world more suitable to provide the opening talk on a conference of the history of programming languages. In many ways she embodies within her own person the entire history of programming languages right down to modern times, including her supervision of the development of the test validation routines for COBOL while she was in the Navy.

Now, when I talked to her about the introduction, she asked me to introduce her merely as "the third programmer on the first large-scale digital computer, Mark I," and not to say anything more because she wanted me to end with that. So in order to comply at least with that part of it, and with very deep personal admiration for the numerous contributions that Grace Hopper has made to this field, and as—in my view at least—the key person in the early history of programming languages, I want to introduce the third programmer on the first large-scale digital computer.

KEYNOTE ADDRESS

Grace Murray Hopper

Thank you very much. That gives me a nice opportunity to point out to the entire audience that the first large-scale digital computer in the United States was a *Navy* computer, operated by a *Navy* crew during World War II. There's a certain upstart service that tries to claim credit for computers nowadays, and they didn't even exist yet! [Applause]

Since this talk is being taped, I am forced therefore to remind you that nothing I say represents the plans and policies of the Department of the Navy or the Naval Service at large. Everything I say is purely the opinion of the speaker!

I'm appalled at you, in a way. You're all "Establishment." And I think I spent 20 years fighting the "Establishment." In the early years of programming languages, the most frequent phrase we heard was that the only way to program a computer was in octal. Of course a few years later a few people admitted that maybe you could use assembly language. But the entire establishment was firmly convinced that the only way to write an efficient program was in octal. They totally forgot what happened to me when I joined Eckert–Mauchly. They were building BINAC, a binary computer. We programmed it in octal. Thinking I was still a mathematician, I taught myself to add, subtract, and multiply, and even divide in octal. I was really good, until the end of the month, and then my checkbook didn't balance! [Laughter] It stayed out of balance for three months until I got hold of my brother who's a banker. After several evenings of work he informed me that at intervals I had subtracted in octal. And I faced the major problem of living in two different worlds. That may have been one of the things that sent me to get rid of octal as far as possible.

In putting material together, I go back to the days before there were programming languages, and I think I have to remind you of a few of the things that started all of this. I have here a copy of the manual for Mark I. I think most of you would be totally flabbergasted if you were faced with programming a computer, using a Mark I manual. All it gives you are the codes. From there on you're on your own to write a program. We were not programmers in those days. The word had not yet come over from England. We were "coders."

On the other hand, we did do some things to help us get programs written. And I managed to find another piece of paper which I find quite interesting because I find I have a relative subroutine in my hands which was written in 1944.

Mark I had built-in programs for sine, cosine, exponential, arctangent, ... On the other hand, because they were wired into the machine, they had to be completely general. Any problem that we solved, we found we did not need complete generality; we always knew something about what we were doing—that was what the problem was. And the answer was—we started writing subroutines, only we thought they were pieces of coding. And if I needed a sine subroutine, angle less than $\pi/4$, I'd whistle at Dick and say, "Can I have your sine subroutine?" and I'd copy it out of his notebook. We soon found that we needed just a generalized format of these if we were going to copy them, and I found a generalized subroutine for Mark I. With substitution of certain numbers, it can be copied into any given program. So as early as 1944 we started putting together things which would make it easier to write more accurate programs and get them written faster. I think we've forgotten to some extent how early that started.

The next thing that brought that to the fore was the building in England of the first computer over there and the book by Wheeler, Wilkes and Gill. They designed a set of subroutines before they built the machine. And they had a library of subroutines for that computer before it was implemented, and there was for the first time an available set of mathematical subroutines which were more or less standardized in the way you put data into them and the way you got data out of them, and made it possible to begin to write programs a *little* more rapidly. We didn't have ACM until 1947. Even then it didn't start publishing journals. There was very little communication throughout the industry unless you knew somebody. And in a way what I'm telling you comes from the basement of Croft Laboratory, and a very old factory building in North Philadelphia where we started work on UNIVAC I.

Those were precarious days. We used to say that if UNIVAC I didn't work, we were going to throw it out one side of the factory, which was a junk yard, and we were going to jump out the other side, which was a cemetary! There was little communication unless you knew someone; there were few places that we met: the University of Pennsylvania, Michigan, Harvard, MIT. Gradually we got to know each other a little bit, but there was very little communication . . . and very little publication, in the sense that you know it today. Therefore much of the interchange of information lay among the people you knew, and the places you went to.

The beginnings of ACM: we began to meet each other. There was a session at Harvard in 1948 which, I think, had the first session which began to cover something about the problems of programming. In fact there was a, . . . well, they had a whole long session on numerical methods and suggested problem solutions. There was also a shorter session, four papers—"Sequencing, Coding, and Problem Preparation." And I think that probably was one of the first sessions that was ever held on "What are we going to do about programming computers?"

The next group that stepped in to try to pull together the people in these fields and to get them to communicate with each other so that we could perhaps do a better job was the Navy. There was a series of three seminars held by the Navy on what was then called Coding, Automatic Programming, and various other names. ("Software" hadn't appeared yet.) And the first of those was held in 1951. I think most of us began to know each other at that time, and began to—even if we had no other way of communicating—would send to each other copies of what we were doing. And that was the beginning of the communications in this area.

This is a seminar on Data Handling and Automatic Computing, and it was held in March

of 1951. There are some papers which verge on the question of programming, but as yet, nothing that even resembles a real assist to writing programs. I think the first two things to appear in this area that influenced me most were John Mauchly's "Short Order Code" and Betty Holberton's "Sort–Merge Generator."

John Mauchly's "Short Order Code" operated on a BINAC in 1949. It turned the BINAC into a floating decimal mathematical computer. He wrote words, and the first word might represent the X; there was a sign for equals, and sign for a, a sign for + and a sign for b—because BINAC was a totally numerical computer. It had no alpha, so it was necessary to use two digits to indicate these various symbols. It was written in true algebraic format, but since there was no alpha, it didn't look very algebraic. But these pairs of digits did match the algebraic expressions. It was an interpretive system; the subroutines were stored in the memory and reference was made to the subroutines by the symbolic code which you put in.

I think this was the first thing that clued me to the fact that you could use some kind of a code other than the actual machine code. In other words, if you will, a "simulated computer" I guess we'd call it nowadays. To us it was a pseudo-code. It was not the original code of the computer, but it was a code which was implemented through a program. And I think Short Code was the first step in moving toward something which gave a programmer an actual power to write a program in a language which bore no resemblance whatsoever to the original machine code.

Short Code operated on BINAC in 1949. It was later transferred, reprogrammed, and used on UNIVAC by 1952. And it did constitute both a set of subroutines and a method for writing something in a pseudo-code. However, it was interpretive in that the program looked at each statement, jumped to the called-for subroutines, executed them, and went back and got the next statement. It did not write out an actual program.

I think the first step to tell us that we could actually use a computer to write programs was Betty Holberton's "Sort–Merge Generator." You fed it the specifications of the files you were operating on, and the Sort–Merge Generator produced the program to do the sorting and merging, including not only carrying out the operations, but also managing all of the input and output in the various tape units, and it contained, I think, what I would define as the first version of a virtual memory in that it made use of overlays automatically without being told to by the programmer. I think that meant a great deal to me. It meant that I could do these things automatically; that you could make a computer write a program.

Of course, at that time the Establishment promptly told us—at least they told me quite frequently—that a computer could not write a program; it was totally impossible; that all that computers could do was arithmetic, and that it couldn't write programs; that it had none of the imagination and dexterity of a human being. I kept trying to explain that we were wrapping up the human being's dexterity in the program that he wrote, the generator, and that of course we could make a computer do these things so long as they were completely defined. We did run up against that problem. I think I can remember sometime along in the middle of 1952 that I flatly made the alarming statement that I could make a computer do anything which I could completely define. I'm still of course involved in proving that because I'm not sure if anybody believes me yet.

But the Sort–Merge Generator gave rise to a whole family of other generators. I could wish that many of you today would go back and look at that Sort–Merge Generator. I think you might find the solution to some of today's problems. I think we've gotten far

away from the concept of generator; with the mini- and microcomputers and the systems of computers, if we're going to dedicate them, we need more powerful generators. I'd love to have a good data handling generator, and I hope somebody's going to write one sometime during this year, to which I can feed specifications and receive as output the required program to do the job I have specified. I don't mean languages. I don't mean compilers. I mean generators for specific jobs.

The Navy sessions meant a great deal. The next one—the first one was 1951—the next one was 1954. I'll wait a minute for that. In the fall of 1951, October, I received an assignment to build a set of mathematical subroutines for UNIVAC I, to sufficiently standardize them so that everybody would be able to use them. As I went along in the process of standardizing the subroutines, I recognized something else was happening in the programming group. We were using subroutines. We were copying routines from one program into another. There were two things wrong with that technique: one was that the subroutines were all started at line 0 and went on sequentially from there. When you copied them into another program, you therefore had to add to all those addresses as you copied them into the new program—you had to add to all those addresses. And programmers are lousy adders!

The second thing that inhibited this was that programmers are lousy copyists! And it was amazing how many times a 4 would turn into a delta which was our space symbol, or into an A—and even Bs turned into 13s. All sorts of interesting things happened when programmers tried to copy subroutines. And there of course stood a gadget whose whole purpose was to copy things accurately and do addition. And it therefore seemed sensible, instead of having programmers copy the subroutines, to have the computer copy the subroutines. Out of that came the A-0 compiler.

Between October of 1951 and May of 1952 I wrote a compiler. Now, there was a curious thing about it: it wasn't what you'd call one today, and it wasn't what you'd call a "language" today. It was a series of specifications. For each subroutine you wrote some specs. The reason it got called a compiler was that each subroutine was given a "call word," because the subroutines were in a library, and when you pull stuff out of a library you compile things. It's as simple as that.

The program was a program which, given a call word, pulled a particular subroutine. Following the call word were the specifications of the arguments and results which were to be entered into the subroutine with no language whatsoever. The programs were butted, one bang against another. There was no attempt, really, at optimization. Because it was designed to let people write quickly the specs for a mathematical program, one time usually—one-time execution, and get an answer fast. And the main purpose was to get programs out fast, and get answers fast.

One of the reasons that we had to write programs faster, I think some people have forgotten. Mark I did three additions every single second, 23 digits to 23 digits. If you go back and look at the newspapers and magazines, you find she was the most fantastic thing man ever built. By the time we got to UNIVAC I, it was doing 3,000 additions every second, and it was chewing up programs awful fast. And the programmers had not accelerated in an equal degree! [Laughter] It was perfectly clear that we were going to have to produce programs faster.

There was also the fact that there were beginning to be more and more people who wanted to solve problems, but who were unwilling to learn octal code and manipulate bits. They wanted an easier way of getting answers out of the computer. So the primary pur-

poses were not to develop a programming language, and we didn't give a hoot about commas and colons. We were after getting correct programs written faster, and getting answers for people faster. I am sorry that to some extent I feel the programming language community has somewhat lost track of those two purposes. We were trying to solve problems and get answers. And I think we should go back somewhat to that stage.

We were also endeavoring to provide a means that people could use, not that programmers or programming language designers could use. But rather, plain, ordinary people, who had problems they wanted to solve. Some were engineers; some were business people. And we tried to meet the needs of the various communities. I think we somewhat lost track of that too. I'm hoping that the development of the microcomputer will bring us back to reality and to recognizing that we have a large variety of people out there who want to solve problems, some of whom are symbol-oriented, some of whom are word-oriented, and that they are going to need different kinds of languages rather than trying to force them all into the pattern of the mathematical logician. A lot of them are not.

In any case, I started out to devise a means by which I could put these subroutines together to produce a program. Curiously, it did not occur to me that I should make two sweeps over it. I felt that it should be a single-pass compiler. In fact, the information defining a problem came in on one tape unit, and the program was written out on another one, because UNIVAC I only had 1000 words of storage and there wasn't room enough to keep a whole lot of junk in there while I was doing the compiling process.

I promptly ran up against the problem that in some cases, after making a test, I would jump back in the program for something I had previously processed and at other times I would jump forward in the program to a section of the program which had not been written and I did not know where it was. In other words, there were two types of jumps to be coped with: one which went back in the program; the other went forward in the program. Therefore, as the program was put together, a record was kept of where each subroutine was: they were numbered; the operations were numbered. And if I wanted to jump back to Operation 10, I could look at the record and find which line Operation 10 was at. But if I was at Operation 17, and I wanted to jump to Operation 28, I didn't yet know where it was.

And here comes in the curious fact that sometimes something totally extraneous to what you are doing will lead you to an answer. It so happened that when I was an undergraduate at college I played basketball under the old women's rules which divided the court into two halves, and there were six on a team; we had both a center and a side center, and I was the side center. Under the rules, you could dribble only once and you couldn't take a step while you had the ball in your hands. Therefore if you got the ball and you wanted to get down there under the basket, you used what we called a "forward pass." You looked for a member of your team, threw the ball over, ran like the dickens up ahead, and she threw the ball back to you. So it seemed to me that this was an appropriate way of solving the problem I was facing of the forward jumps! I tucked a little section down at the end of the memory which I called the "neutral corner." At the time I wanted to jump forward from the routine I was working on, I jumped to a spot in "neutral corner." I then set up a flag for Operation 28 which said, "I've got a message for you." This meant that each routine, as I processed it, had to look and see if it had a flag; if it did, it put a second jump from the neutral corner to the beginning of the routine, and it was possible to make a single-pass compiler and the concept did come from playing basketball! [Applause] I have a feeling that we would do well to train our computer people and specialists in many disciplines rather than in a single discipline because you never know what may prove useful. How-

ever, I'm afraid that that was the only compiler that was ever built that was a single-pass compiler.

Something rather interesting about it, though. At the time that it was written up, it was pointed out that there were some other things we might be able to do, and it was even suggested that it might—in line with the "anything which I can completely define"—it was pointed out that it might be possible even to build a differentiator.

With these concepts—incidentally, I have here the documentation of that A-0 compiler. It interests me today because of the failure of most of our people to document. In fact, I've probably never seen anything quite as completely documented as this is. In the first place, there's a write-up of the compiler and how it's going to work: the neutral corner, and all the rest of it. There is then a block chart of what it's going to do at a high level; a flow chart at a detailed level. The coding is here. Along with the coding, there are comments on it. The coding and the flow charts are keyed together with little circular things in the flow charts. And following that, there is a line-by-line description of exactly how the compiler works, in English, keyed both to the flow charts and to the coding. All this was done because I knew damn well nobody was ever going to believe it was going to work! And probably maybe that's one of the best reasons for documentation that ever existed—when you have to convince somebody that the darn thing will work.

It was running in May of 1952, and as Jean told you, the paper was given at the ACM meeting in Pittsburgh in May of 1952. A curious thing happened there. We did not put out proceedings in those days. I hauled 200 copies of the paper from Philadelphia to Pittsburgh on a Convair, and put them out at the back of the room so that people could pick them up as they wanted to, and I think I hauled 100 of 'em back again, which shows you the major interest that people expressed in trying to get some easier ways of writing programs.

To me the basic argument was that we were using subroutines which were checked out and known to work; we were putting them together with an automatic program which had been pretty well checked out too, and that only left the programmer the possibility of making major mistakes in logic.

In that same year of 1952, November of 1952, came Millie Coss's Editing Generator. One of the major problems in those early programs had been editing. You had to pull apart words and stick periods in, and put on dollar signs, and suppressed zeros, and much of it was done by hand coding. This seemed something which certainly—we knew exactly what was in the computer; we also knew exactly the format we wanted it to look like—that it would be possible to build a generator which would build an editing routine. Millie Coss went to work and in November of 1952 had operating a program which took in the format of the file, the format of the records, and the format of the output printed document, and produced the coding to get from one to the other. It also did a little bit of summing and things, and I think you could well call it the ancestor of most of the RPGs, and of those particular techniques.

Meantime, the long and cumbersome way of writing the input specifications to the A-0 compiler were becoming apparent, and it seemed much better to have a shorter way of writing that stuff. Now, I'll remind you that back in those days we were coding for computers. The concept of doing anything else had not occurred to us. Therefore, when we invented a new way of writing those programs, we used a code of course, and in fact, used a three-address code. Another thing drove us in this direction, and I think we often forget how much our surroundings influence our research decisions. All those things that we're aware of and yet are not aware of when we make decisions. And the fact was that we were

living in an environment which consisted of a 12 alpha-decimal word, and it became perfectly obvious to us that the entire world operated in 12 alpha-decimal characters of which the first three defined an operation, the next three one input, the next three another input, and the last three the result. And it was clear that the world operated this way, and that the world lived in 12 alpha-decimal characters, and a three-address machine code.

That therefore was the code which was imposed on top of A-0. A translator was written, went on the front of A-0 which implemented a three-address machine code, and this became the A-2 compiler. I have here the original code for it and the write-up of the compiler. One interesting thing was that this combined the concepts of compiling and of generating, because when it came to compiling input and output routines, they became voluminous; there were so many of them, and just a few that did the arithmetic. And it was at this time that the I/O generators were written by Jim McGarvey, and the compiler actually contained generators which produced the input–output handling of the data.

Something else happened which curiously influenced and, if you will, slowed down the development of the languages from our point of view, in that that code proved to have some amazing virtues, and this is the reason I asked for a blackboard here. I'm going to try and show you what can happen to you when you become convinced that the whole world operates in a three-address code.

For instance, add x to y to give z. [ADD 00X 00Y 00Z]

Never mind there are some extra zeroes in there. Sometimes we needed those. Multiply z by t to give w. [MUL 00Z 00T 00W] Obviously, a beautiful way to write programs. This was $x + y = z$. This was $zt = w$.

After that astounding remark that you could make a computer do anything which you could completely define, we felt impelled to do something about differentiation. We invented a new definition. We said if we put a number in front of a letter, it means the differential. 01X is dx. 15X obviously then is $d15$ of x. Perfectly permissible thing to do. All we did was leave the "d" out.

If a number follows—A12, A13, that just denotes an auxiliary storage location. And we found we could create a differentiator. What it did was pick up the first three letters, jump to a task routine; if it hit ADD it simply added ones, and it wrote down "ADD 01X 01Y 01Z" and it had nicely written "$dx + dy = dz$" which was totally correct.

When it got to multiply, it had to get a little cleverer. The task routine wrote down "MUL 00Z"—it added a 1 "01T"—moved to the next auxiliary storage location—"A14" "MUL 01Z, 00T, A15, ADD A14 and A15—you get 01W; and we'd written $z\,dt + t\,dz$ is equal to dw"—and the darned thing worked! And it was as silly a trick as that—to write the number in front of the letter.

```
MUL  00Z  01T  A14
MUL  01Z  00T  A15
ADD  A14  A15  01W
```

Of course as it kept on repeating the application of this, it got messier. It was finally given to Harry Kahrimanian, and in May of 1953, Harry put out his thesis at Temple University, "Analytical Differentiation by a Digital Computer." He had completed the work, extending from this original definition and using the A-2 compiler with its three-address code, had developed a differentiator.

Of course everybody promptly said that it couldn't possibly work because computers

could only do arithmetic. They could not do calculus. That was very discouraging to Harry! [Laughter] Of course, but I always wanted to explore, to go a little bit further. We'd only added 1s here. We played with this on paper and on the computer, and we discovered we could get negative derivatives; we could also take *d* to the 1.08 derivative, and all sorts of interesting things like that. And nobody has yet told me what fractional derivatives are, so I hope somebody will explain that to me someday, but I can compute 'em under this definition! [Laughter] There are still some things to be explored from the ancient days.

With the A-2 compiler we began to try to get people to use these things. We found that we had to change from being research and development people and turn ourselves into salesmen, and get out and sell the idea of writing programs this way to customers. DuPont was the first to try A-2. It spread from there to the David Taylor Model Basin, I think, assisted by Betty Holberton and Nora Moser. To the Army Map Service. To the Bureau of the Census. And people began to try this game out. To NYU—it began to spread little bit by little bit to people who wanted to find a more rapid and easier way of writing programs.

The selling was an extremely difficult job because it had to involve not only selling the users, the customers, but also selling management, and persuading them that it was necessary for them to invest the time and money in building these program-assisting routines. And that was almost as difficult, if not more so, than selling to the users.

In July of 1953 John Mauchly prepared a paper on the influence of programming techniques on the design of computers, and it was published in the Proceedings of the IEEE. [IRE?] I think it was one of the first times that we tried to reach the entire community to say, "Look, we are going to be doing this kind of thing." And he mentions in the paper the need for assistance in designing computers. He brings up the Mark I and Mark II, brings up the fact that even when a standard subroutine program for full computer precision was to be used, it had to be copied with appropriate changes. He mentions the coding machine of Mark III. And the coding machine of Mark III was, of course, one of the things that led me to believe you could make programs somehow. The Wheeler, Wilkes, and Gill book. And then he goes on to the beginning of these generators. The Sort-Merge Generator. The Editing Generator. By then, Speedcoding had appeared with the IBM 701. The unwinding of programs and getting the computer to write out a complete program instead of having it all wound up in little knots.

The world began to look at the fact that maybe it was possible some way to provide assistance in developing programs. One of the other things I have here—which I will not go through for you except to mention what had to be done—this is a report written in December 1953, and it's a plea for a budget. All of that. The first two pages, about, are the plea for the budget. The rest of it is back-up documentation of what was happening in the world at that time.

I think it's interesting to find that the interpretive routines are mentioned. "Interpretive routines have been developed at many installations, notably Whirlwind, MIDAC, ILLIAC, SEAC, and IBM 701 Speedcoding." Interpretive routines were growing. It also gives the argument as to why we should be doing—it defines "interpreter" and "compiler"—and then it further gives the argument of why we should write programs instead of continuing to interpret at each step, because of course then the interpretive concept was the more popular one.

There were two papers that went out that year—September of 1953, Dick Ridgway's

paper in Toronto, and bit by bit we were beginning to try and get the word out. July of 1953 was the first workshop sponsored by the Bureau of Census in Washington. By late 1953 we had the differentiator, the editing generator. By then A-2 had been successfully used at the Army Map Service, the Air Controller, Livermore, NYU, the Bureau of Ships, the David Taylor Model Basin. People were beginning to have some interest in the possibility of providing some assistance for writing programs. This even contains the letters—there were testimonial letters that were written saying, "Yeah, we used the compiler, and we find it useful, and it saved us some time." They're almost naïve to look at today, and yet they tell you a little bit about an atmosphere which was totally unwilling, or seemingly so, to accept a new development, because it was obvious that you always wrote your best programs in octal.

Probably the biggest step that year were two simultaneous developments. One was the beginning of FORTRAN. I have a copy here of a treasure which John Backus sent to me at the time: "Preliminary Report—Programming Research Group, Applied Science Division, International Business Machines Corporation, November 10, 1954. Specifications for the IBM Mathematical FORmula TRANslating System—FORTRAN" and the beginning of the true programming languages.

Simultaneous with that, at MIT, Laning and Zierler had written their algebraic translator. I've heard some people say that one led to the other. As far as I can tell, they were close to simultaneous, and neither one led to the other. In no case could it be possible that FORTRAN followed Laning and Zierler because this could not have been written between the time Laning and Zierler published and the time this was published.

It might interest you to look sometime at the original FORTRAN specs, though I suspect John will do his own job of educating you on that, but this was a tremendous step. This was the first of the *true* programming languages.

And yet I doubt if we could have gotten that far had we not had before that the generators, the compilers, the tools with which we would eventually implement the languages. I find today that many people are concentrating on the languages, and not doing very much about the tools with which to implement those languages. The art of building compilers, as far as I'm concerned, has been pretty badly stultified. There are excellent ways of building compilers for FORTRAN and ALGOL and Pascal and all the rest of the mathematical, logical programs, and then people try to use those methods to build a COBOL compiler, and it does not work. They are different, and the languages are different. And the distinction has not been made between the scientific engineering work, the mathematical work, and the data processing work. And the fact remains that in many ways they are different.

A little later, from Univac came AT-3. A-3 again was an extension of A-2, taking the 12-character, three address machine code, and extending it to a mathematical language. It's [AT-3] similar to FORTRAN in some cases; different in others. One of the things it did, which has never been done since, was to use exponents. At that time we prepared programs by typing on a typewriter onto a tape. Back in the early days of Mark I there was a gadget called a Unityper and you typed on tape. Of course, it disappeared and we had to reinvent that wheel some years later.

Since you could type directly on the tape, and since the Remington Rand typewriters had changeable type, we simply changed the type on the typewriter and decided that pulse codes mean what we say they mean. And we typed out stuff which looked like true mathematics: it had exponents and subscripts, and pulse codes meant what we said they mean. I don't think more than about 10 or 12 typewriters were modified like that, but we liked it a

lot better being able to write our mathematics and have it look like mathematics directly, and have our exponents and subscripts explicitly typed out. That was true of MATHMATIC which came out about that time.

Meantime, there was still the major difficulty of the data processors. Working with the people that I had worked with in data processing, I found that very few of them were symbol-oriented; very few of them were mathematically trained. They were business trained, and they were word-manipulaters rather than mathematics people. There was something else that was true of the data processing world: whereas in the mathematical–engineering world, we had a well-defined, well-known language; everybody knew exactly what—if you said "SIN" everybody knew it meant "sine"—they knew exactly what you meant, whether you were in Berlin or Tokyo, or anywhere in between. It was fully defined, already there. Everybody knew it. But there wasn't any such language for data processing. We took something over 500 data processing programs and sat down to write out what people were doing when they did data processing. We finally identified about 30 verbs which seemed to be the operators of data processing.

That December 1953 report proposed to management that mathematical programs should be written in mathematical notation, data processing programs should be written in English statements, and we would be delighted to supply the two corresponding compilers to translate to machine code. I was promptly told that I could not do that. And this time the reason was that computers couldn't understand English words. [Laughter] Well, I allowed as to how I never expected any computer anywhere to understand anything; that all I intended to do was to compare bit patterns. But it was not until January of 1955 that we were able to write a formal proposal for writing a data processing compiler. I keep it with me for a couple of reasons. One, to remind me that I always have to push into the future; and the other to remind me that any given moment in time there's always a line out here, and that line represents what your management will believe at that moment in time. [Laughter] And just you step over the line and you don't get the budget. I'd like to illustrate that:

This is the preliminary definition of a data processing compiler, dated 31 January, 1955. The language—the pseudo-code—is to be variable-length English words separated by spaces, sentences terminated by periods. One of the first things we ran into was nobody believed this. So we adopted an engineering technique. It's one that I wish more people would adopt. I'm surprised that we don't use it more often. When an engineer designs or builds something, he almost always builds either a pilot model or a bread board. He proves feasibility, he uncovers any difficulties he hasn't thought up theoretically, and he gets valid cost estimates. Best reasons in the world for building a pilot model.

We decided we'd build a pilot model of the compiler. And we started to work. Now, in the early compilers, there had been automatic segmentation and overlay. There had to be because of the size of UNIVAC I with only 1000 words of storage. Of course, we didn't call it by an interesting name. We just called it "using auxiliary storage." When it came to writing this data processing compiler, the first thing we found was that we were going to have to keep some lists. An operation list, a subroutine list, a jump list, and a storage list. Of course we quickly realized that management wouldn't know what a list was, and we changed that word to the word "file" and kept "files" so they'd understand all right.

The next thing that happened to us was that those got longer and longer and longer. And we only had 1000 words of storage. We had to do something, so we decided to use auxiliary storage. We tucked a small routine in the corner of the main memory, and if I didn't

need a list for a while, I'd say, "Here's the Op File." It would grab it, put it on a tape unit, remember where it put it, and when I needed it again, I'd say, "Give me the Op File"—it's back in main memory, and I could work on it again. We did the same thing with the segments of the program in that model compiler. For instance, the heart of that compiler, the actual code generator, was called "DUZ". It was called DUZ because back in those days before we worried about phosphorus, DUZ did everything. [DUZ was a brand of soap with the slogan "DUZ does everything."] And if I didn't need DUZ for a while, I'd say, "Here's DUZ." It'd put it somewhere, and when I wanted it again, I'd say, "Give me DUZ"—it was back in the main memory, and I could work with it again.

Burroughs started using that technique about 20 years ago. I guess it was eight or nine years ago RCA reinvented it and called it "virtual storage." But it didn't really get to be respectable until IBM discovered it about five or six years ago. [Laughter and applause]

We got our little compiler running. It wouldn't take more than 20 statements. And on the back of this report, we put a nice little program in English. And we said, "Dear Kind Management: If you come down to the machine room, we'll be delighted to run this program for you." And it read: INPUT INVENTORY FILE A; PRICE FILE B; OUTPUT PRICED INVENTORY FILE C. COMPARE PRODUCT #A WITH PRODUCT #B. IF GREATER, GO TO OPERATION 10; IF EQUAL, GO TO OPERATION 5; OTHERWISE GO TO OPERATION 2. TRANSFER A TO D; WRITE ITEM D; JUMP TO OPERATION 8. It ended up: REWIND B; CLOSE OUT FILE C AND D; and STOP.

Nice little English program. But the more we looked at it, the smaller it looked. And we were asking for the biggest budget we had ever asked for. So we decided it would be a good idea if we did something more. We wrote a program. It went into the compiler and located all the English words and told us where they were, and we then replaced them. We said, "Dear Management: We'd also like to run this program for you. [Program spoken in French.] We'd like to run this French program for you." I don't know whether you recognized it, but the words are exactly the same and are in the same positions.

Well, if you do something once, it's an accident; if you do it twice it's a coincidence; but if you do it three times, you've uncovered a natural law! [Laughter] So—that's why they always tell you to "Try, try, try again." We changed the words again; the third program was an impressive one, of course. [Program in German language this time. With laughter] Beautiful words in this one! "We'd love to run this program for you."

Have you figured out what happened to that? That hit the fan!! It was absolutely obvious that a respectable American computer, built in Philadelphia, Pennsylvania, could not possibly understand French or German! And it took us four months to say no, no, no, no! We wouldn't think of programming it in anything but English.

What to us had been a simple—perfectly simple and obvious—substitution of bit patterns, to management we'd moved into the whole world of foreign languages, which was obviously impossible. That dangerous, dangerous point of something that's obvious, evident to the researcher, to the programmer—when it's faced by Management, is out of this world. It was nothing but a substitution of bit patterns, yet to management it was a move into all the foreign languages. The programs were identical, nothing but the bit patterns for the words changed. Incidentally, in building that compiler, we did something else—this is FLOW-MATIC. The A series was the mathematical compiler; the B series were to be the business compilers; it was those doggone sales people that wanted a fancier name and named them such things as MATH-MATIC and ARITH-MATIC and FLOW-MATIC. You can't do anything about the sales department, you just have to let 'em go ahead. But

there was something we did in building this compiler that I find rather interesting, particularly since it survived so long. In order to quickly pick up the word—we didn't know anything about parsing algorithms at that point in time—and what happened was you picked up the verb, and then jumped to a subroutine which parsed that type of sentence. In order to do that quickly, and also to make it easy to manufacture that jump, the first and third letters of all the verbs in FLOW-MATIC were unique. That survived until last year in COBOL: that the first and third letters of all the verbs were unique. So, two things influenced FLOW-MATIC over and beyond mathematics and beyond programming languages. One was that we wanted to write the programs in anybody's language, which meant all the statements had to be imperative statements, because that was the only type of sentence in German that began with a verb. [Laughter] And the second was that we had to select the verbs, so that the first and third characters would be unique. One of our most difficult situations was of course display and divide, and that's why it had to be first and third characters. And there are certain of the other verbs which ran into that problem as well.

The data descriptions in that case weren't informal. We had forms, and you entered the data description in it. They were formalized. Frankly, I like the formalized things much better because I love filling in squares in answering questionnaires and things, and I always found it much easier to use a fixed format than I ever did to use an indefinite format, particularly because if you have a fixed format, you can always have the period be there; whereas if it's indefinite, then you have to remember to put the period on the end of the sentence, and that's one of my major difficulties.

To put that program—that compiler, on a 1000 word computer—perhaps I can appall you by what it meant. This is the chart of the tape operations. Did you ever see so many tapes spinning around? It took almost two hours sometimes to compile a program, because all the tape units were very busy spinning around at high speed, and yet that compiler was put on a 1000-word computer. Nowadays when people try to tell me that you can't put things on the minis or the micros, I have a tendency to pull some of these things out and say, "Of course you can! Let's not be ridiculous." 64K is a heckuva lot more memory than we had on 1000 words. A thousand 12-character words was only 12K, and all of FLOW-MATIC, and MATH-MATIC fitted in those memories. We can do a lot more with those micro-computers than we think we can. I think we've forgotten the size of the problems that those early computers tackled. UNIVAC I handled the whole premium file for Prudential Life. It did the entire payroll for U.S. Steel, including group incentive pay. It just did it slower, that's all. Now we have all the speed of the micros today.

When I look at the world today, what I see is a world in which I can have 300 UNIVAC I's all in one room, with all the power of those early computers. So, in a way, I'd ask you to look back at some of the tremendous things those early programmers did.

We finally convinced the Marketing Department that this thing could work. You might be interested later in looking at the first brochure that went out to try and sell to the general public the idea of writing data processing programs in English statements. It was a long, tortuous, and difficult job to get that concept accepted, because it was of course obvious that computers couldn't understand plain English, which made life very very difficult.

I also have, just in case anybody'd like to look at it, the list of the original verbs; the format with which they were defined, which incidentally, has survived through into

today's COBOL compilers, COBOL manuals; and the specifications that started out the world in data processing of writing programs in plain English.

Now, of course, I still think we can go too far, and I would point out one thing about that language that implemented FLOW-MATIC: the fact that we could substitute those French and English words means that it was never anything but a code. It was not a language. Because the fact that we could take the words of those statements and substitute French words, German words, or anything else, meant that it was not a true language. It was actually a code.

I think sometimes we've lost track of the fact that we can make simpler languages if we stick to the format of a code of some kind and the exact definition provided by a code.

One other thing that came true from that particular language: by now the computers were going even a thousand times faster, and there was even greater demand for writing programs. Continuing on the subject of how the compilers were built, I think it was interesting that we used FLOW-MATIC to write our first COBOL—that it was possible to use it to write the first COBOL compiler. In between, something else had happened, and another unique compiler was produced, a compiler than ran on two computers.

The Air Materiel Command at Wright–Patterson wanted to write English language programs for their 1103. They had been writing English language programs on their UNIVAC. They wanted to write them on the 1103. A compiler was produced at Wright–Patterson, AIMACO, by the Air Materiel Command. First, you ran it on the UNIVAC I which translated the English into UNICODE, which was the assembly code of the 1103; it then picked the UNICODE off the UNIVAC I and walked it over and put it on the 1103, which produced the final program. And I think it's probably the only compiler I ever knew which worked halfway on one computer and halfway on a second computer, and it was done that way in order to build the compiler very very rapidly, and to make use of two existing pieces of software; the FLOW-MATIC on the UNIVAC I and the UNICODE already existing on the 1103. It's probably unique, and I don't think anything else was ever written in that particular way. But it might help us once in a while, when we need very badly to get a tool and to get it more rapidly than possible.

I was told to start talking to the days before 1956, the concentration was on meeting user needs, the concentration was not on the languages: it was on building the tools which later began to support the languages: the compilers, the generators, the supporting elements which now back up our definitions of languages. Languages came into use. People began to use 'em. There was another stage that had to occur. I think to some extent we had paid little attention to it in the early days. And that was that the implementers interpreted things differently. This particularly occurred in the case of COBOL. The Navy became very much interested in trying to make all the COBOL compilers give the same answers, even though they were on different computers. And it was that reason that I was called back and in the late 1960s at the Navy Department. A set of programs was built which would validate a COBOL compiler. They would compare the execution of the compiler against the standard, and monitor the behavior of the actions of the compiler. It was the first set of programs that was built to try to use software to check software.

I think this is an important element we've omitted. If we're going to have a language, it certainly should have the same answers on all the different computers. The set of COBOL validation routines was the first such set of routines to prove whether or not a compiler did in fact correctly implement the standards. I have the papers here that were published on

the Federal COBOL Compiler Testing System. Recently they have also produced a set of routines for testing FORTRAN.

I think this is something we overlooked in our development of our languages. We overlooked the fact that we must see to it that those compilers properly implemented the language; that they did give common answers on different computers. A language isn't very useful if you get different answers on different computers. At least it isn't to an organization like the Navy which at any given moment has at least one of every kind of computer, and we *would* like to get the same answers from the same program on the different computers.

Another thing that's been happening as people have continued to develop the programming languages is that they persist in invalidating our older programs. I'd like to add something: put a burden on the programming language people. I'd like to insist that if they add something to a language, or change something in a language, they also provide us with the algorithm which will make the transfer in our programs so that we can automatically translate our programs, say, from COBOL 68 to COBOL 74. And if they don't do something like that pretty soon, I'm personally going to go around and shoot up the members of the language defining committees, because they are providing a large number of headaches. [Applause]

As I've gone through this, you may notice that in fact it was the Navy that ordered me to that first computer, the Navy that made it possible for us to develop COBOL, the Navy that made it possible for us to develop those test routines. I'm eternally grateful to an organization known as the United States Navy which ordered me to that first computer, and which throughout the 34 years since then, has given me the opportunities and the privileges of working in the computer industry, in the world of trying to make it easier for people to write programs. It's been a fascinating 34 years, and I'm forever grateful to the Navy for it, and for the privilege and responsibility of serving very proudly in the United States Navy.

Thank you. [Long applause]

TRANSCRIPT OF QUESTION AND ANSWER SESSION

SAMMET: We have questions for you.

CAPTAIN HOPPER: I can't answer questions sitting down.

SAMMET: I can't ask them standing up, so I'll sit down. There have been a number of questions that were turned in. I've tried to organize these in a little bit of coherence. The first one has been asked by a couple of people; JAN Lee for one, Richard Miller for another. If you were the third programmer on Mark I, who were the first two?

HOPPER: Richard Milton Block, now a consultant; he was on the engineering side; and Robert Campbell. I think he's now at MITRE. They were both Ensigns. I found out after I reported, they were brand new Ensigns—90-day wonders—they had heard that this gray-haired old college professor was coming, a j.g., and Dick Block paid Bob Campbell to take the desk next to me because he didn't want to have to. [Laughter] That's what comes of Ensigns! [More laughter]