# Computing Science: Achievements and Challenges

When, at the close of the 20th Century, I am supposed to talk about Computing Science, I am immediately faced with the question "Which Computing Science?". In my personal case I can narrow it down to "European CS or American CS?", but with 32 years of European and 26 years of American employment as a computing scientist, I cannot eliminate the dilemma. (As an aside, don't add the two numbers I gave you, for there was some overlap.) To summarize my position with respect to that transatlantic difference: I cannot ignore it, but am also allowed to address the issue openly (that is, if not qualified, at least entitled).

The major differences between European and American CS are that American CS is more machine-oriented, less mathematical, more closely linked to application areas, more quantitative and more willing to absorb industrial products in its curriculum. For most of these differences there are perfect historical explanations, many of which reflect the general cultural differences between the two continents, but for CS we have also to take into account the special circumstance that

due to the post-war situation, American CS emerged a decade earlier, for instance at a time when design, production, maintenance and reliability of the hardware were still causes for major concern. The names of the early professional societies are in this respect revealing: the "Association for Computing Machinery" and the "British Computer Society". And so are the names of the scientific discipline and the academic departments: in the US, CS is short for Computer Science, in Europe it is short for Computing Science.

The other circumstance responsible for a transatlantic difference in how CS evolved I consider a true accident of history, viz. that for some reason IBM was very slow in getting interested in Europe as a potential market for its computers, and by the time it targeted Europe, this was no longer virgin territory. Consequently, IBM became in Europe never as predominant as it has been in Northern America. Now, several decades later, it is hard to visualize and to believe how strong and how pervasive IBM's influence has been, but let me relate to you an incident that took place in 1975, at a conference that IBM had organized at Montebello for CS professors from all over Canada. (At that moment

even IBM knew that OS/360 was a disgraceful mess and realized that it would become increasingly difficult to sell it to knowledgeable customers. And so, with all the sticks and carrots the company could muster, the Canadian Universities were urged to exclude the topic "Operating Systems" from their curricula, the reason given being that "working on the operating system was not an entry-level job". Having seen IBM in action, I have mixed feelings about today's fashion of strengthening the links, financial and otherwise, between higher education and industry.

It is a pity that they were called Programming Languages, but apart from that unfortunate name, FORTRAN and LISP have been the great contribution of the 50s.

That name was unfortunate because very soon the analogy with natural languages became more misleading than illuminating. It strengthened the school of thought that tried to view programming primarily as a communication problem, it invited the psychologists in, who had nothing to contribute, and it seriously delayed the recognition of the benefits of viewing programs as formulae to be derived. It was the time of Edmund C. Berkeley's

"Giant Brains or Machines that Think", it was the time of rampant anthropomorphism that would lead to the false hope of solving the programming problem by the verbosity of COBOL and would seduce Grace M. Hopper to write a paper titled "The Education of a Computer". Regrettably and amazingly, the habit lingers on: it is still quite easy to infuriate computing scientists by pointing out that anthropomorphizing inanimate objects is in science a symptom of professional immaturity.

As said, FORTAN and LISP were the two great achievements of the 50s. Compared with FORTRAN, LISP embodied a much greater leap of imagination. Conceptually FORTRAN remained on familiar grounds in the sense that its purpose was to aid the mechanization of computational processes we used to do with pen and paper (and mechanical desk calculators if you could afford them). This was in strong contrast to LISP whose purpose was to enable the execution of processes that no one would dream of performing with pen and paper.

At the time LISP's radical novelties were for instance recognized by its characterization as "the most intelligent way of misusing a computer", in retrospect we see its radical

novelty because it was what is now known as a language for "functional programming", while now, 40 years later, functional programming is still considered in many CS departments as something much too fancy, too sophisticated to be taught to undergraduates. LISP had its serious shortcomings: what became known as "shallow binding" (and created a hacker's paradise) was an ordinary design mistake; also its promotion of the idea that a programming language should be able to formulate its own interpreter (which then could be used as the language's definition) has caused a lot of confusion because the incestuous idea of self-definition was fundamentally flawed. But in spite of these shortcomings, LISP was a fantastic and in the long run highly influential achievement that has enabled some of the most sophisticated computer applications.

I must confess that I was very slow in appreciating LISP's merits. My first introduction was via a paper that defined the semantics of LISP in terms of LISP, I did not see how that could make sense, I rejected the paper and LISP with it. My second effort was a study of the LISP 1.5 Manual from MIT which I could not read because it was an

incomplete language definition supplemented by an equally incomplete description of a possible implementation; that manual was so poorly written that I could not take its authors seriously, and rejected the manual and LISP with it. (I am afraid that tolerance of inadequate texts was not my leading characteristic.)

It is much harder to get lyrical about FORTRAN, which as a programming language was denied the epithet "higher-level" already in 1962 (in Rome), but I should be grateful for I learned a few things from it. During an oral examination I had a student develop a program which we would now recognize as implementing pointer manipulations using a one-dimensional array. The candidate did very well until at the very end he got mysteriously stuck, neither of us understanding why. It turned out that he should have written down "a[a[i]] := ...." but that a mental block prevented him from conceiving that because FORTRAN (to which he had been exposed extensively) did not allow index expressions as complicated as "a[i]". It was a revealing warning of the devious influence the tools we use may have on our thinking habits.

I learned a second lesson in the 60s, when I taught a course on programming to sophomores, and discovered to my surprise that 10 % of my audience had the greatest difficulty in coping with the concept of recursive procedures. I was surprised because I knew that the concept of recursion was not difficult. Walking with my five-year old son through Eindhoven, he suddenly said "Dad, not every boat has a life-boat, has it?" "How come?" I said. "Well, the life-boat could have a smaller life-boat, but then that would be without one." It turned out that the students with problems were those who had had prior exposure to FORTRAN, and the source of their difficulties was not that FORTRAN did not permit recursion, but that they had not been taught to distinguish between the definition of a programming language and its implementation and that their only handle on the semantics was trying to visualize what happened during program execution. Their only way of "understanding" recursion was to implement it, something of course they could not do. Their way of thinking was so thoroughly operational that, because they did not see how to implement recursion, they could not under-

stand it. The inability to think about programs in an implementation-independent way still afflicts large sections of the computing community, and FORTRAN played a major role in establishing that regrettable tradition.

But despite its shortcomings, FORTRAN was a great contribution whose significance I realized during aforementioned programming course in the 60s. In that course I developed by way of example the program that transforms a permutation that is not the alphabetically last one into its alphabetical successor. I still vividly remember my excitement that I could do so at the beginning of the 5th lecture for an audience of about 250 novices, and that it took no more than 20 minutes. I was so thrilled because, less than 10 years before, I had tackled that problem and, though by then an experienced programmer, I had been unable to design that program and after a few hours had been forced to give up. For me it was a most vivid illustration of dramatic progress within a decade, and I tried to convey some of that excitement to my audience, but in that effort I was not entirely success-

ful, for while leaving the lecture hall afterwards, I overheard one student remarking to another "If Dykstra couldn't solve that problem in his student days, he cannot have been a very bright one.". The reason I had been unable to design that program was that at the time I was programming in a machine code in which, due to the absence of index registers, programs had to modify their own instructions in store, and, as a consequence, my poor, uneducated head did not know how to make the distinction between the program on the one hand and the variables it operated upon on the other hand. In imperative programs, we now take this distinction for granted, but I would like to report to posterity that for me it was a revelation when I saw the distinction made. And FORTRAN had everything to do with that.

The concept of the self-modifying program has been vigorously promoted by John von Neumann who apparently felt that it was an essential ingredient of the computer's flexibility; in retrospect it was a pun that has been responsible for a decade of confusion.

And then the 60s started with an absolute miracle, viz. ALGOL 60. This was a miracle because on the one hand this programming language had been designed by a committee, while on the other hand its qualities were so outstanding that in retrospect it has been characterized as "a major improvement on most of it successors" (C.A.R. Hoare). It was created by a half-American, half-European committee of a dozen people. As I was not a member, I don't know who contributed what, but I know that we owe to John W. Backus that the syntax of ALGOL 60 got a formal grammar, and to Peter Naur that – among probably a whole lot more – the Report was written so exceedingly well. Several friends of mine, when asked to suggest a date of birth for Computing Science, came up with January 1960, precisely because it was ALGOL 60 that showed the first ways in which automatic computing could and should and did become a topic of academic concern.

The introduction and use of BNF (= Backus/Naur Form) to formalize the syntax was a very courageous novelty, because many feared that it would completely put off the computing community, but the great surprise was that

exactly the opposite happened: it was loved by programmers and implementers alike. It turned out to have all the properties of a helpful formalism, viz. compact, unambiguous and amenable to mechanical manipulation: before the end of the decade the construction of parsers had been mechanized, an achievement that 10 years earlier would have baffled the imagination. In the case of ALGOL 60, the use of BNF has had one regrettable effect. Its power should have been used exclusively to shorten the language definition, but it made the introduction of new syntactic categories so easy that the final syntax became more elaborate and more complicated than desirable.

A formal tool for the definition of the semantics was not available at the time, and so the semantics were given verbally. (This was definitely one of the places where Peter Naur's scrupulous use of English paid off.) It was, for lack of an alternative, still an operational definition, but much more abstract and further away from the machine than we were used to. It was definitely <u>not</u> a definition in terms of an implementation, because when the Report was published,

its authors did not know yet how to implement the language, and in that sense also the design of the semantics was very courageous.

Of the language itself I would like to mention 3 highlights, in order: recursion, the type boolean, and the block structure.

A major milestone of ALGOL 60 was its introduction of recursion into imperative programming.

Aside Its inclusion was almost an accident and certainly a coup. When the ALGOL 60 Report was nearing completion and circulated for final comments, it was discovered that recursion was nowhere explicitly excluded, and, just to be sure that it would be in, one innocent sentence was added at the end of Section 5.4.4, viz. "Any other occurrence of the procedure identifier within the procedure body denotes activation of the procedure.". Some committee members only noticed this sentence when it was too late to oppose, got very cross and refused to implement it. In more than one sense they were the losers. (End of Aside.)

To give you some idea of the significance

of this milestone, it was crucial in enabling Tony Hoare to complete the design of one of computing's most famous algorithms, viz. Quicksort. Prior to knowing ALGOL 60, he had the idea, but it remained an elusive vision, a dream that defied formulation, but as soon as he saw ALGOL 60 offering him just what he needed, Quicksort materialized in all its glory.

But it was not only the invention of individual algorithms that benefited, also program composition as a whole became much cleaner. For instance, once the implementation can cope with nested activations of the same procedure, the function supplied as parameter to an integration routine may itself be defined in terms of a definite integral, and by abolishing the combinatorial constraints that otherwise would be imposed, ALGOL 60 introduced a new standard of cleanliness. In passing it clarified the structure of compilers, for, the syntax being defined recursively, a bunch of mutually recursive procedures provides the simplest structure for a parser (so much so that "syntax-directed compiling" became a recognized technique). For the subsequent multiprogramming systems, in which nested activa-

tion was generalized to interleaved execution of different activations of possibly the same procedure, implementing recursion has been a valuable stepping stone.

The next milestone of ALGOL 60 that I must mention is ALGOL 60's introduction of "boolean" as a full-blown type, i.e. not only with boolean expressions but also with boolean constants, variables, arrays, procedures and parameters. From a programming point of view it was not so shocking: besides "integers" whose values can be represented by some finite number of bits, one can introduce a type for which a single bit suffices, but in a wider culture it is making a major difference.

Remark Even for programmers, who saw that there was no implementation problem at all, the introduction of the boolean variable was —and probably still is— a big leap, as is convincingly illustrated by the fact that for years one would still find even in published programs clauses like

```
if c = true then .....
```

where of course

```
if c then .....
```

would suffice. (End of Remark.)

Evolutions do take time. Mathematicians have used boolean expressions at least since 1557, when Robert Recorde introduced the equality symbol "=" as infix operator, but a formula like "n = 5" was not recognized as an expression, not syntactically and not semantically, because for lack of the type boolean, "n = 5" was not something that could have a value.

This changed in 1854 when George Boole introduced the type we now name after him, an invention, however, that the mathematical community had not yet absorbed a century later: for the average mathematician, and even for the not so average one, in 1954, a boolean expression was not something with a value, but, as in $0 \leq n \leq 10$, a condition or a statement of fact, and, say, $2 + 3 = 7$ was not a complicated way of writing the constant false, it was just wrong, an attitude that precludes the manipulation of uninterpreted formulae containing boolean subexpressions. I now see more and more people doing calculational mathematics such as formulating a theorem as a boolean expression and calculating that its value is true.

This new style of doing mathematics embodies a major simplification of the mathematical argument. Computing Science developed it because it needed it and had the potential to do so; ALGOL 60's promotion of the type boolean has played a historically significant role in that development.

Finally I must mention ALGOL 60's contribution to program structuring. On the statement scale it introduced conditional and repetitive clauses, to which I shall return later. On a more macroscopic scale of structuring it introduced procedures as program modules, but in this respect its greatest contribution was undoubtedly the block structure, an invention of the last weeks of 1959.

Mid 1959 I attended a preliminary meeting at which the notion of "scope" was still very tentative. Computing Science was really still in its infancy, for there was a constant confusion between the timeless concept of the program text and the temporal concept of the computation, a confusion that provoked someone to ask "Do I understand that we are considering scopes extending from here till then?" I re-

member, when I received the final document, the awe with which I observed how in the block structure the textual concept of scope had been related to the nested life times of incarnations. It was a beautiful synthesis, relating computation structure to program structure; as such it represented a quantum leap in our coming to grips with the programming task. I was deeply impressed by the block structure, but have never been able to find out who in particular has been responsible for its invention.

After 1960, academic computing science began to flourish; it did so with many different flowers, so many in fact, that here we can touch on only a very small fraction of them. The now obvious significance of formal grammars raised interest in parsing, recognition, finite automata, Turing machines and, particularly in the United States, in logic, decidability and complexity theory. Not surprisingly, those early flowers could bloom because they could be fertilized by mathematical theories that did already exist. Furthermore it should be noted that —surprisingly or not, probably not— most

of the early theoretical work was closely tied to the notion of a discrete machine and its sequential operation.

But at the same time people began to realize that, for making assertions about computations as they could evolve in time, it was desirable to learn how to reason effectively about the timeless concept of a program. In the 60s one could observe a shift in emphasis: as I have characterized it before, first it was the task of the programs to instruct our machines, a decade later it was the task of the machines to execute our programs.

Peter Naur from Copenhagen was around 1965 the first to contribute to the art of reasoning about programs in the little article in which he launched his "general snapshots" (which later would be called "assertions"). Due to the limited circulation of the journal BIT in which it was published it did not get the attention it deserved. He was followed by Robert W. Floyd from Stanford with the paper on assigning meanings to programs. This was a fundamental paper in that it tackled both correctness and

termination: the fact that he phrased the problem for programs as unstructured as an arbitrary flow chart made his paper at the same time general and unattractive. Half-way the decade, the goto-statement had been identified as a complexity generator and by ignoring it Tony Hoare, at the end of the decade, nicely tied the proof obligations to successions, conditionals, and repetitions, three concepts that are explicitly reflected in the syntactic structure of the program. It was at the time, however, unclear whether Tony Hoare's axiomatic basis for computer programming was based on a programming language semantics defined — presumably operationally — somewhere else, or whether all by itself and on its own, it could be taken as the definition of the language semantics. A further departure from operational considerations took place in the 70s with the introduction of predicate transformers. They allowed the definition of programming language semantics without any reference to computers, their components or, most importantly, their activity, and thus it became possible to reason about programs while ignoring that the program text admits the interpretation of executable code as well.

Programming was becoming of age.

Of course it is not enough to know how, in principle, one can reason about a program, for in practice the amount of reasoning required must be doable. The period I described reduced the amount of reasoning required by two devices, viz. structuring and abstraction. The latter abstracted away from the combinatorial explosion inherent in considering individual computations, i.e. sequences of individual states, while structuring contributed to the disentanglement of arguments.

Life continued and program structure became more widely exploited. Dana S. Scott showed the relevance of lattice theory to the programming world, which, attracted by its simplicity, then used it for its own purposes: in the refinement calculi it found ways of using structure to exploit the kind of monotonicity arguments lattice theory supports.

In some ways programs are among the most complicated artefacts mankind ever tried to design, and personally I find it fascinating to see that reasoning about them is so much aided by simple, elegant

devices such as predicate calculus and lattice theory. After more than 45 years in the field, I am still convinced that in computing, elegance is not a dispensable luxury but a quality that decides between success and failure; in this connection I gratefully quote from The Concise Oxford Dictionary a definition of "elegant", viz. "ingeniously simple and effective". Amen. (For those who have wondered: I don't think object-oriented programming is a structuring paradigm that meets my standards of elegance.)

On the whole, the period after 1970 showed a lot of progress. Pascal, designed by Niklaus Wirth from Zürich, had a wholesome influence worldwide on the teaching of programming, we got typed functional programming languages, logic programming entered the scene and disciplines for parallel programming emerged and were taught.

In retrospect it looks like a kaleidoscopic activity! It was. The turning point had been the NATO Conference on Software Engineering at Garmisch-Partenkirchen in 1968, a conference that had been organized by Friedrich L. Bauer from Munich. It was there and

then that the so-called "Software Crisis" was admitted and the condition was created under which programming as such could become a topic of academic interest.

The latter, not surprisingly, turned programming from an intuitive activity into a formal one in which the program is derived by symbolic manipulation from its functional specification and the chosen form of the correctness proof. In this connection the names of Ralph-Johan Back from Turku, Richard S. Bird and Carroll Morgan from Oxford and Wim Feijen from Eindhoven must be mentioned (with my apologies to all others whose names have been left out). David Gries published his "The Science of Programming", Journals like "The Science of Computer Programming" began to appear, and in the mean time the world has seen 4 conferences devoted to "The Mathematics of Program Construction", held in the Netherlands, England, Germany and Sweden respectively.

As you may have noticed, with the exception of Ithaca, the home town of David Gries, that I left out, all affiliations are European. We must come to the conclusion that, like Love,

the mathematization of computer programming is a bad sailor. At the American side the whole idea of correctness by construction, of developing the program and its correctness proof hand in hand has been rejected, and we have to understand why, for if we are planning our future, we had better establish whether the idea has been rejected here because the idea is no good or whether this society has rejected it because it is not ready for it. I'm afraid that all sorts of symptoms point into the direction of the latter.

- The ongoing process of becoming more and more an amathematical society is more an American specialty than anything else. (It is also a tragic accident of history.)

- The idea of a formal design discipline is often rejected on account of vague cultural/philosophical condemnations such as "stifling creativity"; this is more pronounced in the Anglo-Saxon world where a romantic vision of "the humanities" in fact idealizes technical incompetence. Another aspect of that same trait is the cult of iterative design.

- Industry suffers from the managerial

dogma that for the sake of stability and continuity, the company should be independent of the competence of individual employees. Hence industry rejects any methodological proposal that can be viewed as making intellectual demands on its work force. Since in the US the influence of industry is more pervasive than elsewhere, the above dogma hurts American computing science most. The moral of this sad part of the story is that as long as computing science is not allowed to save the computer industry, we had better see to it that the computer industry does not kill computing science.

But let me end on a more joyful note. One remark is that we should not let ourselves be discouraged by the huge amount of hacking that is going on as if computing science has been completely silent on how to realize a clean design. Many people have learned what precautions to take, what ugliness to avoid and how to disentangle a design, and all sound structure systems "out there" sometimes display is owed to computing science. The spread of computing science's insights has grown most impressively, but we

sometimes fail to see this because the number of people involved in computing has grown even faster. (It has been said that Physics alone has produced computing ignoramusses faster than we could educate them!)

The other remark is that we have still a lot to learn before we can teach it. For instance, we know that for the sake of reliability and intellectual control we have to keep the design simple and disentangled, and in individual cases we have been remarkably successful, but we do <u>not</u> know how to reach simplicity in a systematic manner.

Another class of unsolved problems has to do with increased system vulnerability. Systems get bigger and faster and as a result there comes much more room for something to go wrong. One would like to contain mishaps, which is not a trivial task since by their very structure these systems can propagate any bit of confusion with the speed of light. In the early days, John von Neumann has looked at the problem of constructing a reliable computer out

of unreliable components but the study stopped as von Neumann died and transistors replaced the unreliable valves. We are now faced with this problem on a global scale and it is not a management problem but a scientific challenge.

If I were younger I would try to find a mathematically meaningful definition for the complexity of interfaces between the components of proofs and other sophisticated discrete systems, and I would look for lower bounds, but I leave this in confidence to the next generation: either they will come with a more beautiful theory than I would ever have concocted, or they come with something of more significance.

I thank you for your attention.

[Keynote address to be given on 1 March 1999 at the ACM Symposium on Applied Computing at San Antonio, TX]

prof. dr Edsger W. Dykstra
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
USA