3

LEARNING FROM THE MASTERS 1963–66

Regnecentralen – Algol 60 – Peter Naur and Jørn Jensen – Dask and Gier Algol – The mysterious Cobol 61 report – I join the compiler group – Playing roulette at Marienlyst resort – Jump-starting Siemens Cobol at Mogenstrup Inn – Negotiating salary – Compiler testing in Munich – Naur and Dijkstra smile in Stockholm – The Cobol compiler is finished – Milena and I are married in Slovenia.

On January 31, 1963, I graduated from The Technical University of Denmark with a master's degree in electronic engineering. Shortly before, I started looking for my first job as an electronic engineer:

> I want to be sure I get a good job—one concerned with electronic computers, and the main thing is not my wages, but rather that I constantly learn something new. The question about what to learn is quite tricky. First I wanted to learn "everything" about computers, but lately a professor at our technical university has convinced me that there is the danger, that I will spend my most productive years merely trying to understand, what others have done, without having time to contribute anything myself. So the question comes up: When and what to specialize in? Anyhow I'm going to have a talk with the manager at our biggest computing center on January 3rd. (Letter to Milena, January 1, 1963.)

Actually, I did have some idea about my professional goals. I just didn't know, if I could pursue them in Denmark.

The only place in Denmark that developed computers was Regnecentralen, a research institution under The Danish Academy of Technical Sciences. In 1957, Regnecentralen completed the first Danish computer, Dask, in an old villa on Bjerregaardsvej 5, in Valby, a suburb of Copenhagen. Built

under the leadership of Bent Scharøe Petersen, Dask used thousands of vacuum tubes in its electronic circuits and tens of thousands of magnetic cores in its memory. It executed 18,000 instructions per second.

Only one copy of Dask was built. Weighing three and a half metric tons, it was installed in the former dining room of the villa. The oak parquet floor had to be reinforced to support this computational monster. A large cooling and ventilation system was installed in the basement.

The power supply of Dask emitted a sharp blue light that was visible from the street. An elderly lady, with a vivid imagination, complained that she felt a prickly sensation from "these electrons and atoms" whenever she walked past the villa.

In November 1961, Regnecentralen finished a small, transistorized computer, named Gier. Housed in a wardrobe-sized closet with teak paneling, Gier looked like a piece of modern Danish furniture. It had a core store of 1024 words and a drum of 12800 words (about 5K and 60K bytes). Eventually about fifty Giers were produced.

My job interview at Regnecentralen started in the Rialto Center, a new office building on Falkoner Alle 1, within walking distance of the apartment I shared with my sister in Frederiksberg.

I talked briefly with the director, Niels Ivar Bech, a charming, dynamic man, who asked me: "Where will you be in ten years?" With tongue in cheek, I said: "In your chair!" He smiled—that was the kind of answer he liked. Looking back, my answer was absurd. There was no chance that I would ever be able to replace Bech's inspired leadership. But I didn't know that at the time.

For the next six hours I had unscheduled meetings with various department heads. Whenever they realized I was looking for something else, they would suggest that I visit another department.

I spoke to Aage Melbye about administrative data processing. His people programmed some of the most demanding computer applications. The main problem was to update large files efficiently and reliably. Since drums and disks were still small, the files were stored on magnetic tapes. To avoid wasting computer time after a tape failure, it was necessary to include restart facilities in these programs. A few years later, I would gain first-hand knowledge of these problems, when I programmed the input/output system for the Siemens Cobol compiler.

My next stop was the hardware group in Valby, headed by Henning Isaksson. Two years earlier, they had finished the Gier computer. I explained

my interest in computer architecture and mentioned that I would prefer a job that would constantly teach me something new. Henning made it clear that, if he needed two flip flops, I would have to do the same thing twice. This made sense from his point of view (but not mine). I could not have predicted that Henning eventually would make my dream of becoming a computer architect come true. However, on that day, he suggested that I go back to the Rialto center and talk to the compiler group.

On the fifth floor, I met the leaders of Regnecentralen's compiler group: Peter Naur, a tall man with a serious expression and a full beard, and Jørn Jensen, a short man with a friendly smile and an unruly mop of hair. When I had explained my interest in understanding the relationship between programming languages and computer architecture, they handed me a thick yellow report with a devious smile and said: "Come back next week if you understand this." The report was entitled *Cobol-1961, Report to Conference on Data Systems Languages* (U.S. Department of Defense 1961).

James Joyce would have given the Cobol 61 report high marks for unreadability (but low marks for consistency). I did not understand a word of it. Fortunately, nobody asked me about it when I joined Regnecentralen's compiler group. To Milena, I wrote: "At last I found the right thing—a group working on advanced problems in computer languages."

⋆ ⋆ ⋆

Peter Naur was educated as an astronomer. He joined Regnecentralen in 1959 and became heavily involved in the international development of the programming language Algol 60.

The invention of programming languages is surely one of the most significant milestones in the history of computing. The science writer, Isaac Asimov (1976), put it this way:

> I strongly suspect that the advance of science or any branch of it depends upon the development of a simple and standardized language into which its concepts can be put. Only in this manner can one scientist understand another in his field.

Now, a programming language can only serve as a *standard* if it is concisely defined in a *language report*. In practice, however, most language definitions rely heavily on the reader's ability to fill in gaps and remove inconsistencies by educated guessing. I believe there is a reason for this sad state of affairs:

The task of writing a report that defines a programming language with complete clarity to its implementors and users may look deceptively easy to someone who hasn't done it before. But in reality it is one of the most difficult intellectual tasks in the field of programming.

The programming language Algol 60 introduced recursive procedures, block structure, scope rules, and type declarations in imperative programming. It was developed by an international committee that included John Backus (the developer of Fortran), Fritz Bauer and Klaus Samelson (who, together, developed the stack method of expression evaluation), John McCarthy (the inventor of LISP and one of the founding fathers of artificial intelligence), Alan Perlis (a pioneer of compiler development and the first chair of computer science at Carnegie-Mellon), and Peter Naur (whose contribution to Algol would be a landmark in computing).

Now, it is one thing to have a group of smart people sitting around a table discussing clever ideas. It is quite another thing for these people to describe their best ideas concisely in writing.

In 1959, at the initiative of Peter Naur, the *Algol Bulletin* was issued, which served as an international forum for discussing the development of the language. The bulletin was published by Regnecentralen.

For a meeting of the committee in January 1960, Naur prepared an unsolicited draft of the Algol report. Throughout the draft, he used a recent notation introduced by John Backus to define the syntax of *all possible* Algol programs! Naur's improvements of Backus's notation became known as *BNF notation* (or *Backus-Naur form*).

This was a huge step forward compared to the Fortran report, that defined the programming language by examples only. The problem with this informal method is illustrated by the old joke that "French is easy: 'horse' is *cheval*, 'dog' is *chien*,... and so on."

John Backus (1981) acknowledged Naur as the driving intellectual force behind the definition of Algol 60:

> Peter Naur's conduct of the Algol Bulletin and his incredible preparation for that [January 1960] meeting in which Algol was all written down already in his notebook—he changed it a little bit in accordance with the wishes of the committee, but it was that stuff that really made Algol 60 the language that it is, and it wouldn't have even come about, I don't think, had he not done that.

After seeing his draft, the committee asked Naur to be the editor of the official *Algol 60 report*.[1] Twelve years later, the Dutch computer scientist Edsger Dijkstra (1972) wrote:

> The famous Report on the Algorithmic Language Algol 60 is the fruit of a genuine effort to carry abstraction a vital step further and to define a programming language in an implementation-independent way...The report gloriously demonstrated the power of the formal method BNF, now fairly known as Backus-Naur-Form, and the power of carefully phrased English, at least when used by someone as brilliant as Peter Naur. I think that it is fair to say that only very few documents as short as this have had an equally profound influence on the computing community.

⋆ ⋆ ⋆

A computer program, written in a programming language, like Algol 60, is just another *text*. You can print it and edit it, but a computer cannot execute it as it stands. Before an Algol program can be executed, it must be "translated" into numeric machine code for a particular computer. The system program, that performs this translation, is called a *compiler*.

The small core memories of the mid 1960s made it impractical to write a program, as large as an Algol compiler, in a programming language, such as Fortran. Why? Because machine code generated by a Fortran compiler (or any other compiler) occupied significantly more memory than hand-written code. To use a small memory efficiently, a compiler had to be written in *assembly language*—a cryptic notation that required a programmer to specify each machine instruction in the code. A large program written in assembly language usually only made sense to the person who wrote it.

After the completion of the Algol 60 report, Regnecentralen's next challenge was to design Algol compilers for Dask and Gier. In this effort, Jørn Jensen's genius for machine coding would play a key role.

The American computer scientist, Alan Perlis (1981), left this impression of Jørn:

[1] P. Naur (ed.), J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauer, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger, Report on the algorithmic language Algol 60, *Communications of the ACM*, May 1960.

> When the [first Bendix G-20 computer] arrived at Carnegie it came with a full load of software: one binary load routine used by the engineers for testing memory. That was the sole extent of the software.
>
> How could one build a compiler [for Perlis's language, named Gate] quickly? We were fortunate at that time to have with us a visitor from Denmark, Jørn Jensen, who was with the Regnecentralen. Jørn was a magnificent programmer.
>
> Jensen sat at one desk; he was building the assembler. Arthur Evans sat at another desk; he was building the parser. Harold van Zoren sat at the third desk; he was building the code generator. All three were being defined simultaneously.
>
> The method of construction worked as follows. Jensen would decide that a certain construction ought to be in assembly language, and he would broadcast to the other two...When they decided...what changes would be required, working in good code-team fashion, they suspended and broadcast back to Jensen their proposals. Jensen would drop what he was doing and start another independent process. The amount of code that each wrote turned out to be of the order of about 2,000-3,000 machine language instructions. It turned out that at that size of code, such a technique worked magnificently. Each of the programmers could keep two to four processes in the air simultaneously, and changes progressed very fast. All three parts were completed at the same time. Jensen debugged the assembly language on the computer, simultaneously with the debugging of the parser and the code generator.

In June 1960, the Dutch computer scientists, Edsger Dijkstra and Jaap Zonneveld, completed the world's first Algol 60 compiler. At Regnecentralen, Jørn Jensen, Toke Jensen, Per Mondrup, and Peter Naur completed an Algol compiler for Dask in late 1961. This was followed by the much more elegant compiler for the Gier, which Dijkstra called "a masterpiece." To appreciate the achievements of the Dutch and Danish software pioneers, you need to know that there were no textbooks on compilers at the time. In 1964, Brian Randell and Lawford John Russell would publish the first book on *Algol 60 Implementation*.

Regnecentralen's second compiler implemented Algol 60 on the Gier computer with a memory of only 1K words and a drum of 12K words. Inspired

by the Atlas computer at Manchester University in England, the run-time system implemented "demand paging" of compiled code—without any hardware support! The demand paging used the drum to simulate a "virtual memory" that was much larger than core memory, and, in most cases, almost as fast.

The programs and tables of the Gier compiler occupied about 5,200 memory words. The machine programming was done almost entirely by Jørn Jensen. It is amazing that a human being could comprehend such a large program written in unreadable assembly language!

Gier Algol (Naur 1963a) introduced several innovations in compiler technology, which I cannot go into here. The compiler was checked by using it to compile small Algol programs constructed specifically to ensure that each instruction of the compiler would be executed at least once. This systematic approach to testing made the compiler virtually error-free.

Regnecentralen planned to demonstrate Gier Algol at the IFIP Congress in Munich, Germany, in August 1962. As luck would have it, the Gier, that was shipped to Munich by truck, was shaken to pieces. Bech immediately borrowed another Gier, that had already been delivered to a customer, and sent it to Munich (again by truck).

The demonstration of Gier Algol in Munich was a success, and Regnecentralen got a contract with Siemens to develop a Cobol compiler for the Siemens 3003 computer.

This is where things stood, when I joined Regnecentralen's compiler group on March 1, 1963.

⋆ ⋆ ⋆

At Control Data Corporation in Minnesota, Seymour Craig designed the most powerful computers of the 1960s. The CDC 1604 was a large computer intended for scientific customers. It had been used to provide on-line quotations of stock prices from the New York Stock Exchange.

In 1963, Regnecentralen acquired a CDC 1604A computer for its main service center. Many years later, I had dinner with a former CDC executive, who remembered Niels Ivar Bech. He had never forgotten this Dane who traveled to the corporate headquarters in Minneapolis and tried to convince a roomful of rugged executives to give Regnecentralen a CDC 1604A computer, in return for a Cobol compiler—with Danish keywords! When that didn't work, Bech negotiated an agreement with the Danish shipbuilding yard, Burmeister & Wain, to buy a quarter of the computer time.

Regnecentralen was a lively place. On the afternoon of March 28, 1963, seventy people gathered in the small cafeteria at the Rialto center to celebrate the news of the CDC computer. An hour later, after liberal consumption of the strong beer, known as Easter brew, the room was very noisy. At dinner time, I followed this happy group of people on a tour of local restaurants. Around midnight, I walked home with a splitting headache.

In August 1963, the CDC machine was installed in the Rialto center. Bent Bagger and Henning Bernhard Hansen would replace the English keywords in the CDC Cobol compiler with Danish words. The creative part of this task was to suggest Danish terminology for data processing concepts. Once they had settled on the terminology, the actual replacement of keywords in the CDC compiler must have been straightforward, compared to the task of building a complete Cobol compiler from scratch (as the compiler group had to do).

Peter Naur was also interested in the development of computer terminology. One day he entered the office, in which Paul Lindgreen and I were working, and started talking about finding Danish words for computing. Since computers are not just number crunchers, he felt that "computer science" was a misnomer. He had decided to call the discipline *datalogy* (in Danish: "datalogi"). The architect of Gier, Bjarner Svejgaard, would later remark, that English may not be good Danish, but apparently a mixture of Latin and Greek is all right.

Naur was now trying to decide what a "computer" ought to be called. In a facetious mood, I suggested calling it a *datamaton* (Danish: "datamat"). I didn't tell Naur that my inspiration was a self-service laundry in Frederiksberg, named "laundromat" (Danish: "vascomat"). On the spot, Paul Lindgreen added the word *datamatics* (Danish: "datamatik") to denote automatic data processing (Naur 1966b). (Paul Lindgreen's recollection is that he was the one who asked me to suggest a Danish word for a computer.)

This is the only time, I have added a new word to my own language. For many years, everybody in Denmark called it a "datamat," until a new generation of PC users decided that smart people speak "Denglish." So, nobody says "datamat" anymore. It is now "computeren"—pronounced with a heavy Danish accent.

⋆ ⋆ ⋆

Before I could contribute anything to a Cobol compiler, I obviously needed to teach myself to write small computer programs.

One Sunday, my father wanted to invite me to the horse races, but, since I didn't care which horses won, I preferred to stay at home. The following Sunday, he proposed that we try our luck at the casino. After a pleasant dinner, we drove to the Marienlyst resort, north of Copenhagen.

At the roulette, my father followed a simple strategy for postponing the inevitable loss of his money, by slowly increasing his bets, until he reached the maximum amount permitted by the resort. When that happened, he would begin another round of bets, starting with the smallest possible bet. If he won anything, he would immediately start another round. When his total losses exceeded the modest amount, he was prepared to lose for the evening, he quit.

To my great surprise, I won a small amount of money that evening. The same thing happened to me on another occasion. Of course, something *must* be wrong, I thought—every roulette is designed to have only one winner in the long run: the owner! This reminds me of the classic exchange in the movie *Never give a sucker an even break* (1941): "Is this a game of chance?" asks the patsy. "Not the way I play it!" responds the card shark. So I decided to write an Algol program that would make the Gier act like a roulette.

Since Gier had no operating system, I signed up for a block of time, say 15 minutes, and had the machine all to myself. Most of the time, the computer was idle, while I input my program text from paper tape, typed user commands, or printed my output on the typewriter terminal.

For larger computers, the inefficiency of open shop operation inspired computer manufacturers to develop batch processing or multiprogramming systems. However, for the small Gier computer, the manual operation was not a serious problem.

At first, my computer roulette was not very random: it stopped twice as often on odd numbers as it did on even ones. But, after a while, it worked:

> The other day I invited my father to Regnecentralen, because I had succeeded in making the computer Gier play roulette. My father was very amused indeed. The computer is connected to a typewriter, and the roulette-program was made so, that the computer started the performance by asking: "How many games are you prepared to risk?" I typed: "10000 games." Then the computer started playing the many games, saying BZZ, BZZ... for nearly three minutes. And, finally it typed: "Sorry!—you have lost 45980 Dinars." [For the benefit of Milena, I replaced

> Danish kroner with Yugoslav dinars.] (Letter to Milena, June 6, 1963.)

Using elementary probability theory, it was easy to verify the results of my simulation. Once I understood the exact nature of the game, I lost all interest in playing roulette.

In my book, *Programming for Everyone in Java* (1999), I used roulette simulation as a beginner's exercise.

Milena felt that my simulation was a frivolous way of using an expensive computer. I explained that it was an example of the Monte Carlo Method of computing, named after the famous resort town in Monaco, which has the world's oldest casino.

Thirty years later, I used simulation on a supercomputer with 48 processors to find the shortest tour through 100 cities in two minutes (Brinch Hansen 1995). This is obviously a problem of some practical importance. (It is also a much harder problem than roulette simulation.)

⋆ ⋆ ⋆

The programming language Cobol was designed about the same time as Algol. At the 1978 conference on the History of Programming Languages (HOPL), Joe Wegstein, National Bureau of Standards, commented (Cobol Discussion 1981):

> The Cobol Committee had these people from various manufacturers who had a lot of vested interest, and were very intense about that sort of thing in connection with everything being done. Whereas, the Algol Committee had a bunch of senior professors of Europe and an oddball collection from the U.S., and—all very temperamental and intense about mathematical aspects of programming.

Now, Algol was designed for numeric computations. The only data structures supported by the language were tables ("arrays") of numbers.

Cobol, on the other hand, was designed for business data processing. The most important contribution of Cobol was the introduction of data records and sequential files, which were needed to process data on punched cards and magnetic tapes.

Ten years later, Niklaus Wirth combined both forms of computing by including records and files in his Algol-like language, Pascal.

In his History of Modern Computing, Paul Ceruzzi (2003) writes:

> Cobol became one of the first languages to be standardized to a point where the same program could run on different computers from different vendors and produce the same results.

Alas, this worthy goal was *not* reached. After completing the Siemens Cobol compiler, Regnecentralen concluded that:

> The major problem of implementation turned out to be the numerous definition problems created by the vagueness of the official Cobol report. (Brinch Hansen 1966)

Compared to Algol 60, Cobol was poorly defined. In places, where the Cobol report was incomprehensible, Regnecentralen's compiler group had to *guess* what the intention of the Cobol committee *might* have been. More than likely, other compiler groups interpreted the report differently and implemented incompatible variants of the language.

A peculiar feature of Cobol was its attempt to replace well-known algebraic notation by verbose English: whereas you might write a/b in Algol, this became DIVIDE A BY B in Cobol (or even DIVIDE B INTO A). This was supposed to make it easier for managers to read programs.

At the HOPL conference, the Cobol notation provoked the following exchanges of questions and answers:

> *Question*: Did the participants in the original Cobol development sincerely believe that the use of an English-like language would enable nonprogrammers (e.g. managers) to understand programs.
>
> *Answer*: Yes. We sincerely believed managers would be able to read the programs and that more people would find them easier to write.
>
> *Question*: Did the Cobol committee seriously believe that the users could not handle grade school operators +,−, ×, /?
>
> *Answer*: Quite seriously, there was a strong sentiment. . .that the users did not want to use algebraic symbols in the normal course of writing an arithmetic expression.

How can one explain that Cobol remained the most widely used programming language on the planet for decades? Well, in 1960, the U.S. Department of Defense announced that it would only use computers that supported Cobol. That guaranteed the commercial success of Cobol—independent of its merits!

Needless to say, the government could not dictate the opinions of computer scientists:

> *Question*: [Cobol] continues to be viewed with great disdain, as is data processing in general, by many computer scientists. It is rarely taught in "prestigious" computer science departments, where it is still regarded as an abomination. Have you any comments?
>
> *Answer*: I think Cobol ought to be taught because there are concepts in there which are important and which are useful, and business data processing has a large significant, intellectual component. But most of the senior key computer science people don't agree.

This was the programming language that Regnecentralen's compiler group would be responsible for implementing for Siemens in Munich.

⋆ ⋆ ⋆

Peter Naur and Jørn Jensen worked so closely together that they hardly needed to say anything to solve a problem. I remember a discussion where Peter was writing something on the blackboard, when Jørn suddenly said "but Peter ..." and immediately was interrupted with the reply "yes, of course, Jørn." I swear that nothing else was said. It made quite an impression on me, especially since I didn't even know what the discussion was about in the first place.

As an electronic engineer, I was used to circuit diagrams showing resistors, capacitors, and transistors. What the compiler guys did was completely different. On the blackboard, they would illustrate their ideas with small Algol 60 fragments. Since Algol was not a natural language for thinking about data structures, they would also draw complicated pictures of tables linked together in mysterious ways by arrows.

However, in truth, the Cobol compiler was progressing very slowly (if at all). Naur and Jensen had already finished their second compiler for the

elegant Algol language. Now they had to do it again with a far less attractive language. It seemed to me that their hearts were not in it.

After three months, I began to catch on. On May 23, 1963, I wrote to Milena:

> We have common meetings sometimes on Fridays, just to coordinate things and settle issues of doubt. You see, the normal situation is that everyone gets a small part of the project to work on. First, everyone will work enthusiastically for a few weeks or so, independent of the others, but gradually the tempo slows down for a lot of psychological reasons—some details cannot be solved, before you know what the others are doing, and other problems you simply close your eyes to (and put them in a drawer).
>
> So every now and then, Peter Naur calls for a meeting to make us face the problems. You can't imagine, how I enjoy the atmosphere of a group of people, who have to convince each other, defend their views, and reach decisions.
>
> Sometimes I get permission to work at home for several days—mainly, when I have to write a report on what I have been doing lately. (There are too many distracting factors at work: noise from the street below, and the temptation to chat with the others.) [Throughout my professional career, I would continue to do all my writing at home.]
>
> Well, the latest crazy and wonderful idea is, that the whole department of some ten engineers is going to work "at home" for one week to jump-start the project. From Monday, the 21st of October, until Saturday, the 26th, we are moving to a small inn [Mogenstrup Kro] in the southern part of Zealand, far away from any big, noisy town. [Zealand, on which Copenhagen is also situated, is the largest island in Denmark.] Each of us will have his own small room to work in, and often we will gather for a common discussion. In the evenings, we can walk in the woods and get to know each other outside the office. I find it a splendid idea.

The Mogenstrup meeting clarified many things: The Cobol compiler would be divided into ten phases (known as "passes"). Since the Siemens computer had no drum or disk, the compiler would use three magnetic tapes. The compiler would be input, one pass at a time, from a system tape. The other two tapes would be used as scratch tapes during compilation.

Pass 1 would input a Cobol program from punched cards, perform a partial compilation and output intermediate code on one of the scratch tapes. Pass 2 would then input the intermediate code from tape, perform another partial compilation, and output slightly more detailed code on the other scratch tape, and so on. In this way, the compiler passes would be loaded, one at a time, from the system tape, while the compiled code would move back and forth between the scratch tapes, being gradually refined. The last pass would leave final code on a scratch tape, from which it could be loaded and executed.

Since every pass performed a single scan of the original Cobol program (or the intermediate code), this scheme was known as multipass compilation. Multipass compilation made it possible to use a compiler that was much larger than the available core memory. The compiler group had already used multipass compilation of Algol programs on Dask and Gier.

Peter Naur and Jørn Jensen would be responsible for the overall design of the Cobol compiler. However, in reality, Peter Villemoes became the project leader. The design, programming, and testing of the individual passes would be done by Sven Eriksen, Roger House, Jørn Jensen, Peter Kraft, Paul Lindgreen, Ole Riis, Peter Villemoes, and me. Naur's cousin, Berta Kiær, would be our secretary.

Back in the Rialto center, my first task was to program the parser, a compiler phase that would check if the syntax of a Cobol program (that is, the sequence of programming symbols) was correct.

Instead of having a few basic constituents, that could be used in many contexts (as in Algol 60), Cobol 61 consisted of a large number of unrelated clauses, each of which required a special piece of code in each pass. This complexity made it impractical to perform syntax analysis the same way it was done in Gier Algol (by simulating a so-called "finite state machine").

I invented a different method of representing the Cobol syntax by linked lists of symbols. The parser would input a Cobol program, one symbol at a time, and use the linked lists to check the syntax. The parser would also erase all clauses with illegal syntax. This was my first (modest) invention in system programming.

When it was finished, the machine code and fixed tables of the syntax analyser occupied about 5,000 lines in assembly language—a fairly hefty program for a beginner. Other members of the group programmed compiler passes that were more complicated than the parser. However, after forty years, I no longer remember exactly who did what.

As I mentioned earlier, assembly language is extremely difficult to understand. Even after a short vacation, you may find it difficult to remember the meaning of your own assembly language program. I solved the problem of program documentation by adopting a brilliant method used to document the Gier Algol compiler: I divided each program page into two halves. The left half defined the program in assembly language, while the right half defined the same program in Algol 60. The assembler treated the Algol 60 statements as comments to be ignored. However, these comments simplified my job tremendously, since it was fairly easy to determine if a sequence of assembly language instructions implemented an Algol 60 statement correctly.

Now, if a program and its description are two separate documents, a programmer may not always remember to update the description, every time the program is modified. However, since the documentation method combined an assembly language program and its definition in Algol 60 in one document, it became natural for me to update both parts simultaneously.

My yearly salary of 22,900 kroner (about $3,300) was not a lot of money in 1963. So at the end of my first year at Regnecentralen, I asked Jørn for a raise. I told him, that I liked my job and would hate to give it up at this point. On the other hand, I felt obliged to take the consequences of my request—otherwise, how could I expect him to take it seriously? So I asked him to reach a decision within a fortnight. If I got no raise, I would find myself another job. Jørn smiled and said: "This is a viewpoint I can only respect. I will talk to Bech and tell you, whether you will get a raise or lose your job."

On April 5, 1964, I wrote to Milena: "Don't be nervous: I got my raise the following day."

A month later, serious testing of the Cobol compiler began. The compiler passes were tested, one at a time, in their natural sequence (pass 1 was debugged first, then pass 2, and so on). The compiler was tested by letting it compile small test programs written in Cobol—a method borrowed from Gier Algol.

The parser was the second compiler pass. When I began my tests, pass 1 had already been tested and was therefore able to compile my test programs into correct input for pass 2. In each test run, the compiler printed the test program, that was being compiled, followed by the output produced by pass 2. By comparing the test program and the corresponding output from the parser, it was easy to see which symbols it handled incorrectly. I would then correct the parser and repeat the same test case, until it worked.

You must remember, that the compiler was being programmed in one country and tested in another. In Munich, Siemens was still testing various aspects of the hardware. The machine was in so much demand, that we also had to use it in the evenings and during weekends, when the Germans went home. Let me tell you, walking towards Siemens on Hofmannstrasse 51, at 4 in the afternoon, while 10,000 workers walked the other way, was an experience!

With our limited access to the computer, there was no opportunity to experiment with incomplete programs. We took turns arriving in Munich with a complete compiler pass and a set of test programs, that had already been punched on cards and proofread in Copenhagen. The compiler passes were so carefully planned that few (if any of them) had logical flaws. The main purpose of our systematic testing was to remove the inevitable clerical errors.

I continued to use this method of program development for forty years. In my experience the combination of careful design, proof reading, and systematic testing can make programs more reliable than the hardware they run on.

Of course, this glib description does not reveal my early frustration with the parser, when nothing worked, and nothing was printed! The only thing I could do in that situation, was to read the beginning of my program, instruction by instruction, until I figured out why it produced no output.

From then on, my testing went as planned:

> May 26, 1964
>
> Dear Father—My program works! Believe me, it is an experience, finally to work on a large computer. I have been to Yugoslavia twice on my weekends.
>
> Soberly yours, Per

Although I now 'knew' how a computer worked, I still found it unbelievable that a machine would follow thousands of instructions I had written in pencil. It *is* pure magic that human beings have learned to construct computational processes by combining electricity, transistors, circuits, computer architecture, assemblers, compilers, operating systems, and user programs. If you don't share that feeling of awe, you haven't really understood the miracle of computing.

In Munich, we stayed at Hotel Daniel, Karlsplatz 15, close to the main railroad station. Here is a letter to Milena, mailed from the hotel on June 8, 1964:

> You are quite right, we had troubles with the program. In such a large program, there are always bound to be some errors. In fact, we are only here to detect and correct such 'bugs' and it has been quite a tricky task. But it works now, and I think I can fly back to Copenhagen by the middle of this week. However, first I must have some talks with the Germans about my next program.
>
> We are running to and from the computer all day long, from 9 in the morning till 7 in the evening. In the begining, the Germans were a bit puzzled by our unsocial behavior: we never spent much time chatting with them and would often criticize the way they had designed their computer.
>
> Last week, however, we had occasion to repair this impression. We were invited to join them in their monthly 'lab evening' ('Labor-Abend'). This is an evening where they go with their spouses to a restaurant and talk about anything but their work. When we arrived at the outdoor restaurant in Schwabing, the Germans had already been drinking for three hours and were in high spirits. One shy fellow was making speeches for all the girls at Siemens. That evening, artists exhibited their paintings by candlelight, while a group of teenagers played guitar and sang the blues. An endless stream of people crowded the pavement and the outdoor restaurants.
>
> We had several bottles of a not-so-famous white wine, labeled 'No 1a,' and engaged the Germans in the conversation they had missed. The evening ended in some strange restaurant at 3 a.m. The next day, I felt like a dying man in the computer room.

⋆ ⋆ ⋆

We were young and cocky and not always as polite to our German hosts, as we should have been.

The Siemens 3003 had a hardware feature that was meant to prevent its operating system from being destroyed by incorrect (or malicious) user programs. The operating system resided in a fixed part of memory. When

you flipped a switch on the computer, it became impossible to change any memory location within the protected area. This certainly guaranteed that the operating system code could not be changed during program execution.

However, the computer architect had overlooked one thing: an operating system must be able to record various data about running programs to function correctly. Since it was impossible to update memory locations in the protected area, the operating system had to keep its variables in unprotected memory locations, where they were completely at the mercy of user programs. If programs made arbitrary changes to these locations, it would soon crash the operating system.

In short, the so-called "memory protection" was a hacker's dream. The members of the compiler group knew this. When the Germans demonstrated the machine for us, it was a favorite joke of our American programmer, Roger House, to say: "Excuse me, you forgot to turn the protection switch on!"

⋆ ⋆ ⋆

Once in a while, the computer broke down, leaving us with a perfect excuse to relax:

> Last friday, our computer broke down completely, so we have had a quiet weekend without any work whatsoever. It is very hot in Munich, so yesterday we took a trip to the countryside to a small mountain lake, where we ate a tremendous dinner. Afterwards, we rented a rowing boat and drifted around the lake. Our 'real' work has been delayed quite a bit by repeated computer failures, so I will probably have to stay here at least another week. Not that I mind, since time passes easily in a city like Munich: we go to the theater and concerts, and eat mostly in 'foreign' restaurants—Chinese, Hungarian, Bosnian, Russian, Spanish, and Italian. (Letter to my father, June 11, 1964)

Breakfast was included in our hotel bill, and Siemens gave us a free lunch. Since Regnecentralen allowed us to spend a fixed daily amount for meals, we had plenty left for sumptuous dinners around town. We became connoisseurs of Munich's restaurant scene, and knew, for example, which of the two Russian restaurants was the best one. I soon learned that putting on weight is much easier than losing it again (which I never did).

⋆ ⋆ ⋆

On August, 21, 1964, I presented my first scientific paper at the NordSAM conference in Stockholm, Sweden. It described a method that made the evaluation of logical expressions slightly faster during the execution of a compiled Cobol program.

A logical expression of the form [IF] A GREATER B AND LESS C ..., is evaluated in three steps: (1) First, check if it is true (or false) that A is greater than B; (2) Then, check if it is it is true (or false) that A is less than C; (3) Finally, check if both conditions turned out to be true (or not). Since the GREATER and LESS relations are evaluated before the AND relation, GREATER and LESS are said to be operators of *higher priority* than AND.

However, if it turns out in step 1 that A is not greater than B, then step 2 is superfluous, and can be skipped. That was the whole idea behind my code optimization.

To me, this was an elegant compilation technique. But, looking back, I don't think it served any practical purpose. Before you go to the trouble of optimizing compiled code, you should conduct an experiment to find out, if it has any measurable effect. Otherwise, you are just increasing the size of your compiler for no good reason.

I cannot imagine that the efficiency of business data processing will ever depend on the speed at which a computer evaluates logical expressions. However, this minor optimization (which I do not claim to have invented) would later be included as a language feature in C and Java.

Today, it is still common for programmers to confidently recommend a method, because "it is faster" than another one—without offering any performance measurements to document the magnitude of the improvement.

Anyhow, here I was at my first computer conference lecturing in English to an international audience that included Peter Naur and Edsger Dijkstra. I was very encouraged to see both of them smiling broadly during my presentation. Afterwards, I discovered why: In my talk, I constantly said "the priority of this operator is higher than the priority of that one." Since English was not my native language, I mispronounced the word "higher" as "hi-ger" (with a hard "g" as in "good"). Everytime I did that, Naur and Dijkstra smiled.

⋆ ⋆ ⋆

My most difficult programming task was to write the input/output procedures for files stored on punched cards, magnetic tapes, and line printer

forms. This file system would be used during the execution of compiled Cobol programs.

Each tape station was about the size of a small closet. While a tape was being read or written, it moved from one reel across a magnetic head to another reel. To prevent the fast moving tape from breaking during frequent starts and stops, it also moved through two vacuum chambers, which held enough loose tape to absorb the forces of acceleration and deceleration.

The tape stations on the Siemens computer were rather unreliable. Jørn Jensen witnessed a faulty tape station jam a tape by rewinding both reels at the same time! Dust particles on the tape caused transient input/output errors. I handled these by transferring the same block of data again. This could, of course, have been done simply by backspacing over the last block and reading (or writing) it again. Instead I backspaced ten blocks—enough to move the bad block into the nearest vacuum chamber, where the air current would blow the dust off the tape. I would then upspace nine blocks and transfer the same block again.

A more serious problem was the permanent errors caused by spots of missing oxide on the tape. The only thing you could do with a bad spot was to ignore it and write the same block again after the spot. To facilitate error recovery, each block was output with a block number and a block length. A block that did not have the correct number and expected length during input was assumed to be a bad spot and was skipped.

When Jørn Jensen first told Siemens about the need to extend data blocks on user tapes with two additional words, they would not agree to this. And who could blame them. It would be a major problem for Siemens to ask its customers to adopt a new tape format, that would make their existing tapes unreadable.

I then asked Jørn to tell the Germans that Regnecentralen could not be responsible for the reliability of tape input/output, unless they agreed to our proposal. That did it! They agreed, and the tapes worked fine.

Peter Villemoes had developed the techniques for dealing with tape errors during compilation. However, the more general file system I was writing for the execution of Cobol programs, posed additional challenges.

Compiled Cobol programs were supposed to run in a core memory of 8K words only. When a program opened a file, it was assigned a buffer space in memory. To make the best use of the small memory, the buffer space was reclaimed, when the file was closed. Over time, the memory ended up being full of active buffers separated by gaps of unused space left behind by closed

files. If a buffer space could not be found for a new file, the gaps between the buffers were closed by moving all the buffers to one end of the memory.

The trickiest feature was probably the ability to restart a Cobol program from a previous point of execution, after a hardware failure. At regular intervals, my input/output procedures would stop the running program briefly and output restart data on a tape. When the hardware had been fixed, the most recent restart data were used to instruct the operator to mount the same tapes, one at a time. The tapes would then automatically move to the same spots, where they had been, when the restart data were written, and the computation would resume, as if nothing had happened. This was fun—and awesome—to watch!

I regard the run-time filing system as my graduation project from the compiler group. I now understand that it was really a small operating system, I had programmed. However, in the mid 1960s, the dividing line between language implementation and operating systems was still not clearly understood.

⋆ ⋆ ⋆

After a total effort of 15 man-years, Regnecentralen delivered a complete Cobol implementation of 39,000 instructions to Siemens in July 1965. We had used about 600 hours of computer time to assemble and test the system. In human time, it took about 45 minutes to program each instruction and less than 1 minute to test it.

The Siemens Cobol compiler was eight times faster than the fastest American compiler evaluated by the Bureau of Ships (Siegel 1962). After a basic input/output time of 45 sec, our compiler translated a Cobol program at the rate of 250 cards per minute, generating final machine code.

When the compiler was completed, Sven Eriksen joined Siemens in Munich and became responsible for maintaining the compiler. He was incredibly well organized. We mailed every compiler change to him as a deck of punched cards with a separate test program to verify that the correction worked. He kept the punched cards of our original Cobol implementation, and all subsequent modifications, in chronological order in a card filing cabinet.

About a year after the delivery of the Cobol system, a user reported that my filing system did not work correctly. Since Siemens had modified the Cobol system in places, it could have been a nightmare for me to travel to Munich and determine what the error was and who was responsible for correcting it.

Instead I asked Eriksen to reestablish the compiler exactly as it was a year ago. He was able to do that, and demonstrate that the customer's program worked under the original file system, we had delivered. That got me off the hook and left Siemens with the problem of figuring out, what they had done wrong after the delivery.

I wonder, how many software developers today treat system updates in the same professional manner?

⋆ ⋆ ⋆

Peter Naur encouraged Roger House, whose native language was English, to write a paper about the Cobol compiler. Since this idea got nowhere, I wrote the paper with helpful comments from Roger. It was published in the Scandinavian journal BIT in 1966.

Today, few people (if any) have access to the Cobol compiler for the Siemens 3003 computer. But anyone, who is interested, can still read about it in BIT. In the long run, it seems to me, the most important aspect of programming is the description of interesting ideas in readable papers. The programs themselves are merely useful by-products of this effort. Besides intellect, the most valuable asset of a programmer is the ability to write clearly! Needless to say, this viewpoint is not popular among students, who prefer free-style "coding" without the burden of documentation.

⋆ ⋆ ⋆

Before joining Regnecentralen in 1963, I met Milena in Slovenia. During my trips to Munich over the next two years, I visited her ten times. This was the most romantic episode in my life. On March 27, 1965, we were married in the townhall of Ljubljana.

The compiler group sent us a telegram with amusing comments on the wisdom of marrying (Fig. 3.1).

27 3 65

MILENA AND PER BRINCH HANSEN
HOTEL BELVEDERE
IZOLA ISTRIA

LUCKY TEST BERTA STOP
NO PROTEST RIIS STOP
POOR YOU BUT YET GERDA KRAFT STOP
LUCKY YOU LINDGREEN STOP
AND ALL THAT JAZZ DIANE STOP
MONDRUP TOKE JOHANSEN STOP
BELIEVE US IT IS NOT TOO BAD ROGER AND JEANNE STOP
A CHALLENGE BUT WORTH IT NAUR STOP
A HUGE GRATULATION FROM THE ABSENT GUYS

Figure 3.1 Wedding telegram from the compiler group.