up to now believed the following account of the origin of the FORTRAN project and have for a long time responded to questions about it accordingly: work on FORTRAN began in the summer of 1954 after some friends and I had earlier seen a demonstration of the Laning and Zierler algebraic system, and it was this demonstration which gave us the idea to use algebraic expressions as an important part of the FORTRAN language.

I have recently learned the facts of the matter from discussions with Irving Ziller, from a copy of the letter I wrote to Laning in 1954 asking for the demonstration (which Dr. Laning has kindly sent me), and from the 1954 Speedcoding article [2] mentioned earlier. The facts are these: work on FORTRAN began about January 1954 by Ziller and myself. By about April we had been joined by Harlan Herrick, who coauthored with me the paper "IBM 701 Speedcoding and other Automatic Programming systems" for the

ONR symposium on 13 and 14 of May. In that paper [2, pp. 111–112] we observe that a programmer "would like to write '$X + Y$' instead of (the machine code)" and that

> he would like to write $\Sigma a_{ij} \cdot b_{jk}$ instead of the fairly involved set of instructions corresponding to this expression. In fact a programmer might not be considered too unreasonable if he were willing only to produce the formulas for the numerical solution of his problem and perhaps a plan showing how the data was to be moved from one storage hierarchy to another and then demand that the machine produce the results for his problem.

The article goes on to raise the following questions:

> The question is, can a machine translate a sufficiently rich mathematical language into a sufficiently economical program at a sufficiently low cost to make the whole affair feasible?
>
> consider the advantages of being able to state the calculations . . . for a problem solution in a concise, fairly natural mathematical language.

I had long assumed that this article was written *after* the Laning and Zierler demonstration. It turns out, however, that my letter to Laning requesting it is dated 21 May, 1954 and the demonstration evidently took place on 2 June, 1954. The letter shows that (1) our article was written *before* we first heard of Laning's work at the ONR symposium and before the demonstration, and (2) by 21 May the FORTRAN group comprised four people: Herrick, Robert A. Nelson, Ziller, and myself. The letter states that after his talk about their work Adams had given me a copy of the Laning and Zierler report [3]. It says "our formulation of the problem is very similar to yours: however, we have done no programming or even detailed planning." It goes on to ask for a meeting on 2 June at MIT among Laning and Herrick, Ziller and myself.

The article and the letter therefore show that, much to my surprise, the FORTRAN effort was well under way before the ONR symposium and that, independently of Laning (but later), we had already formulated more ambitious plans for algebraic notation (e.g., $\Sigma a_{ij} \cdot b_{jk}$) than we were later to find in Laning and Zierler's report and see demonstrated at MIT. It is therefore unclear what we learned from seeing their pioneering work, despite my mistaken assumption over the years that we had gotten our basic ideas from them.

7. The Origins of FORTRAN

FORTRAN did not really grow out of some brainstorm about the beauty of programming in mathematical notation; instead it began with the recognition

of a basic problem of economics: programming and debugging costs already exceeded the cost of running a program, and as computers became faster and cheaper this imbalance would become more and more intolerable. This prosaic economic insight, plus experience with the drudgery of coding, plus an unusually lazy nature led to my continuing interest in making programming easier. This interest led directly to work on Speedcoding for the 701 and to efforts to have floating point as well as indexing built into the 704.

The viability of most compilers and interpreters prior to FORTRAN had rested on the fact that most source operations were not machine operations. Thus even large inefficiencies in compiling or interpreting looping and testing operations and in computing addresses were masked by the fact that most operating time was spent in floating-point subroutines. But the advent of the 704 with built-in floating-point and indexing radically altered the situation. The 704 presented a double challenge to those who wanted to simplify programming; first it removed the raison d'être of earlier systems by providing in hardware the operations they existed to provide, and second, it increased the problem of generating efficient programs by an order of magnitude by speeding up floating-point operations by a factor of ten and thereby leaving inefficiencies nowhere to hide. So what could be done now to ease the programmer's job? Once asked, the answer to this question had to be: Let him use mathematical notation. But behind that answer (in the new 704 environment) there was the really new and hard question: Can a machine translate a sufficiently rich mathematical language into a sufficiently economical machine program to make the whole affair feasible? Having asked the question and having got Cuthbert Hurd, my boss, to approve the effort, a few friends and associates of mine finally did answer it after three years of pioneering invention and hard work.

The initial external design for FORTRAN was completed in November 1954, a paper describing it was circulated, and a number of talks about it were given to prospective 704 users. All of this was met with the usual indifference and skepticism of the priesthood, with a few notable exceptions. Walter Ramshaw at United Aircraft agreed to let Roy Nutt work with us; he eventually designed and implemented most of the *I/O* features of the system plus its special assembly program. Charles W. Adams at MIT agreed that Sheldon Best could go on leave to work with us. Sidney Fernback at the Livermore Radiation Laboratory lent us the help of Bob Hughes for a short time. Harry Cantrell at G.E. in Schenectady was an enthusiastic supporter of our effort from the beginning. And my successive bosses at IBM, Hurd, Charles DeCarlo, and John McPherson, cheerfully endured our requests for more help and more time and our many missed deadlines. But, with a few other exceptions, our plans and efforts were regarded with a mixture of indifference and scorn until the final checking out of the compiler, at which time some other groups became more interested.

8. Optimization Techniques in FORTRAN

A large number of difficult problems had to be solved by the nine persons* who were the principal planners and programmers of the six sections of the 704 FORTRAN I compiler. It was their collective efforts that proved for the first time that efficient object programs could be compiled for a machine with built-in floating point and indexing. Without belittling the important contributions of the whole group, I should like to comment especially on the work of the three principal architects of the key optimization techniques which made it possible for FORTRAN-coded programs to compete with and often exceed the efficiency of hand-coded ones.

Robert A. Nelson and Irving Ziller devised general methods for analyzing and optimizing loops and references to arrays which were truly remarkable in the number of situations they could treat optimally. Their methods could move computations from the object program to the compiler and from inner to outer loops when the situation permitted. They could identify special circumstances in which even a single, usually required instruction in the exit path of a loop could be eliminated.

Sheldon Best invented methods for optimizing the use of index registers based on the expected frequency of execution of various parts of the program. As of 1970 there were no known provably optimal algorithms for the problem he dealt with; his methods were the basis of many subsequent storage-allocation algorithms and produced code that is very difficult to improve. (For more details of Best's methods see [4, pp. 510–515].)

The result of the optimization efforts of the FORTRAN group and particularly of the pioneering work of Best, Nelson, and Ziller was a level of optimization of object programs which was not to be found again in subsequent compilers until the late 1960s.

9. Emil Post and Syntax Description

The notation for syntax description known as BNF offers another example of a development which began with a prosaic recognition of a need. After involvement with two language design efforts—FORTRAN and IAL (ALGOL 58)—it became clear, as I was trying to describe IAL in 1959, that difficulties were occurring due to the absence of precise language definitions. In a recent course on computability given by Martin Davis, I had been exposed to the work of the logician Emil Post and his notion of a "production." As soon as the need for precise description was noted, it became obvious that Post's productions were well suited for that purpose. I hastily adapted them for use in describing the syntax of IAL. The resulting paper [5] was received

* These were S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, I. Ziller (IBM), and R. Nutt (United Aircraft).

with a silence that made it seem that precise syntax description was an idea whose time had not yet come. As far as I know that paper had only one reader, Peter Naur. Fortunately, he had independently recognized the need for precision; he improved the notation (replacing $\overline{\text{or}}$ by | and $:\equiv$ by $:=$), improved its readability by not abbreviating the names of metavariables, and then used it to describe the syntax of ALGOL 60 in the definitive paper on that language. He thus proved the usefulness of the idea in a widely read paper and it was accepted.

10. What Is a Compiler?

There is an obstacle to understanding, now, developments in programming in the early 1950s. There was a rapid change in the meaning of some important terms during the 1950s. We tend to assume that the modern meaning of a word is the same one it had in an early paper, but this is sometimes not the case. Let me illustrate this point with examples concerning the word "compiler."

In the proceedings of the 1954 ONR symposium there are a number of articles about "compilers." Three use the word in their title, and at least one other contains a description of a program called a compiler. (The articles are: "Compiler method of automatic programming" by Nora B. Moser; "New York University compiler system" by Roy Goldfinger; "The LMO edit compiler" by Merritt Elmore; and "Automatic programming on the Burroughs Laboratory computer" by Hubert M. Livingston.) As noted earlier there is also a brief description of the Laning and Zierler algebraic compiler, although it is never called a compiler but rather a "system." The most elaborate system called a compiler is the A-2 compiler described in the article by Nora Moser [6]. The NYU and Burroughs "compilers" were essentially assembly programs of a primitive type; the LMO edit "compiler" produced "editing" routines for formatting and printing output by inserting parameters from simple specifications into skeleton programs.

The Moser article on A-2 is somewhat difficult for an outsider to understand fully, but the following points emerge with reasonable certainty. (Quotes are from the Moser article [6]. Keep in mind that the A-2 compiler Moser describes in May 1954 is apparently quite different from the A-2 compiler described in material available in 1955.)

(a) ". . . The compiler method of automatic programming consists of assembling and organizing a program from . . . routines or . . . sequences of computer code which have been made up previously."

(b) There appears to be no standard "pseudocode" for this compiler; rather the problem input to the compiler is a sequence of "compiling instructions" as indicated below.

(c) "The compiling instructions for one operation include the call word

of the subroutine, the serial number, the working storage location of each argument and result of the particular subroutine. . . . A generator may require all these and in addition one or more words of specifications. . . . Hand-tailored coding begins with a sentinel and the number of lines in the sequence, followed by the coding itself."

(d) A routine, "the Translator, permits many compiling instructions to be written in abbreviated form, where one word replaces up to seven words."

(e) Apparently the user had to assign absolute working storage locations manually to all symbolic and "relative" addresses he used.

(f) The "abbreviated (compiling) instructions" are apparent forerunners of "pseudocode" (a term nowhere used in the article); however, the article suggests that they are a recent and less-than-major item in the system.

(g) In addition to inserting addresses and parameters into coding, replacing relative and symbolic addresses by programmer-assigned values and translation of "abbreviated instructions" into their "complete form," the compiler "segmented" the object program to fit into the available storage space and arranged to call in the next segment, performed various checks, and searched and updated library tapes.

The above items give some idea of what the word "compiler" meant to one group in early 1954. It may amuse us today to find "compiler" used for such a system, but it is difficult for us to imagine the constraints and difficulties under which its authors worked. After studying the seven pages of the A-2 article, it is startling to find in the same volume three scant pages devoted to Laning's algebraic system with its elegant source language and its use of combined compilation and interpretation. Oddly enough, Laning and Zierler's abstract for their report calls their system "an interpretive program," and "compilation of closed subroutines" is the closest they come to using the word "compiler."

By 1955, the A-2 compiler had acquired a definite set of fixed format "pseudo-instructions" not unlike those of earlier interpretive systems in form but with more sophisticated operations added, such as "repeat."

I have tried to assess the A-2 compiler of early 1954 on the basis of a single, not-too-clear article. I realize it is possible to have misjudged it. Much of the difficulty appears to come from changes in the system between early 1954 and 1955.

REFERENCES

1. Wilkes, M. V., Wheeler, D. J., and Gill, S., "The Preparation of Programs for an Electronic Digital Computer." Addison-Wesley, Reading, Massachusetts, 1957. (First edition published in 1951.)
2. *Proc. Symp. Automatic Programming Digital Comput., Office of Naval Research, Washington, D.C.* 13–14 May 1954).

3. Laning, J. H., and Zierler, N., A Program for Translation of Mathematical Equations for Whirlwind I. Engineering Memorandum E-364, Instrumentation Lab., MIT, Cambridge, Massachusetts (January 1954).
4. Cocke, J., and Schwartz, J. T., "Programming Languages and Their Compilers," Preliminary notes, second revised version. Courant Inst. of Math. Sci., New York Univ. (April 1970).
5. Backus, J. W., "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference, *Proc. Internat. Conf. Inf. Proc., UNESCO, Paris* (June 1959).
6. Moser, N. B., Compiler method of automatic programming, *Proc. Symp. Automatic Programming Digital Comput.* Office of Naval Research, Washington, D.C. (May 13–14, 1954).

IBM RESEARCH LABORATORY
SAN JOSE, CALIFORNIA