Following by chance in the Masters' foot prints
T.L. van Raalte

One's attitudes to anything new is a function of its inherent qualities and one's own background. So I will start from a bit further back than April 1957 to explain my use of FORTRAN.

My first experience of computing was trying to program optical designs on a Ferranti Mk I* in 1951 or 1952 at the Ministry of Supply, Fort Halstead, and later at AWRE (now AWE) Aldermaston. (Dates may be unreliable as I was not allowed to keep my working papers on my retirement 20 years ago.)

The Mk I* had 768 words of memory but this was not as generous as it sounds. Each numerical datum occupied 2 words. The memory included a routine for division, entry to which used 2 or 3 words, and a creaking mechanism which provided only one level of subroutine and used 4 words to do it. In addition the range of numbers was from -2 to just less than +2, so scaling was as much a problem as the problem itself. Programming was done using a 5 bit code: if I remember correctly 11011 was multiply, and so on.

I soon realised that programming wasn't as easy as expected and I needed help, so I invited a colleague, June Stanley, to become a programmer. I demonstrated a counted loop and one that iterated to a solution, gave her a copy of the 32 instructions, and she joined me as a fully qualified programmer on my second attempt to design lenses. At that time we were the only members of AWRE outside the Theoretical Physics Division allowed to use their facilities.

At some stage the Mk I* was closed down and we moved to the IBM 704 which offered the unbelievable luxury of floating-point arithmetic in the unfillable space of 4K. My delight in such opulence was tempered by the realisation that in time the 704 would also be replaced and this might involve reprogramming again.

So when I was introduced to the first version of FORTRAN in 1957 (as a "novelty" by the AWRE programming adviser, and as a "joke" by the IBM manager who shook his head and added 'It's amazing what they can make a computer do') I was quite excited. So long as something like FORTRAN was provided on subsequent computers I could see a far simpler future. This was not a universal reaction to FORTRAN. Some programmers had a rather macho attitude to programming and FORTRAN was definitely for wimps. One complaint was that FORTRAN didn't allow one to write programs in a sneaky way, and another that it didn't call for the exercise of one's professional skills; anybody could write programs in FORTRAN.

True, the first FORTRAN lacked a formal subroutine mechanism but to anyone who had survived the do-it-yourself way the Mk I*, the COMPUTED GO TO was a splendid way of writing subroutines (in the plural) and nested as deeply as needed.

June and I produced a program that contributed to the design of some lenses but politics dictated that the projected automatic optimisation would never be completed. British nuclear atmospheric tests ceased and part of the Trials Division became watch-dogs seeking methods of detecting other nations' tests against the various backgrounds of natural events. At the same time the restrictions on "outside" users of the computer were lifted and most of the scientists and engineers at AWRE became FORTRAN programmers.

Having programmed at the bit level on the Mk I* and by the Symbolic Assembly Program on the 704, June and I acquired a good understanding of the code that the FORTRAN compiler produced. And because we had been using computers for some years by then novice programmers in difficulty tended to turn to us for help, and we acquired very flattering reputations for finding errors in programs. In fact, of course, it was very rare that we did anything of the sort, but knowing how the compiler would treat code enabled us to ask the questions that led the authors to find the faults themselves. Fortunately they never seemed to realise it. I say fortunately not just because it's nice to be regarded (even if wrongly) as particularly gifted, but because it exposed us to a wide range of problems and it was this experience that defined the future direction of my work.

I'm sure that other people at the Anniversary Meeting will deal with popular problems in detail so I will just briefly mention four that I remember. Two were genuine bugs. One was a subroutine defined with an integer argument that was increased by 1 during its execution and was called with the constant 1, so that thereafter 1 had become 2. The other was the statement DO 20 I=1,10 typed as DO 20 I=1.10 a fault that achieved some notoriety years later in an Apollo mission. Two others were correct programs which seemed to simply fade away in the 2 minute slots in which they ran. One was a nest of DO loops so deeply nested that the core code would have been repeated millions of times. The other was a program with so many exponentials that page after page looked like star maps with assignments containing expressions like (A**2.0+B**2.0)**0.5. It ran in seconds when simplified.

Just before I come to the commonest errors I will add a personal note. In the Theoretical Physics Division the task of fault finding was done by Ian Smith. June and Ian now married, are at the Anniversary Meeting.

I don't know whether Ian's experience with professional mathematicians was the same as June's and mine with electrical and mechanical engineers. For me the great surprise was that on the whole they could cope with calculations (not always written in ways most sympathetic to the compiler as shown above) but they stumbled in the most imaginative ways over DIMENSIONs, FORMATs, and EQUIVALENCEs. The result was either the repeated rewriting and recompiling of input routines or the error prone process of editing the data.

My background becomes relevant again. I was a mathematician but I nearly became a social anthropologist and out of work hours I was closely connected with the anthropological fraternity. Anthropology includes linguistics and linguistics had introduced me to a thrilling and seminal work – number 4 in the Janua Linguarum series published around 1956 by Mouton and Co.: Syntactic structures by Noam Chomsky. This slim booklet explains how humans understand the most complicated language and possibly malformed sentences and the contrast with computing struck me dramatically. A program can be as complicated and clever as you like but put a decimal point where it doesn't expect one and it crashes.

The implication is simple. Data defined syntactically could be read with the format (say of) 120A1 and then analysed in the computer. Each datum identifies itself. Omitting some detail, a string of digits is an INTEGER; one decimal point makes it REAL; a point and a capital E or D is floating point; .T. or .TRUE. etc. are BOOLEAN, and anything else is alpha-numeric. Nothing could go wrong: conforming data would be identified and prepared for use and ungrammatical data could be treated in such a way that the program would not crash.

The first version did call for a format as part of the data. June and I had learned better how to judge what we would achieve in a limited time, and as I couldn't get approval to develop my ideas officially we had to do so in such spare time as we could create, so we didn't try to do everything at once. A later version treated embedded formats as the data equivalent of such verbal punctuation as "y'know" in popular speech. An input giving the array lengths became irrelevant but provided a useful check.

Of course, what I was doing was not as novel as I thought at the time. I have never (to my shame) read any of Dr. Backus' papers but I realised later that syntactic structures must have provided the foundation for FORTRAN itself and what I was doing was merely applying to data what he had already done to analysing code.

In my definition a file consisted of a heading which identified the data and an END statement. Between these could be any number of what I called sheaves of data. A sheaf of data could contain any number of strings or multiplexed strings of numbers in any form and with any spacing and any number of so-called parameters were relevant to the data. These parameters could be house-keeping things like the degree of multiplexing, or the length of the strings, or experimental measurements such as instrumental settings and sensitivities, the interval at which the wave form is digitised, and so on. All of these strings and parameters are named and as the file is read directories of names are built so that every parameter and every item in an array is addressable.

So the data is safely read. Now, what to do with it? A typical process is to Fourier analyse a wave form, modify it by a transmission function and finish with an inverse transform. But in some cases the data might need a preliminary smoothing or the application of a tailor-made instrumental correction peculiar to the particular recording instrument. A framework can be constructed which is simply a series of calls to subroutines which perform the various steps. In the case of instrumental corrections etc., appropriate subroutines can be loaded. An interpreter then reads instructions and data names and sets up pointers to collect the appropriate inputs and dispatch the outputs to addresses whose names are added to the directories, for which space is allocated. As mentioned before the program can include procedures to handle ungrammatical data in a way appropriate to the location of the error.

A program of this sort is like a constructional toy such as MECCANO in which a number of pre-formed pieces are selected from the toy box and a new model is created. I cannot remember the devious way in which I made the name MECCANO describe the program but it is now irrelevant as the manufacturers (of MECCANO) refused my request to use the name.

So I substituted the more succinct (and rather silly) name ICE which stood for Input Controlled Execution – silly because very few programs are not controlled by their input. It was around this time that June moved to other fields and John Young joined me. He embraced the ideas of ICE enthusiastically which was particularly refreshing as no one other than June had had the slightest understanding of what I was doing. In particular my various masters thought that 'jam today' (in the form of results produced at whatever costs in manpower and wasted machine time) was worth more than the promise of easily and efficiently produced 'jam tomorrow'. So John and I advanced ICE as much as our other commitments allowed. In fact, those commitments produced some amazingly long-lasting programs – a credit to the sound design of FORTRAN but also to the quality of John's work, and for an account of them see his contributions.

As for me, I moved elsewhere later and finally produced a fully working ICE2 with dynamic storage allocation (not then available in FORTRAN) in Pascal. I am told that on my retirement ICE2 was wiped from the computer.