L2

CWI

AM

Centrum voor Wiskunde en Informatica

REPORTRAPPORT

Conference on the history of ALGOL 68

G. Alberts (ed.)

Department of Analysis, Algebra and Geometry

Historical Note AM-HN9301 January 1993

Conference on the history of ALGOL 68

G. Alberts (ed.)

Department of Analysis, Algebra and Geometry

Historical Note AM-HN9301 January 1993

Contents

Introduction to the Conference

Gerard Alberts

Among the many records, recordings and minutes one is missing: the very announcement of ALGOL 68, the occasion which gave it '68 rather than any other affix. Aad van Wijngaarden had been invited to talk on ALGOL 68 at the 1968 IFIP Congress in Edinburgh. Thus ALGOL X had unalterably become ALGOL 68. Van Wijngaarden addressed an overloaded lecture hall, expectations were high. He did raise the room (cf. the reflections by Kees Koster in this volume and by Heinz Zemanek [7] and Wladislaw Turski [8]). Probably, in the whole history of this programming language, there has not been a second moment of such glory. The address was never published.

The quarter century that elapsed since the presentation of ALGOL 68 is worth noting. And the 47th anniversary of the Stichting Mathematisch Centrum is the adequate occasion to do so. Strong Dutch influence tainted this language and, conversely, ALGOL 68 has been of great importance to the Dutch computing community and to the Mathematical Centre (as the CWI was called at the time) in particular. The leading role of Aad van Wijngaarden, then director of the Mathematical Centre, in its design explains why to the CWI looking at the history of ALGOL 68 is tantamount to regarding part of its own past.
In the Netherlands ALGOL 68 served as a point of identification, both positive and negative, and it gave direction to frontline research in computer science. Should one say that it is the Dutch among programming languages? Taking into account the relative number of computer scientists "speaking" it and adding to this the fact that its users are proud of speaking so many foreign languages, the comparison is quite convincing. Then again ALGOL 68 has these distinct qualities of rationality and elegance never attributed to Dutch - except by Simon Stevin [1]-. Moreover it was designed, one might say overloaded with design, by a committee. It has become classic without ever reaching real widespread use. My suggestion is that it is the Sanskrit of informatics. The future computer scientist as well as the historian of computing who would know the truth about programming languages, will always return to ALGOL 68. This conference is meant to contribute to the documentation of its history.

Survey

What one hopes for in bringing together different perspectives on one topic in a conference, already gleams in the papers presented here, the more so if we may take into account Charles Lindseys report to the ACM, to be published in the *SIGPLAN Notices* of March 1993 [10]. The combination allows for a synthesis and a first historical evaluation of ALGOL 68. Lindsey presents us with an extremely well-informed and technically evaluating chronicle of the timespan between settling the last details of ALGOL 60, in 1962, and the last issue of the ALGOL Bulletin in 1978. Friedrich Bauer, in his contribution, embeds this in a broad historical perspective on programming and programming languages. Lambert Meertens continues that line of thought to the present, focusing on the zeal of designing elegant languages. By their recollections Kees Koster and John Peck add to Lindseys account, their piercing views make the dramatis personae leap from the pages. The reception of ALGOL 68 is treated for the Netherlands by Sietse van der Meulen and for the USSR by Rar, Bulyonkov and Terekhov. The impact in the USSR has been reported from both Novosibirsk and St. Petersburg. The result is a lively, although fully inside view - except for mention of an "iron curtain" there is no reference to what was going on "in the real world" in 1968. It remains a hard task here, just as for the whole of history of computing, to sensibly relate the developments in informatics and automation to the cultural and social events of the time. No attempt to that end is made here.

Personal notes and other aids to interpretation

Peck and Koster linger long enough to state in full what is implicit with Lindsey, viz. that ALGOL 68 emerged as van Wijngaardens project. He was the boss and his methods of maintaining power were not always subtle. Wirth must have suffered on occasions. Moreover the reader of the quotes presented by Van der Poel in [6] cannot fail to perceive the scientific generation- conflict which gave the design process in IFIPs Working Group 2.1 its special dynamics. In an interview, shortly before he passed away, van Wijngaarden readily admitted that he might have been too extreme and too uncompromising: “Algol 68 may show some traces of that attitude. But listen: being a mathematician, once one wants to create something, of course one would make a thing of beauty."[9] Bauer in turn, confronted with this attitude in the WG2.1, is reported to sigh: “The words you use are so ambitious"[6].

Pervading all the contributions are two striking features. First is the fact of so many recordings and minutes, exposing a high degree of selfconsciousness. Second is the repentant undertone. “It may be true that a committee wastes hours, it keeps minutes"[6]. And minutes were kept. Van der Poels anthology from these is no exception in constantly stressing the fights and tensions in the Working Group. The reader is time and again implicitly remembered of how bad the "we" behaved. The silent question is how the author and his colleagues could have treated each other so nastily - less than thirty years ago. Such question, however, is ill-posed. The proper question should be, why in retrospect the WG 2.1-members involved reproach themselves and seem to deplore their behaviour. Is such behaviour not what normally happens in committees? Then why in their own retrospective judgement should they have been saints? Being reasonable, rational scientists, they expected themselves to do better. After all, they were the pioneers of modern technology and at the forefront of the search for truth in this domain. The combination of pride and responsibility is not hard to imagine.

Points of synthesis

A higher goal was cherished in WG 2.1. It was out for the truth, for the true universal programming language. The result was a twofold success: a universal programming language and the insight that no truth exists in this respect. Such result can of course only be gained once. Superposition of the perspectives offered by Bauer and Koster upon Lindseys rich material suggests that a certain development culminated in this algorithmic language. There seems to reign a touch of inner logic in the development of computer science, deciding that programming gains recognition as an autonomous subject only gradually, and that once this occurs - say from 1955 onwards - the algorithmic perspective is the first to be developed. As Bauer puts it: ".. algorithm is more directed towards the essential content. Mathematicians therefore frequently prefer it; algorithmic languages are (or should be) programming languages under some abstractions from notational details." Now we can understand why the strive for machine-indepence, for universal language and for algorithmic approach could so well coincide; why WG 2.1 had no trouble in believing that the language had been more or less decided upon, independent of van Wijngaardens description; and finally how van Wijngaardens extremely mathematical approach could drive things to the paradoxical result that content and description became fully entangled, instead of independent. Therefore ALGOL 68 had to be both the limit and the end of the algorithmic era in informatics.

The malcontents within WG 2.1, writing a minority report and creating what was to become WG 2.3 on Programming Methodology, in fact stood up for the next era; whether one calls the next episode that of programming methodology, of systems development, or of software engineering is a matter of taste and viewpoint - and of further historical research -. Turski in retrospect [8] notes that within WG 2.1 from the draft MR 93 onwards common ground was simply lost, because the critics did not understand tenor of the desciption method and the authors, unaware of that fact, basicly resented the misdirected comments. Put otherwise: if one party claimed to search for truth, the very process the committee was in, is there to testify the lack of ultimate truth in this newest and most modern of sciences. In the process, of course, neither party could perceive the paradox noted above.

Maybe the participants in the vehement debates of WG 2.1 did intend to be personal in their remarks; however the purport of controversy is quite recognizable as a logical one, as growing pains, in the development of computer science.

Hence, in a somewhat detached long-term perspective the history of ALGOL 68 is marked by two milestones. One was van Wijngaardens introduction of orthogonal design and two level grammar [2], where he drove home the mathematical import of the algorithmic approach. The other was the informal creation of WG 2.3 on programming methodology during the meeting in Scotland, North Berwick, August 1968 (cf. [10]) in the week before van Wijngaarden gave his glorious presentation before the IFIP community.

References

[1] Simon Stevin 'Uytspraeck over de Weerdicheyt der Duytsche Tael/Discourse on the Worth of the Dutch Language'(Introductory chapter to *The art of Weighing.* Leyden: Plantijn, 1586). E.J. Dijksterhuis e.a. (eds.) *The principle works of Simon Stevin* (V vols.) Amsterdam: Swets en Zeitlinger, 1955-1966, Vol. I pp. 58-93.

[2] A. van Wijngaarden 'Orthogonal design and description of formal language' *Report MR 76.* Amsterdam: Mathematical Centre Computing Dpt., 1965

[3] J.W. de Bakkker e.a. *MC-25 Informatica Symposium.* Amsterdam: Mathematisch Centrum (Mathematical Centre Tracts 37), 1971.

[4] J.W. de Bakkker and J.C. van Vliet (eds.) *Algorithmic Languages. (a tribute to Prof.Dr.Ir. A. van Wijngaarden on the occasion of his retirement from the Mathematical Centre).* Amsterdam: North-Holland, 1981.

[5] H. Zemanek (ed.) *A quarter century of IFIP; The IFIP Silver Summary.* Amsterdam: North-Holland, 1986.

[6] W.L. van der Poel 'Some notes on the history of ALGOL'. In [3]; reprinted in [5: pp. 373-392].

[7] H. Zemanek 'The Role of Professor A. van Wijngaarden in the History of IFIP'. In [4: pp. 1-28]; [5: pp. 303-331]

[8] W.L. Turski 'ALGOL 68 revisited twelve years later or from AAD to ADA'. In [4: pp. 417-431]

[9] G. Alberts and P.C. Baayen 'Ingenieur van taal; interview met A. van Wijngaarden' (= Language engineer). In: G. Alberts, F. van der Blij en J. Nuis (eds.) *Zij mogen uiteraard daarbij de zuivere wiskunde niet verwaarloozen.* Amsterdam: Stichting Mathematisch Centrum, 1987, pp. 276-288.

[10] C.H. Lindsey 'A History of ALGOL 68'. In: *SIGPLAN Notices* March 1993 (to appear, abstract in this volume).

History of Programming Languages, a Survey

Friedrich L. Bauer
Technische Universität München

1 The Origins of Programming and of Programming Languages

The word *program(me)* denotes today in common usage 'a list of planned activities, a plan for further action' (Longman), a written display of goal-oriented principles, stipulated inferences, previewed run-offs.

The word of greek-latin descent came up in 19th century with the prevailing meaning 'public announcement, proclamation, agenda'. While the two latter meanings, dominating in 'party program' and 'theater program' , are rather specific, the meaning of *program(me)* in todays Computer Science is closer to the ethymology of *pro-gráphein* 'prescribe':

Greek *graphma, gramma* means 'precept, prescription, specification, instruction, rule, order, law'. *prescribe* (German *vorschreiben*, French *prescrire*, Russian *predpisivat* stands for (Longman) 'to state what must happen or be done in certain conditions'; *precept* (German *Vorschrift*, French *précepte*, Russian *predpisanie* stands for 'a guiding rule on which behaviour, a way of thought or action is based' (Longman), 'a principle governing action or procedure' (Merriam-Webster). *Action* as well as *procedure* mean 'something done, effected or accomplished'.

1.1 Where did the term *program(me)* in its Computer Science meaning turn up first? One would not expect it earlier than in Babbage's work, but in fact in his paper of Dec. 26, 1837, where the Analytical Engine, the 'mill' is outlined, the word is not used, nor does it show up in the work of Ada Countess Lovelace, who speaks on the other hand of 'weaving algebraic patterns'. Nevertheless, Babbage does discuss e.g. a routine for calculating Bernoulli numbers, using as building elements instructions such as '$V_4 \times V_4 = V_{10}$' with operation signs like $\times$ appearing on an 'operation card' and subscript numbers like (4, 0, 10) appearing on a separate 'variable card'. That Babbage has understood programs to be entities can be seen by his speaking of 'small pieces of formulae previously made by the first cards'.

Likewise, Percy E. Ludgate (1909) and Leonardo Torres y Quevedo (1914) have no terminological need for 'programs'. Torres, however, uses the term *automaton* and says 'the automaton acts exactly as an intelligent being who follows certain rules', his point is that it chooses the action and follows one or another path. He uses a 'diagram' for showing how this is done (fig. 1), indeed his program is a circuitry. Programming by circuitry was later the daily business of Hollerith type tabulating machines.

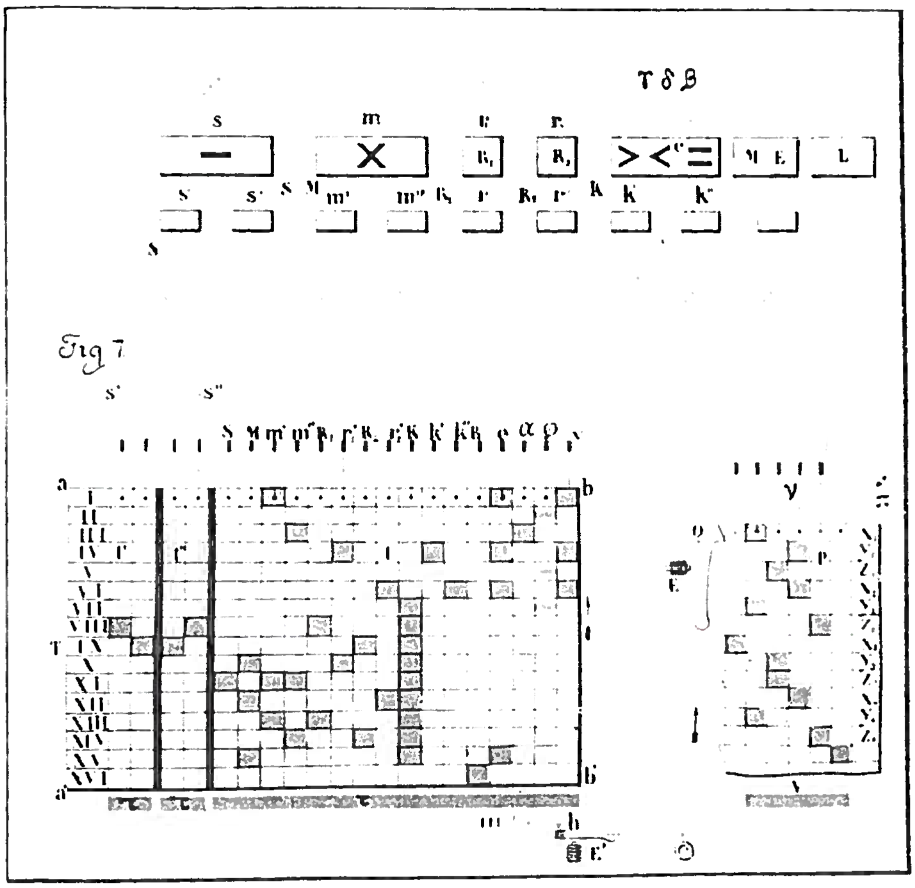


fig. 1 Torres y Quevedo's program circuitry

Typically, Torres thinks of sequences of actions between branchings. That programs can split into trees of actions or actions can perform collaterally until they are collected into one path is beyond the machinery he is accustomed to using.

1.2 In the definition above, the term 'plan' is used. *plan* means (Merriam-Webster) 'a method devised for making or doing something or achieving an end' and thus is rather parallel to *program.* In fact, the German inventor Konrad Zuse uses systematically *plan* (German *Plan*) in his German Patent Application Z23 139 IX/42m from April 11, 1936; his *computing plan* (*Rechenplan*) is strictly sequential and has no branching (fig. 2). Heinz

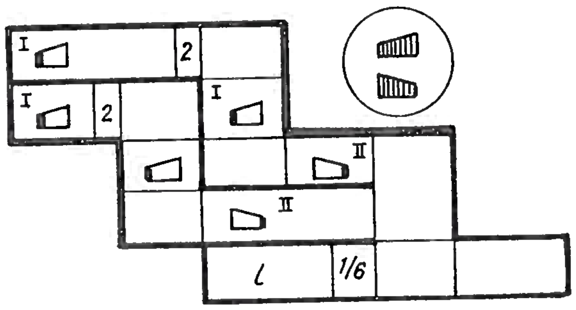
Rutishauser in Zürch took over from Zuse the term *Rechenplan*. The use of German *Rechenplan* in Germany for this purpose was not entirely new, however more frequently the terms 'Rechenschema' or 'Rechenformular' and 'Formblatt' would be used. This was meant to provide space for recording the intermediate results of some calculation, the individual steps of which were to be performed using a 4-species calculator; the sheet did indicate which steps to be taken. In some areas of scientific computations, such as geodesy and orbit astronomy, rather complicated calculations were organized in this way; with the particular motive to minimize the number of recordings, which was supported by the use of multiple-carriage-machines like the THALES-GEO , the BRUNSVIGA -HAMANN DUPLEX and the BRUNSVIGA-TRINKS TRIPLEX. Konrad Zuse started his thinking about a program-controlled computer under the impression of tiring numerical work he had to do in civil engineering statics.

Operation No.	Adress of cell for operands		Type of basic arithmetic operation	Adress of cell containing result
	1	2		
1.)	1	5	Mult.	0
2.)	0	9	"	10
3.)	2	6	"	0
4.)	0	7	"	11
5.)	3	4	"	0
6.)	0	8	"	12
7.)	1	6	"	0
8.)	0	8	"	13
9.)	2	4	"	0
10.)	0	9	"	14
11.)	3	5	"	0
12.)	0	7	"	15
13.)	10	11	Add.	0
14.)	0	12	"	0
15.)	0	13	Subt.	0
16.)	0	14	"	0
17.)	0	15	"	$\underline{\underline{0}}$ = Result

fig. 2 Zuse's program for a 3 by 3 determinant

1.3 A third word comes into play: *algorithm.* Its origin points to Al-Choresmi, the Usbekian scholar who lived around 800 at the court of the caliphes and was by his books, translated into Latin, highly influential for the development of mathematics in Europe since 1200. The algorithm of multiplication (with denary or binary numbers) is an expression still used by Leibniz, the algorithm of extracting the square root had to be learned at secondary schools. They and the algorithm, described already by Euklid, for forming the greatest common divisor of two natural numbers, they all are programs; usually stated in verbal form or even by showing only an illustrative example. Only rather late, the word comes again into use in its very general meaning, Householder and Faadeva, around 1950, do not yet use it in their Numerical Analysis monographs.

While *program* and *computing plan* rather synonymously concentrate more on the write-up, *algorithm* is more directed towards the essential content. Mathematicians therefore frequently prefer it; algorithmic languages are (or should be) programming languages under some abstractions from notational details. In this sense, we are going to use these words interchangingly.



$$\int_0^l M^I \cdot M^{II} ds = \frac{l}{6}\left[M_l^{II}\left(2M_l^I + M_r^I\right) + M_r^{II}\left(M_l^I + 2M_r^I\right)\right]$$

fig. 3 Zuse's computation sheet

1.4 All the early examples of programs, plans and algorithms given above fall into a particular class: the class of **state-oriented** programming and programming languages. It just happens that human performance, restricted as it is with respect to mental concentration, makes it advisable to do "just one thing at any time" and thus at any moment the human calculator is occupied with just one task. If asked "where are you working" there will always be one and only one state. This worthwhile caution with respect to human operators was fatuously applied also to mechanization and automatization; quite unnecessarily, since machines do not think and therefore can be much more reliable than humans.

The transition between states is done by the execution of **commands** or **instructions**, fittingly one speaks in this context also of **imperative** programming languages. The **machine language** of early machines was almost exclusively an imperative one.

There is an obvious generalization of this **single-state-oriented** programs to **multiple-state-oriented** programs. They describe, what happens if a group of human operators works jointly and under division of labor on one task. Each person will have its **individual** program, but it also needs an overall **organization** program. Such work was done e.g. 1938 in New York in a large project for the calculation of tables, using hitherto unemployed persons; and during World War II, Alwin Walther in Darmstadt organized a group of women from the Aids Corps to do a network flow of calculations. No documentation of the organizational details has survived.

1.5 While state-oriented programming is, or at least was always close to machines in some sense, there is a very different attitude found among logicians and pure mathematicians. In the world of functions, propositions and predicates, there is no need and also no room for states. Where Torres y Quevedo uses fig. 1 to indicate his program, a mathematician is quite satisfied to write $\alpha = a \times x \times (y - z)^2$ instead; and for Zuses Rechenplan

$$\begin{aligned} a &\Rightarrow V_1 \\ b &\Rightarrow V_2 \\ V_1 : 2 &\Rightarrow V_1 \\ V_1 \times V_1 &\Rightarrow V_4 \\ V_4 - V_2 &\Rightarrow V_5 \\ \sqrt{V_5} &\Rightarrow V_5 \\ V_1 \times (-1) &\Rightarrow V_7 \\ V_7 + V_5 &\Rightarrow V_8 = x_1 \\ V_7 - V_5 &\Rightarrow V_9 = x_2 \end{aligned}$$

the mathematician prefers to write a *functional* ('algebraic') program

$$(a:2) \times (-1) + \sqrt{(a:2) \times (a:2) - b} = x_1$$
$$(a:2) \times (-1) - \sqrt{(a:2) \times (a:2) - b} = x_2$$

and finds it at least helpful, possibly even preferable to be told that it means

compute the roots x_1 and x_2 of $x^2 + a \times x + b = 0$.

Why did Zuse then bother to write the 'complicated' state-oriented program, and how could he even be proud to accomplish it? It is because the machine he had invented could be controlled by his imperative program, but not by the mathematicians functional program. He was wrong in as far that the mathematician Rutishauser only a few years later was able to write a programming program which translated the functional program the mathematician likes and understands better into a state-oriented program that Zuses machine Z4, working at Zürich, understood. Thus, Rutishauser showed that there exists the meaningful concept of a **functional program** and of **functional programming**. Zuse, however, did not appreciate this. Samelson and Bauer obtained with priority 1957 even a patent for a machine that directly obeys functional programs of this simple, purely applicative form.

That a mathematician prefers the functional write-up, is of course to a large extent also due to the fact that she or he is used to this; any time human perception is involved, habits play a decisive role. If the mathematician would however declare that he also considers the construction above

$$\text{compute the roots } x_1 \text{ and } x_2 \text{ of } x^2 + a \times x + b = 0$$

to be a program, caution is advised: clever mathematical systems today may break this down into an algorithm, but will not do it with an arbitrary 5th order equation with unknown coefficients. Even

$$\text{give in the form of terms all the roots of } x^{31} - 1 = 0$$

will cause great difficulty to some of them.

We have seen in the last example programs and a programming style of a third sort, called **descriptive**. The history of programming languages can not yet say much about them, since some of their aspects are still a research subject.

1.6 Quite innocently in the beginning, logicians did run in the 30's into studying programs. The reason was that after Kurt Gödels revolutionary result in 1931 about *incompleteness* (formale Unentscheidbarkeit) of some systems, e.g. arithmetic, *computability* also became an issue. In 1936, independently two persons conceived a convincing definition of what 'computable' should mean: Alan Mathison Turing (1912-1954) and Emil Leon Post (1897-1954). Strikingly, they came out with almost an identical proposal:

they both tried to imitate in very abstract form what a human being would do, reduced for matters of simplicity to the most elementary actions. Clear, that such a proposal would have the character of a machine, a hypothetical machine, a Gedanken-machine, not good for practical work, but well suited to theoretical studies about it. Anyhow, the programming languages of the Turing machine and of the Post machine are single-state languages. An attentive observer recognizes, however, that these machines, in contrast to the Babbage-Ludgate-Zuse line, store programs together with numbers and thus allow programs to be changed by themselves, an idea Zuse did not have and Aiken found so horrible that he fought it until the end of his life. Logicians showed that the Turing machine and the Post machine are equivalent. Why Post is rarely mentioned, remains unclear; both Turing and Post were somewhat strange characters.

In the same year 1936, a very different approach was done by Alonzo Church. He found it natural for a logician to start from expressions and from the work with expressions, and thus formulated a system (the 'Lambda Calculus') that could be called a functional machine, although he did not see it this way. As a functional language of application and abstraction, it included, well hidden, a feature that was somewhat controversial at that time and even later: recursion. After Ackermann had shown in 1928 that the great Hilbert was mislead in his intuitive power when he thought that so-called primitive recursion was the ***ultimate ratio*** and in this sense universal, general recursion became a dreadful instrument, in particular for those who did not understand it. Church then posed the thesis that there can be only one equivalence class of universal machines or programming languages, and all fruitful definitions should stand the test to be shown to be equivalent. This Church Thesis is generally accepted today.

A widely used class of non-universal languages is historically connected with single-state-oriented programming: programming with loops, to be more precise with a finite number of fixed loops. This dates back for several centuries with the construction of clocks and music instruments having a rotating barrel usually equipped with pegs that triggered action — a **single loop machine.** It was soon generalized to **multiple loop machines**, for example clocks with a second loop for the quarter hours, and until the middle of 20th century, multiple loop machines flourished, examples being automatic multiplication and division in mechanical desk calculators. In the Babbage-Ludgate-Zuse-line, fixed loop structure (Zuse:

'starre Rechenpläne') is assumed — Zuse's Z3 of 1941 even had only one loop; however the tape spliced to a loop could be exchanged manually. The more liberal von-Neumann-architecture of state-oriented machines allowed a system of loops, an arrangement to be set up with the help of a cleverly designed crutch, the jump instruction. Since a von-Neumann-program, in contrast to the stubborn Aiken doctrine, could change itself, the loop structure, although always present and finite, could change during computation. This made the von-Neumann-machine of 1945 technically universal; that it was so was recognized by Mauchly on the one side, by von Neumann on the other side more accidentally and made public quite later (1946, 1947).

A question that caused much trouble is the following: Do programming languages that are to be used in practice need to be universal? My experienced and intelligent friend Heinz Rutishauser thought for a long time that the answer is *No*, and I think he never convinced himself completely that the answer should be *Yes*. Samelson and myself said *Yes, but* ..... , indicating that full generality would have its price ('Do not let the simple, but frequent case pay the price for the complicated, but rare case'). We were afraid that full generality might thus hinder the acceptance of a programming language intended for practical use, like ALGOL. In the 70's, however, it became clear to us that it would no longer matter to pay the price: the price can be paid and should be paid. Most people agree today to this.

1.7 In the next chapter, where we shall deal with practical programming languages that developed in the 50's , grew in the 60's and matured in the 70's and 80's, we shall structure the events according to the scheme: state-oriented imperative programming, state-free functional programming, descriptive programming. There is a further dimension which positions the items: originally languages dealt with one sort of individuals only, with numbers — usually integers and even limited in size: fixed-point numbers, floating-point numbers. Multi-sorted programming arose late in the 50's, but this was not the end. Free definition of 'Data Types' was added in the 60's; programming with Abstract Data Types was studied by Liskov and Zilles 1974, Guttag 1975. From an area where one would not expect it, from simulation, a new philosophy came up in 1967 with SIMULA: that individuals and the operations defined on them should be considered to be an entity, the so-called classes in SIMULA. Now this entity is somewhat confusingly called **object** and **object-oriented** programming is at present the highest level of sophistication. It combines both with a state-oriented at the one hand, and with a functional style at the other hand.

Since a mixture of functional programming and state-oriented programming, if not stirred well, has a tendency to produce a turbid potion, blending such a mixture with object-oriented style does not make the situation much easier, but less transparent.

2 The Landscape of Programming Languages

Twenty of the early specific programming languages originating between 1945 and 1956 — contributions by Zuse; Goldstine and von Neumann, Curry; Mauchly; Burks; Rutishauser; Böhm; Glennie; Hopper; Laning and Zierler; Backus; Brooker; Shura-Bura, Kamynin, and Ljubimski; Ershov; Grems and Porter; Elsworth; Blum; Perlis; Katz; Bauer and Samelson; — have been thoroughly discussed by D.E. Knuth and L.T. Pardo at the History of Computing Conference in 1980. Altogether, Jean E. Sammet has listed over 700 programming languages that have been developed, proposed and described till 1966. Meanwhile, nobody will any longer count them.

2.1 Among the twenty early ones, the majority is imperative and very close to machine languages — so-called 'low level'. This list starts 1946 with the informal flow diagram language introduced by Goldstine and von Neumann, and is carried on by assemblers designed by Mauchly (1949), Burks (1950), Glennie (1952), Hopper (1953), Brooker (1954), Grems and Porter (1955), Elsworth (1955); each one for a different machine.

Machine independence, however, was an important goal from the beginning; programming languages that were not depending too much on particular machines were called 'high level' . Rutishauser opens this line with the first attempt to bring applicative pieces into an imperative language (1951), followed by Laning and Zierler (1953), this work fertilizes a few others that become highly important: IT by Perlis (since 1956), AT3 (later MATH-MATIC) by Katz (since 1956) and in particular FORTRAN by Backus (since 1954).

The high level imperative programming languages with functional features of application and somehow abstraction led via FORTRAN and ALGOL58 to ALGOL60 and then to the Algol family of ALGOL68 (1968) and PASCAL (1969); PASCAL branching again into MODULA-2, ELAN and ADA; BASIC (1965) and C being low-level relapses, COMAL an upgrading of BASIC and PASCAL. COBOL, a 'business oriented' sister of FORTRAN from 1959, is a relict that, blended with FORTRAN, lead to PL/I (1964). Several languages specialized on particular sorts, e.g. on Boolean values (LOGALGOL, 1961), on text strings (SNOBOL, 1963) or on matrices (APL, 1962).

Step by step, these imperative languages developed a richer reservoir of sorts and even the capabilities to define new sorts. Programming in problem-adapted complex data structures makes not only life easier, it also helps save storage. Furthermore, introducing corresponding problem-adapted complex operations helps to save time. Thus, the development towards abstract sorts and operations has an economical advantage, and the transition to *objects* in object-oriented programming is the natural conclusion. SIMULA (1967) was the first step in this direction, followed by full-fledged, however untyped SMALLTALK-80 (1980) and typed EIFFEL (Meyer, 1985). Today we see object-oriented extensions to classical imperative programming languages: OBERON (PASCAL), MODULA-3, C^{++} and OBJECTIVE-C.

In the beginning, imperative programming was strictly sequential (single-state programming). Under the work of Dijkstra and Hoare, multi-state programming captivated the hearts of many, although not all programmers and entered some programming languages, starting with ALGOL68 and leading to OCCAM, based on more theoretical calculi like CSP (Hoare), CCS (Milner), COSY (Lauer) (*parallel programming*). In a corner lives the subculture of Petri nets. General Recursion, by the way, is sometimes excluded, sometimes tolerated: Wirth, in *Algorithmen und Datenstrukturen*, 1975, dissuades the reader from using recursion ('Wo Rekursion zu vermeiden ist'). On the other hand, jump instructions are frequently placed at a disadvantage ('Go to considered harmful', Dijkstra).

There are also special programming languages for real-time applications, like PEARL and BASEX.

Zuse, who is at the beginning of our list, has not been discussed so far. His early *Plankalkül* shares the fate of the two Russian contributions, the *programming programs* of Shura-Bura, Kamynin, Ljubimski and those of Ershov. They have not had the impact they deserved, possibly due to world-political circumstances. Interestingly, they show certain parallels in notation; Plankalkül und the Liapunov notation the Russians used have a two-dimensional outlook. For the input devices that were available in the 50's, this was not very attractive and was one of the reasons that Rutishauser, Samelson and myself did not follow Zuse notationally. We did so, however, in our language philosophy as much as we thought to be appropriate, but this did not find Zuses approval.

Altogether, imperative programming, although several times said to be moribund, is still dominating in practical use; imperative programming languages are also dominating in number. This does not necessarily qualify them.

2.2 There remains in our list of twenty early birds, apart from the Samelson-Bauer machine, the logician Curry. His rudimentary programming language, deeply connected with his idea of 'combinators', is much more functional and less imperative than all the ones discussed so far. His functions can have several arguments and several results, thus common notation cannot be used. Apart from this strangeness, function composition (*program composition*) is his main instrument, and he is not refraining from recursion. His language, however, was not brought to the test of practical use. Function composition is a particular pure form of functional programming.

In most cases, functional programming is *applicative*. This is true for LISP, designed 1958 by John McCarthy. It has only one sort of individuals, lists which are formed by something, namely lists again or atomic symbols. Its individuals are therefore recursively defined, and General Recursion gives also all the power to the operations.

Pure LISP, e.g. SCHEME, has no program variables, i.e. does not know storage elements that can be overwritten, it is absolutely state-free. Declarations are made in Lambda-notation, thus a bridge to the ancient ideas of Church is established. Jean Sammet has correctly spoken of 'one of the most important ideas in programming'. Credit for this should also go to H. Gelernter and N. Gerberich. Unfortunately, McCarthy' s work was too late to be fed into the Zurich ALGOL Conference in May 1958, and in the Paris ALGOL Conference in January 1960, McCarthy was not strong enough to push his concept into ALGOL60, against a majority of state-oriented people, led by Perlis and van Wijngaarden.

LISP was for a long time the uncontested champion in its class. More recently, LOGO was created, which has advantages when used for didactic purposes. LISP is an untyped language often used in AI applications. In introductory courses at universities, typed variants like ML (Milner 1979), MIRANDA (Turner 1986), HASKELL (Hudak, Wagner 1990) are often preferred. LISP has also found object-oriented extensions, like COMMON LISP including CLOS. The lack of states brings the idea of object-orientation much better into effect.

Functional programming languages based on combinators are rare. They avoid even parameters in applications and know nothing but function composition. Curry's attempts might have led into this, had Curry continued them. John Backus has for a while been the advocate of variable-less programming. More recently, however, Curry's 'combinators' have found new interest.

2.3 Today, some people speak of two worlds — the world of imperative programming and the world of functional programming. But these worlds are not isolated. The Munich project CIP has studied in great detail the transitions between them. Only two problem areas should be mentioned:

In imperative programming, a program variable may change its value. In applicative programming, this may not happen. It will not happen if in imperative languages single-assignments only occur. If however a statement like

$$a \times a \Rightarrow a$$

occurs in the imperative world, it can be made harmless in the functional world by introducing, as was done in LUCID, appropriate subscripted variables which reflect the state history:

$$a_i \times a_i = a_{i'}$$

where i' denotes the successor of i. For single-state-oriented programs, integers will do it for the subscript family; this was already advocated by Rutishauser in the early 50's. For multiple-state-oriented programs, the state history will no longer be linear, but a general digraph; the subscript family should reflect this. There is a deep connection between these state digraph systems and the Kripke structures of the modal logics that belong to the branching time situations of the program run-off — connections that show up in the work of Zohar Manna.

Another difficulty arises with the structured objects in an imperative programming language: a program variable for a structure is not the same as a corresponding structure of program variables. Efficiency arguments suggest frequently to update single components instead of overwriting the whole variable. Pointer realizations that are used in this context lead to tricky situations; the field has been studied carefully by Bernhard Möller. Since storage is getting more and more cheap, functional programming style, which avoids these risks, is increasingly advantageous.

2.4 Finally, a third world of programming is to be mentioned: *descriptive programming style.* Some of its effective forms go under the names *predicative programming* or *logical programming.* Executing such a program means to give a proof of an *existence formula*, e.g.

$$\exists x.\ x^2 + a \times x + b = 0 \quad .$$

Descriptive programming is particularly appropriate in the specification process that should always precede a programming process; this has also been studied to some length in the Munich CIP project. Other specification languages are CLEAR (Burstall, Goguen 1977) and LOTUS (1983), an extension of CCS.

Logical programming is necessarily, even in first order logic, subject of Gödel type limitations of decidability and therefore needs syntactical restrictions in order to be safe. Particularly simple, although in some cases inconvenient, is a restriction to Horn clauses, which is safe und nevertheless preserves universality; it has found application and some admirers in the PROLOG family and in other inference systems.

Related are special programming languages for mathematical proofs and in connection with this for program specification and development; among others LCF ('Logic for Computable Functions') and SML by Milner. There are many derivatives, e.g. AUTOMATH (De Bruijn 1975), NUPRL (Constable 1986), DEVA (Sintzoff, 1990), LEGO (Burstall 1990).

Other forms of descriptive programming showing up in a variety of mathematics-oriented applications have not yet established programming languages in the proper sense.

2.5 Many interesting details of the History of Programming Languages cannot be discussed here in the given framework. Also, some technical aspects, e.g. modularity, encapsulation, scoping, remain undiscussed here.

The History of ALGOL 68

C.H. Lindsey
Honorary Fellow, University of Manchester

The History of ALGOL 68.

C. H. Lindsey
Honorary Fellow, University of Manchester.

Extended Abstract

The full text of this paper is to be published in the March 1993 issue of SIGPLAN Notices, and it is to be presented at the forthcoming ACM History of Programming Languages Conference.

ALGOL 68 was developed by Working Group 2.1 of I.F.I.P., the Working Group itself being formed from the authors of the original ALGOL 60 Report. The development took place over the period from 1965–1969, followed by revision of the language from 1970–1974. Active support continued up to about 1979.

Ideas for a future language, which became known as ALGOL X, were being discussed by the WG from 1964 onwards. There was also a language ALGOL Y – originally it was a language which could modify its own programs, but in actual fact it turned out to be a "scapegoat" for features that would not fit into ALGOL X. By 1965 at Princeton they thought they were ready, and accordingly they solicited drafts of a complete language to be presented at the next meeting.

At the next meeting there were officially three full drafts on the table, from Wirth, Seegmüller and Van Wijngaarden, although there was also a contribution on record handling from Hoare. Van Wijngaarden's document was entitled "Orthogonal design and description of a formal language". The Working Group resolved to adopt Van Wijngaarden's method of description. These 4 authors were now charged to write the Report for the language.

The history of what followed is very much concerned with parameter passing. ALGOL 60 has call-by-name, which is used both for result parameters and for tricks such as Jensen's device. Initially, the Working Group tried to indicate call-by-name at the point of call, rather than in the procedure heading. There were also proposals for call-by-reference, and to use procedure parameters instead of Jensen's device. Hoare proposed call-by-value-result, or "anti-value" as he called it at the time. Hoare's records, and references to them, could construct the usual dynamic data structures, but, quite deliberately, the records could only be created on the heap, and therefore the references could not point to local variables.

When the subcommittee of Wirth, Hoare, Seegmüller and Van Wijngaarden met, there were two competing drafts on the table, one describing the right language and the other using the right formalism. But the subcommittee became completely split over the method of parameter passing, because of Van Wijngaarden's principle of 'Orthogonality' and its associated method of parameters passing which was at variance with Hoare's views on references. The issue was debated at the Warsaw meeting, but Van Wijngaarden was able to demonstrate that his system was entirely self-consistent. The meeting commissioned Van Wijngaarden to prepare the final Report, and to publish it in the Algol Bulletin. He was given discretion also in incorporate McCarthy's scheme for overloaded operators, if it was found to be feasible.

By the next meeting McCarthy's overloading was in "in all its glory". This is a fine example of a feature which Van Wijngaarden rejected when first proposed, as being too late and too much of an upheaval, but then incorporated by a major upheaval of the Report between meetings.

The Draft Report, MR93, was duly published in the Algol Bulletin in February 1968, and was the cause of much shock, horror and dissent, even (and perhaps especially) amongst the membership of WG2.1, and at an ALGOL 58 Anniversary meeting at Zurich in May it was "attacked for its alleged obscurity, complexity, inadequacy, and length, and defended by its authors for its alleged clarity, simplicity, generality, and conciseness". But the document could be read. I read it myself, and succeeded in extracting the language from it, which I wrote up in an informal paper entitled "ALGOL 68 with Fewer Tears".

Although there seemed to be a reasonable agreement on the language at this time, there were clearly grave misgivings about the document. Clearly, it could not go upwards to TC2, but clearly also it was the document that had been commissioned at Warsaw, and so the author had fulfilled his obligation. So Van Wijngaarden was given one last chance, with the final decision to be taken in December 1968.

Before that, there was a stormy, and almost totally non-technical, meeting in North Berwick, where the possibility of a Minority Report was raised for the first time. The document available at this meeting was a considerable improvement on MR93 and more improvement was to come. But MR93 had done the harm, and much of the mud flung at it is probably still sticking, even to this day.

The final meeting at Munich prepared a covering letter to accompany the document to TC2, implying some disagreement within the Working Group, but stating that "the design has reached the stage to be submitted to the test of implementation and use by the computing community". However, in the last hour of the meeting, some who were unhappy with the final form of the Covering Letter, and with interpretations which were being put upon it, produced a minority report signed by some very worthy names. Regrettably, TC2 refused to let it be printed with the Report.

In addition to the four authors of the Report, mention should also be made of the "Brussels Brainstormers". Effectively there were two teams. One in Amsterdam creating the text, and another in Brussels taking it apart again, and this mechanism was actually very fruitful.

ALGOL 68 introduced many new features to programming languages and the orthogonality of these features does indeed provide serendipitous benefits not foreseen at the time of language design.

The first implementation of something like ALGOL 68, by the Royal Radar Establishment at Malvern, appeared early in 1970. It now became clear that there were many irksome features of the language that hindered implementation and upwards-compatible extensions, whilst not providing any user benefit. Thus a revision was called for, and the decision was to do it sooner rather than later, so that not too many implementations would be affected. It soon became apparent that, to make the language definition both watertight and readable, a major rewrite was going to be necessary.

With hindsight, the 2-level Grammar used in the Report can be described as a variant of Prolog, with many of the restrictions of Prolog removed – giving it greater power, and at the same time providing greater opportunities for misuse. In the Revision, we found newer and cleaner ways to use W-Grammars, most notably through the use of "predicates". In the original Report, a whole chapter had been devoted to what we now call "static semantics". But we found too many bugs – shocks and surprises – in this chapter to have any confidence in its correctness. We were now determined that the Report should be written in such a way that we could successfully argue the correctness of any part of it. Hence this static semantics was put into the grammar, using predicates.

But the real problem with the original Report was not the W-Grammar; it was the style of the semantics, which was pedantic, verbose and turgid. So we treated the Report as a large-scale programming project, consciously applying the principles of structured programming. The semantics of the original Report was clearly a program full of **goto**s, and it conveyed a general feeling of wall-to-wall verbosity. Our revised semantics had nice indentation, cases, local declarations, and for-loops.

We took some examples of the new style of syntax and semantics to the next Working Group meeting in Vienna, where they were greeted with horror by some members. A "Petition for the Status Quo of ALGOL 68" was presented which nearly wrecked the whole project. But by the next meeting in Dresden all the fuss had subsided, and people even complimented us on the improved clarity. The Revised Report was accepted, subject to polishing, in September 1973 at Los Angeles.

Finally, there are many lessons to be learned concerning the design of programming languages and the preparation of their defining documents, although Standard Bodies do not always encourage these practices in their guidelines.

The Making of Algol 68

C. H. A. Koster
Informatics department
University of Nijmegen
The Netherlands

Speak, memory ...

In September 1967, Aad van Wijngaarden, the director of the Mathematical Centre in Amsterdam, asked me to join the ongoing project to define a successor to Algol 60.

I had just finished 18 months in the Dutch Army, and was looking forward to the resumption of a carefree life, working on Natural Language Processing. Van Wijngaarden's eye fell on me for a number of reasons: I was steeped in the spirit and implementation of Algol, well-versed in two-level grammars and I knew everything there was to know about operating systems and I/O, having just completed a vast Fortran program on an IBM 7094 under IBSYS that made use of all its 16 tape drives.

In this way I became one of the Authors of Algol 68, and a participant in one of the formative events of our profession — the making of Algol 68. In this note I will try to give you an eyewitness account. I am not a historian, and have kept very few notes from this period, but the Informal Minutes of the Tirrenia [7] and North Berwick [8] meetings bring back many memories.

Let me describe to you four parties, the Editor, the Authors, the Committee and the Computing Community, taking part in a cosmic struggle for Truth. The period is November 1967 to December 1968.

The persistent Capitalization of certain important substantives in this text is entirely in keeping with the IFIP-style of those times.

1 The Computing Community

The design of Algol 68 was firmly rooted in the Computing Community, a contemporary term for the small but growing international community of computer professionals and scientists, whose eyes had been opened by Algol 60, and who were giving shape to the Science and practice of Informatics.

In those days, proceedings of conferences came with a record of the discussion following each presentation. It is fascinating to see the great of that period learnedly discussing, and taking standpoints that foreshadow the various paradigms that over the last decades have come to split the community.

But the most interesting discussions could be found in the Algol Bulletin, edited by Fraser Duncan. Many of the criticisms of Algol 60 contributed the aims of Algol 68, as described in chapter 0 of [MR101]. Many features found in Algol 68 were first proposed in the AB, by the original Authors of Algol 60 like Peter Naur, by new Members of IFIP's Working Group 2.1 on Algol like Tony Hoare and Niklaus Wirth, and by a large and literate group of outsiders who all felt part of the world-wide Algol movement.

Starting from the appearance of the Draft Report [MR93] as a supplement to AB 26 in February 1968, both the Authors and the Editor of the AB were deluged with mail from people who had read the document well, and gave very pertinent proposals for extensions and generalizations, as well as detailed typographical and syntactical corrections[1]. What strikes me in retrospect is that hardly any correspondent ever suggested an important simplification of the language. Some groups of correspondents went through each iteration of the Report with a fine comb, sending in regular notes as the Brussels Brainstorms, Lindsey's Lamentations, the Philips Philosophies or the Munich Meditations. Remote correspondence was followed by personal contact. Some correspondents were invited as Observers to WG2.1 meetings, and became Members of this august Committee.

2 The Authors

Professor Doctor A. van Wijngaarden, Aad for friends, the Director of the Mathematical Centre in Amsterdam, had been commissioned by WG2.1 at its Zandvoort meeting to edit the Report on a successor to Algol 60, and there was no question that he was the Boss. Short, very alert, highly dynamical, he could be the most charming person as well as the most exasperating. He introduced the formal description of the syntax by means of two-level grammars [3], which gave the Algol 68 definition much of its flavour and many of its controversial properties.

Van Wijngaarden had been joined in 1966 by Barry James Mailloux from Calgary. With his lopsided grin, crew cut and unfamiliar Canadian clothes, Barry looked like to me like a native indian. Together with his wife Isobel, he lived in an apartment right across from the Mathematical Centre. After one year in Holland, he spoke Dutch fluently[2]. He worked day and night, and was invaluable to van Wijngaarden.

In 1967 van Wijngaarden was joined by John Peck from Vancouver, a quiet and dry professor with twinkling eyes. When I was drafted as an Author, John had just returned to Canada, and I first met him at the Tirrenia WG2.1 meeting, but by then I already knew his style, which found its expression in the elegant and precise syntax of the language.

Van Wijngaarden once characterized the four authors, somewhat tongue-in-cheek, as:

[1]One reaction to MR93 came in June 1968 from the Russian logician G. Tseytin, who wrote: "I have read your report during the last three months. I did not know anything about computers before, but now I do", and proceeded to give relevant criticisms.

[2]although he never managed to recognize the difference between words like 'man' and 'maan'.

Koster:	transputter,
Peck:	syntaxer,
Mailloux:	implementer,
Van Wijngaarden:	party ideologist.

Although in reality the roles were not not so clearly separated, this list does reflect the particular interests and responsibilities.

Barry Mailloux and I shared a small room on the top floor of the Mathematical Centre, overlooking the Amstel Brewery. In the mornings we worked separately or discussed together, in the afternoons we joined van Wijngaarden in the Director's office. Curiously, although all our social conversation was in Dutch, whenever we talked about Algol 68 we turned to English.

Barry was responsible for the parseability of the syntax and the implementability of the semantics of the language. At this time he was designing in detail the run-time memory management, consisting of stack and heap holding reference and procedures with different scopes. We were aware of the existence of SIMULA 67, which gave us an existence proof, but there were at that time few implementers who had designed and implemented a system of such complexity. And Barry was interested not only in existence proofs but in efficient implementation on the hardware of the time (a modern PC is faster and has more memory than the mainframes of 1967).

He was already suffering from the illness which was to make for him the following decade into a hell of aggressive chemotherapy, and he had a terrible sense of urgency. Immediately after the acceptance of the Draft Report, the ideas he had developed about the implementation of Algol 68 were described in his Doctor's thesis [6]. A few years later, with the help of some of his students, they led to one of the few implementations of the full language Algol 68.

As the junior Author I was saddled with the definition of transput (input and output). It was an established opinion that one of the reasons for the non-success of Algol 60 outside of academic circles was the lack of well-defined transput facilities: for academics, the algorithm was all that counted, the way in which its results were transmitted to the outside world was left to the local implementation.

Both WG2.1 and the Authors were grimly decided that Algol 68 should not be scorned by Industry so easily. In order to prevent the language from going the same way as Algol 60 (see [1]), it had to surpass Fortran in its strong points: efficiency of the implementations and practicality of the transput.

When I started working, the body of the Report was there, the introductory chapters, syntax, semantics and most examples. For the transput only a rough draft existed [2], very much in the style of the "Knuthput" which had been described a few years earlier by Don Knuth for Algol 60. Barry and I realised that this approach was going to make Algol 68 neither popular nor easy to use. I started again from scratch. It was decided to first make a detailed model of the file system and devices of contemporary and imaginable computer systems, in Algol 68 itself, apart from a very few fundamentals which could not reasonably be modeled within the language.

This operational description provided also the first large-scale experience of programming in Algol 68, and gave important feedback about the pragmatic consequences of various design decisions. The work was hampered by the lack of any automated support (not even a syntax checker) and the fact that the language was in steady flux. Among other things it provided the first convincing example of a triple reference (a variable holding a mutable pointer [3]), a vindication of the orthogonal reference mechanism.

All this was slowly taking form at the same time as the syntax of Algol 68 was being finalized and fleshed out with semantics and pragmatic remarks. The semantics was formulated in a very strict form of English, formulated so densely and polished so precisely (especially by Barry Mailloux, with Roget's Thesaurus always at hand) that I had the feeling that it would have been more mercyful on the human reader to consider the sentences as formulae. In fact, the English text comes close to denotational semantics, but a suitable formalism was not yet available.

The result of all this polishing was sent out to the Computing Community as the Draft Report on the Algorithmic Language Algol 68 in February 1968.

3 The Fun we had

The writing of the Report was not only Work, it was also Fun, as should be apparent to all readers. It was fun to find appropriate quotations to illustrate the text, taken from authors like Lewis Carrol; the unavoidable William Shakespeare; van Wijngaarden's favourite, the Danish poet and inventor Piet Hein; A.A. Milne, who wrote some of the world's most memorable lines about Input/Output and user interfaces; or the frivolous W.S. Gilbert.

{*Merely corroborative detail, intended to give artistic verisimilitude to an otherwise bald and unconvincing narrative.*
Mikado, W.S. Gilbert.}

The semi-final version [MR95] even contained a quotation from the final Report, which was to follow it (R2.3.c)!

The strict and sober syntax permits itself small puns, as well as a liberal use of portmanteau words. Transput is input or output. 'Stowed' is the word for structured or rowed. Hipping is the coercion for the hop, skip and jump. `MOID` is `MODE` or `void`. All metanotions ending on `ETY` have an empty production.

{*Well, 'slithy' means 'lithe and slimy'. ...*
You see it's like a portmanteau – there are two meanings packed up into one word.
Through the looking-glass, Lewis Carrol.}

[3]See 10.5.1.2.b of [MR101]

Just reading aloud certain lines of the syntax, slightly raising the voice for capitalized words, conveys a feeling of heroic and pagan fun.
8.6.1.1. Syntax

```
a) REFETY ROWSETY ROWWSETY NONROW slice{860a} :
     weak REFETY ROWS ROWWSETY NONROW primary{81d}, sub symbol{31e},
       ROWS leaving ROWSETY indexer{b,c,d,e}, bus symbol{31e}.
```

Such lines can not be read or written with a straight face.

This sense of fun carries over into the translations of the Report in other languages, like the Russian version [11], which came with a whole fresh battery of quotations. Great ingenuity was spent on the portmanteau words, e.g. in the German version by Immo Kerner [15]. It used `MODUS` for `MODE` and translated `MOID` by `MÖSCH`:

```
   MÖSCH ::  MODUS; lösch.
```

leading up to `MOID vacuum` being translated as a `MÖSCH lücke`. Now it so happened that Mösch was (and is) the name of a West-Berlin construction firm, or rather a destruction firm renowned for the large holes it has made in the city of West Berlin. Not a bad joke for Eastern Germans, who were supposed to be unaware of this capitalistic hellhole!

On a more learned level, there was the problem of the unspeakable field names. In the description of the transput, structures are introduced with a number of public and secret fields. But even a secret field had to have a name. In order to prevent the programmer from accidentally discovering a secret name, in MR93 such names were made unwriteable by the artifact of starting them with an infinite number of f's. An infinite sequence of f's contains $\aleph_0$ f's, it is all f's; so we called this sequence `ALEPH`. Since the letter Aleph was not in our type fonts, we denoted it by the percent sign %. In the final version of the Report, this was turned into an unwriteable letter, because the Committee didn't appreciate the joke.

4 The Committee

My first direct contact with IFIP Working Group 2.1, the Committee, was the meeting in Tirrenia, near Pisa, in June 1968.

At the start, the Authors were asked to summarize the reactions to MR93 that had been received, and van Wijngaarden took this opportunity to describe the changes that had been made to the language since the previous meeting (saying "The changes are based on the reactions"). The Committee was much less positive than I expected. Peter Landin deplored that the name "Algol" appeared on the document ("some of my colleagues switched from Algol to Fortran"). Gerhard Seegmüller and Brian Randell took exception to the name "Algol 68" — as if the language had already been accepted by the Committee. Van Wijngaarden was adamant and unrepentant: "I have been asked by the IFIP council to give at the IFIP Congress 68 a talk on "Algol 68"". Salt in the wounds!

Van Wijngaarden pressed for a decision by WG2.1 at the next meeting, in North Berwick. "Here is the document on Algol 68. Stamp it under the provision that after two years, in which it is planned to implement, teach and make primers, the language is subject to possible revision as result of this effort. Big companies wouldn't implement without this stamp."

Gerhard Seegmüller then formulated four conditions for acceptance of the Report:

1. Clarification and expansion of the syntax
2. Formalization of the semantics in competitive descriptions
3. Implementation of the language
4. an Informal Introduction.

Apart from the Authors, the Committee agreed with these conditions. Van Wijngaarden, dramatically: "Does anybody know of a language of world-wide scope (such as Algol 60, Fortran, PL/1), that has been published after a compiler"? Landin: "Lisp is an example".

The discussion returned to technical matters. The coercion mechanism, at that time still using some explicit coercion operators, was criticized as unclear, not fundamental enough and superfluous due to the overloading of operators. The transput proposal was discussed, briefly but quite constructively, I thought. A special evening session was held with the small number of members really interested in the subject: Merner, Seegmüller, Goos and myself. The main outcome was that formats had to be made more general, including dynamic replicators.

The dynamic checking of array bounds and scopes of references was discussed. It is striking to note the great concern for micro efficiency, which has in many respects hampered the development of Algol 68. At this point the desire to avoid bound checks still led to awkward syntax, like `proc ([1:int n] real vec):`, an integer constant-declaration within a formal bound to obtain the size of the array `vec`. Similarly, a discussion of dynamic scope checks (rightly considered unavoidable in some cases), followed by a discussion on procedures delivering procedures (in some cases limited by scope problems) did not lead to the obvious conclusion to do away totally with the (statically unenforceable) scope restrictions. Algol 68, which has higher-order functions, narrowly missed having Currying, which would have made it possess a complete functional sublanguage, even though Gerhard Goos saw no problem in implementing it. In fact, the drastic and simple proposal to give every object an infinite scope was made by Hans Bekic at a later meeting (where, I do not recall), but this beautifully simple and effective proposal was not accepted for reasons of efficiency.

Another chance at more generality that was missed was the extension of operator overloading to procedures, mentioned briefly by Peter Landin. Unfortunately, the discussion veered off from this subject.

At various points in the discussion, a sore point in the description came up: the question of infinite productions (such as ALEPH), infinite modes (caused by recursive type declarations) and an infinite number of Context-Free production rules. Van Wijngaarden, who was a purely constructive mathematician, surprised me by his flippancy on the subject. When Nobuo Yoneda and Peter Landin criticized him, he responded: "This problem has puzzled us (not me). My machine may execute steps in geometrically decreasing time intervals". Of course this would also allow "his machine" to prove or disprove Fermat's Theorem in finite time, so this caused general laughter[4].

Nobuo Yoneda deplored that the unions in MR93 were not commutative and not cumulative, so that union (int, real) was not equivalent to union (real, int) and to union (int, union (int, real)). The Committee decided that unions should be made both commutative and accumulative. Van Wijngaarden protested that this was damned difficult, it would cause a terrible amount of work. Amid general catcalls that his description method was to blame, he promised a revised syntax. "We have only one life. Of course, if one of us gets ill ... – you are drawing such strong time limits on us! Give me time till after lunch." That night, he and John Peck started scribbling, and the next morning he showed us one page of syntax which solved the problem, a nice little nondeterministic automaton [5].

Then came the last phase of the meeting: what would happen next. The majority of the Committee seemed to want to thank the Authors politely for their trouble, and invite others to make alternative definitions. Against this mood van Wijngaarden fought valiantly, pressing for a decision to be taken in the next meeting. At one point, he told the Committee: "This Working Group has worn out its first editor, Peter Naur. Then it has worn out two authors, Wirth and Hoare. If I understand right, it has now worn out four authors."

Against tremendous opposition, using every rhetorical device [6], he managed to commit WG2.1 to a definite resolution:

> The authors are invited to undertake to edit a document based on MR93, taking into account the questions and remarks received before, on, and possibly after, this meeting to the best of their power in the time they can afford for this job. This document will be submitted to the members of WG2.1 before 1 October, 68. This document will be considered by WG2.1. Either WG2.1 accepts this document, i.e. submits it to TC 2 as Report on Algol 68, or it rejects it.

Even though a large part of the meeting had been very constructive, it ended on a sour note. The behaviour of the great scientists present showed me that the progress of science is not just a matter of objective truths but also strongly influenced by human

[4]But I know that privately he was worried, and only articles by Lambert Meertens and myself in AB 29, showing how to deal finitely with recursive modes, put his mind at rest. In the Revised Report, this matter was resolved very elegantly.

[5]See 7.1.1.aa-jj and 8.2.4.a-d of [MR101]

[6][7], 8.23:
SEEGMÜLLER: Then I have to vote against it!
VAN WIJNGAARDEN: I never say something final.

emotions. I concluded, still naively, that only a very good language defined in a very clear report could convince the members of WG2.1.

5 Mending the fences

We had barely 7 weeks to make a new version of the Report, for the Committee to vote on in North Berwick. In this short time we completely revised the coercion mechanism (so that all coercions were now implicit), as well as the syntax and semantics of arrays. Unions were made commutative and absorbing. Formatted transput was made much more flexible, taking up ever more pages in the Report (and code in eventual implementations).

In order to help the reader of the Report, all syntax rules were adorned with compact but helpful crossreferences. A vast number of small examples and explanations were spread as pragmatic remarks all over the text, which therefore grew appreciably in size.

All in all, I felt quite satisfied with our work, as I was driving over the appropriately named A68 to meet WG2.1 in North Berwick, with 50 copies of the revised Draft Report [MR95] in the back of my deux cheveaux. The Authors had done what they had been instructed to do.

6 The North-Berwick meeting

The meeting started off badly. Since only 15 of the Working Group members (out of 34) had been present at Tirrenia, there had been no quorum. The validity of the Resolutions taken, and therefore even the legality of the present meeting, was in doubt. After much debate they were re-voted, and accepted. I wondered whether an illegal meeting could legalize itself by re-voting history.

Now and then, all parties took time off to blame the Chairman, Willem Louis van der Poel, for procedural errors or well-meaning remarks that managed to throw oil on the troubled flames. In fact, shouting at the Chairman seemed to be the only activity in which the WG2.1 members found themselves united.

A few hours of desultory technical discussion led to the main issue: a discussion of future work. Until then, the future had been clear: after finishing Algol X (apparently with X = 68), the Committee would turn to the study of Algol Y, the language which was to include self-extension. Edsger Dijkstra now proposed a complicated experiment in soulsearching, which led to a heated debate: who would like to do what, and with whom? A small majority considered the finalization and maintenance of Algol 68 the most important, a large minority (including Hoare, Dijkstra and Turski) had wider plans. Actually what was happening was the birth of WG 2.3 [7].The overwhelming interest was in "primitives", i.e. elements of semantics.

[7]DIJKSTRA: "The group to which I would be most attracted would be less decided by the subject of the work and more by the attitudes of other members in such a party"

Tony Hoare suggested the production of a brief document by each of the Members on each of his favourite subjects. Van Wijngaarden: "I like to thank Hoare for the distinction between the members who supply us with documents and those who do not." This remark did nothing to clear the atmosphere.

Heinz Zemanek gave a stirring address, describing what TC2 expected from WG2.1: "You have to admit either that the document you have in your hands is the new Algol or that the editors have failed. In the second case you may charge the editors with further work or abandon the project. In the latter case you may decide that the contents is O.K. but the description has to be changed. You may also select new authors or issue the document as a preliminary one. You have, however, to make some decision, you cannot escape your responsibility, you cannot get rid of the problem."

It was at this point that Doug Ross expressed his desire for a Minority Report, to be part of the Report. Immediately, people started discussing the modalities and timescale for preparation of such a report, rather than its desireability. Van Wijngaarden protested that at all earlier occasions, the Algol Working Committee had done without a minority report, although there was no one who agreed in every respect with the documents. But the ominous M-word was there to stay.

Discussion went on to technical subjects, array bounds and efficiency, the implementer's burden and the Bauer Principle [8]. The proposal to enforce definition before application (which later led to a number of Algol 68 subsets) was rejected because it would eliminate recursive modes. The same fate befell a suggestion to consider operators as macros rather than procedures. Micro efficiency, again. [9]

Back we were, on the question of decision to be taken in December, the publication of the Report and the Minority Report and the use of the name Algol 68 in courses and seminars. Most members seemed to like the language described, but this was not the case for the description. Hoare and van Wijngaarden both mentioned the possibility that the Report be published (first) under the names of the Authors, without WG2.1 responsibility, but it was preferred to accompany the Report with an eventual Minority Report, to be drawn up by the end of the next meeting.

Van Wijngaarden then brought up the fact that Peter Naur had published in AB28 a paper very critical of WG2.1 and Algol 68, sparked off by MR93. "As a Council member I will have to bring up the subject of AB28. It contains a piece of mud. IFIP pays the money for AB. I would suggest to the Council to reconsider the money appropriated for editing AB." He was all the more offended, because it was reproduced and distributed by the Mathematical Centre. Why he chose to put this matter before WG2.1, I do not know. His remarks were of course very unfair to Frazer Duncan, the Editor of the AB, and led to a heated discussion, covering most of the morning, regarding censorship,

[8]VAN WIJNGAARDEN: "Who does not want to use complex facilities, does not pay for them. If the user wants to use them, he has to pay a little."

[9]VAN WIJNGAARDEN: "... If we accept this point of Mr Lindsey we will produce a FORTRAN-like language, by which I mean its intellectual level."
RANDELL: "It is well to remember that there are also good sides in the FORTRAN intellectual level – do not forget its efficiency."

refereeing of articles and duties towards the Computing Community.

Van Wijngaarden could make himself either greatly liked or immensely impopular. Barry Mailloux had to say "Chucks, fella's" many times in his most reconciliatory tone before order was restored. Later, in the absence of van Wijngaarden, Barry felt it necessary to apologize for him. "He did not refuse to reproduce AB, he wanted to express his personal dislike, but nothing more. The second thing is that his statement "I am offended" is a technical and not a personal remark. On my own behalf, I would like to say the following: it was suggested that I and some other people seek fame, credit and fortune. I would like to deny it. I did a lot of work because I do believe in the language and believe strongly enough to wish to propagate it."

Thus ended the meeting, which had brought little technical progress but which had prepared the stage for the drama to be enacted in Munich, in December 1968.

7 The IFIP Congress

The IFIP 1968 Congress took place that August in Edinburgh, just a few hours drive away from North Berwick. Van Wijngaarden's invited lecture on Algol 68 was to me the high point of the conference, and not only to me. The auditorium was packed, people were standing on all sides, even in the corridors and outside, in front of the hall. Van Wijngaarden appeared in the centre, smiling radiantly. "Let me sell you a language", he started, and proceeded to outline the ideas behind the language. He showed some examples. "Can you define triangular arrays?" someone (Tony Hoare?) interrupted. "Not just triangular, but even elliptical" replied Aad, and showed how. He carried the listeners with him, from scepsis to enthusiasm. There was a prolonged applause.

Vehemently discussing, people streamed out of the hall. A small man pushed through the throng, straight at me. "Conkratulations, your Master hass done it" said Niklaus Wirth in his inimitable Swiss-German English.

8 Towards Munich

At Munich, it was then, that the ultimate choice would be made. Again van Wijngaarden, Mailloux and I went over the whole text, making the last changes in the language and its description, cleaning up various dark formulations, correcting small errors and in the process retyping everything. Van Wijngaarden loved the freedom in typefonts offered by the new IBM "golfball" printer, and introduced outlandish symbols for various operators with boldface names (e.g. `ELEM` and `UPB`). He found a use for every symbol on the APL-typeball. He would have loved TEX and the possibilities to define new typefonts!

There existed no wordprocessing software to speak of, and we had not even the support of an editor to mechanize the production of the Report. How many times have I glued small strips of white paper over Snopake-encrusted originals? By now even sometimes the wording of sentences was influenced by the fact that they had to fit within a given space. The nearer the deadline, the more frantic the work became. We were joined by Lambert Meertens, but still things went too slowly. The text kept changing, always for good reasons, and there was no chance to leave the normal period for an orderly offset production. The printer taught us to make matrices for the offset machine. In the end we had to learn how to bind and glue the whole document. Van Wijngaarden took time off to design and produce a suitable cover. It showed a pattern built out of hundreds of elem-signs — and one little commercial at-sign, his personal mark.

On the morning of the last day, after a frantic night and just before our flight left, the work was finished: we had produced the first printing of the Final Draft[10]. No time to catch up on sleep. Lambert and I found ourselves sitting in front of the plane, dog tired. What would the Committee decide? We were too tired even to speculate. The stewardess brought us, unbidden, two baby bottles of champagne each, from a Gentleman in the back. We looked over the back of our seats: Van Wijngaarden sat there, besides a sleeping Barry Mailloux, prim as a daisy, and waved his hand at us. The Authors had done their job.

9 And after

Algol 68 was accepted by WG2.1 as its own child at the Munich meeting in December 1968, but it was a Pyrrhus victory for van Wijngaarden: a large minority dissented, and wrote a minority report.

Translations of the Report in many languages appeared [11, 13, 14, 15], as well as an Informal Introduction [12] and textbooks explaining the two-level formalism [16]. Implementations were slow in coming, apart from some (limited but successful) subset implementations. Before implementations of the full language became available, the state-of-the-art in compiler making had to be advanced quite a lot. The language was used in many courses. Its effect, through teaching, on the minds of a generation of computer scientists was much greater than its utility in practical applications.

The announced Revision of Algol 68 started almost immediately and took until 1974 [RR]. It resulted in an exemplarily clear, precise and consistent description of an elegant and orthogonal language that was at that time already classical but dead — the Ilias and Odyssee of Computer Science.

What has gone wrong? Many convincing reasons can be found. The lack of timely implementations (but those people who have actually programmed in Algol 68 remember it as a beautiful means of expression); the obscurity of the description (which is denied by virtually anyone who has bothered to study it); the lack of political and industrial backing (the fate of Algol 60, all over again). I think that Algol 68 was the result and the conclusion of a decade of search for the ideal Algorithmic Language, but that the time for a unique programming language was already over when it appeared. In the seventies, research went on to other problems: Software Engineering, System

Implementation Languages, Databases and Computer Graphics.

Algol 68 lives on, not only in the minds of people formed by it but also in very unlikely places, like C and C++, whose concepts and terminology at numerous places give a weird echo of Algol 68, even though the orthogonality in the syntax, elegance and security have been mostly lost. A whole new generation of programmers uses coercions and casts. In fact, the boisterous discussions in the programming community about the shortcomings of C++ and solutions to overcome them gives me a strong feeling of déjà vu, reminding me of the making of Algol 68. The Far West of Computer Science.

In 1974, during an IFIP WG2.4 Meeting in Berlin, I was stopped in the corridor by Jean Ichbiah, the author of the language LIS and designer of what was to become ADA, on his way to the success that brought him the Legion d'Honneur. He said to me with great emphasis: "We are going to do right what Algol 68 has done wrong". Have they really, I wonder?

References

[1] R.W. Bemer, *A politico-social history of ALGOL*. In: *Annual Review in Automatic Programming* 5, 1969.

[2] J.V. Garwick, J.M. Merner, P.Z. Ingerman and M. Paul, *Report of the Algol - X - I - O Subcommittee*, WG2.1 Working Paper, July 1966.

[3] A. van Wijngaarden, *Orthogonal Design and Description of a Formal Language*, Mathematisch Centrum, MR 76, October 1965.

[4] A. van Wijngaarden and B.J. Mailloux, *A Draft Proposal for the Algorithmic Language Algol X*, WG 2.1 Working Paper, October 1966.

[5] A. van Wijngaarden, B.J. Mailloux and J.E.L. Peck, *A Draft Proposal for the Algorithmic Language Algol 67*, Mathematisch Centrum, MR 92, November 1967.

[MR93] A. van Wijngaarden (Editor), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, *Draft Report on the Algorithmic Language Algol 68*, Mathematisch Centrum, MR 93, January 1968.

[6] B.J. Mailloux, *On the implementation of Algol 68*, Ph.D. thesis, University of Amsterdam, june 1968.

[7] Informal Minutes of the IFIP WG2.1 Meeting in Tirrenia, June 3 - 7, 1968.

[MR95] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, *Working Document on the Algorithmic Language Algol 68*, Mathematisch Centrum, MR 95, July 1968.

[8] Informal Minutes of the IFIP WG2.1 Meeting in North Berwick, July 28 - August 1, 1968.

[9] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, *Penultimate Draft Report on the Algorithmic Language Algol 68*, Mathematisch Centrum, MR 99, October 1968.

[10] A. van Wijngaarden (Editor), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, *Final Draft Report on the Algorithmic Language Algol 68*, Mathematisch Centrum, MR 100, December 1968.

[MR101] A. van Wijngaarden (Editor), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, *Report on the Algorithmic Language Algol 68*, Mathematisch Centrum, MR 101, February 1969; also Numerische Mathematik, Vol 14, pp 79-218, 1969.

[11] A.A. Bährs, A.P. Ershov, L.L. Zmieskaya and A.F. Rar, *Soobshchenije ob algorimicheskom jazyke Algol 68*, Kybernetika, Kiev, Vol. 6, 1969 and Vol. 1, 1970.

[12] C.H. Lindsey and S.G. van der Meulen, *Informal introduction to Algol 68*, North Holland Publishing Company, Amsterdam 1971.

[13] D. Toshkov and St. Buchvarov, *Algoritmichniyat yezik Algol 68*, Nauka i Yzkustvo, Sofia 1971.

[14] J. Buffet, P. Arnal and A. Quere, *Definition du Language Algorithmique Algol 68*, Hermann, Paris, 1972.

[15] I.O Kerner, *Bericht über die Algorithmische Sprache Algol 68*, Akademie-Verlag, Berlin 1972.

[RR] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens and R.G. Fisker, *Revised Report on the Algorithmic Language Algol 68*, Acta Informatica, Vol. 5 Tome 1-3, Springer-Verlag, Berlin, 1975.

[16] J. C. Cleaveland and R.C. Uzgalis, *Grammars for Programming Languages*. Elsevier/North Holland, 1977.

An Orthogonal first Programming Language

Sietse van der Meulen
RU Utrecht

Algol68 in Dutch academic curricula

'Informatica' ('Computer Science') was accepted as a principal study in the Dutch academic statute as late as 1981. By that time ALGOL68 was already very much a dying language. After a long pregnancy of about 8 years – including the six years of breeding in the pouch of WG2.1 – the actual birth announcement came in the REVISED REPORT ON THE ALGORITHMIC LANGUAGE ALGOL68 in 1974. Though stages of life in programming languages may be quite different from those in its inventors, AD 1981 the dying hero was not even in its puberty. In that light it can hardly be a surprise that we do not find many trails in our present academic curricula. ALGOL68 still shows up here and there as a subject in courses on the structure of programming languages and its defining formalisms, as also in courses on implementation. Nowhere, however, it still is in practical use. There are at least two simple reasons: usable compilers are not longer available on the computer sites in our universities, and there have never been usable compilers on the wide variety of small computers and students' PC's.

Remarkably enough, the rôle of the language has been more important in academic curricula in the seventies – the pre-Informatica era in The Netherlands, the embryo-/pouch- and baby-years of ALGOL68. In these years 'Informatica' existed 'hidingly' at some Dutch universities as a more or less important subsidiary subject in the main streams of mathematics, science or engineering.

In that context ALGOL68 could be 'tried out' in lectures on the language itself, on its defining mechanism and on its implementation. We were in the middle of the heavy fought battles on the issues of programming languages: the debate on the pro's and con's of FORTRAN, ALGOL60, LISP, COBOL, PL/I, APL, SNOBOL4, SIMULA67, PASCAL, ... , and what more did we have until and already including C, MODULA and ADA. In this debate ALGOL68 for many of us was the most interesting point of departure for stopping the crazy proliferation in a programmers Babylon. In almost every comparison it seemed to be by far the best of all.

It was also our strong belief that ALGOL68 in particular could and should be used as the first programming language – the language in which important concepts of programming and datastructures could best be taught on the basis of its orthogonal structure. We were convinced that the language was easy to teach, to use and to understand. There existed already textbooks, companions, practical guides and informal introductions, and we had good hope that precisely this language could be the native language of the generation to come in the new study of 'Informatica'. We can thus roughly distinguish two periods: **1966-1981** and **1981-now** – the raise and fall of ALGOL68 in The Netherlands.

1966-1981

While a subgroup of WG2.1 was still elaborating the revision of ALGOL68 and more so its defining mechanism, courses already began at the Mathematical Centre in Amsterdam, the Technical University of Delft and the University of Utrecht. At the Mathematical Centre the first lectures on the new language, still under the name ALGOL-X, came as early as 1966. The lecturer, of course, was Aard van Wijngaarden.

Here I want to make a personal note. I attended one of these lectures and I was impressed by the metaphors van Wijngaarden used on that occasion:

How external objects as generated by a two-level grammar could – by 'teasing' them in a proper way – be brought to yield internal objects in a computer memory and how – by 'teasing' these in their turn decently – the internal objects could be brought to yield other internal objects, until the ultimate 'plain values' that could become external objects again, by putting them out. I liked this description of what happened at interpretation, or through compilation and execution. In a brief conversation after the lecture he said to me:

> Listen, things are not that simple. However, when you tease unprepared students with all complications at once, they will never yield an internal object of any value. Of course, later you will have to tell them, in the proper accurate way, that many details lead to certain complications. But these should disappear as soon as they are fully understood.

The above is a rather precise translation of what he said to me in Dutch – and what he said is quite remarkable from his mouth. Van Wijngaarden never showed much interest in popularizing his brain-child. *I lack any talent in that direction* he once said. Moreover, he was much more involved in the defining language than in the language defined.

After that conversation I studied some of the many 'draft's of the premature and too early issued first 'final' Report – and became more and more frustrated and alarmed by its extreme complexity. Fortunately, I also read at an early stage Charles Lindsey's amusing little pamphlet "ALGOL68 without tears". I became convinced that the language was as simple as suggested by van Wijngaardens' metaphors, but that its defining document would be an absolute disaster, in particular for the 'unprepared students' – and thereby for the language itself.

I communicated my frustrated feelings and despair to Aard and he, kindly and maliciously (I always found him kind and malicious) challenged me to write an 'informal introduction'. I did not dare to take the challenge before I met Charles in 1967 in North Berwick. At a later meeting the two of us were commissioned to prepare an "Informal Introduction to ALGOL68" as a 'companion volume' to the Report.

We have reason to believe that the 'companion volumes' (to the Report and later to the Revised Report) at least helped to postpone the final break down. John Peck also wrote a 'companion volume'. Another early bird was Frank Pagans' "A Practical Guide to ALGOL68". They all eased to some extend the fear, trembling and horror in the computing community.

In Delft Wim van der Poel and his group gave an early start to the promulgation of the language in The Netherlands, and became active on the front of implementation already in 1969.

In Utrecht, from 1970 to 1975, I tried our Informal Introduction out on several groups of students from quite different disciplines, who were interested in programming and programming languages as such. Actually using the language was not yet possible because we did not have a compiler at our disposal before 1976.

In Berlin at the Technical University Cees Koster started introductory courses in 1972/1973 and 1973/1974. He invited me to give the first of these courses which led to a two-volume textbook on the language in German which I wrote together with Peter Kühling. This Berlin-experience was very stimulating for my activities in Utrecht where 'Informatica' was hiding somewhere deep in the Department of Mathematics. In 1977 Cees took everything, including a few colleagues, with him to Nijmegen.

In Nijmegen ALGOL68, as implemented on the FLACC-compiler on IBM, became the first programming language in the 1978 curriculum and remained so until 1987. Cees wrote several textbooks on structured programming using ALGOL68 and later ELAN – an in Germany developed educational dialect.

A very important event for the position of ALGOL68 in Utrecht, as also in Manchester and Amsterdam, was the completion in 1976 of an almost perfect full implementation on the CD68xxx (what is in a number) later renamed as 'Cyber'. Control Data was **coerced by contract** to deliver a full implementation of the language together with its (for that time huge) numbercruncher at the computer centres of Utrecht and the two universities of Amsterdam. Joost Schlichting and his group in Rijswijk (Wim van der Poel was one of the consultants) did a quick, but also incredibly clean, job and produced an excellent compiler – by taking the Report **literally**, as was the mandate. Everybody – in favour or against the language – has to accept that achievement as a quite convincing proof of the quality of the Revised Report!

At last then we in Utrecht could use ALGOL68. We immediately started to define and test a superset for operations on vectors and matrices over arbitrary fields and of variable size named TORRIX. It has been used for some time at the Numerical Applications Group (NAG) in the United Kingdom.

From 1976 until 1982 we taught ALGOL68 in Utrecht as a first programming language. In the first instance we confined ourselves to an 'educational subset' (nickname SPEEDY, sometimes even SPEEDY68), though always emphasizing the full orthogonal structure and encouraging students to experiment with it.

Occasionally we even mixed the subset taught, with superset features like TORRIX. The main blessing of an orthogonal language is precisely that: you can leave out everything you don't like, and orthogonally add everything you would like to have – with all its orthogonal consequences.

1981-now

Alas, we can be very brief on this period.

At the start of 'Informatica' in the academic statute in 1981, the Dutch departments were confronted with a considerable inflow of young students coming straight from secondary schools, and eager to make a good career in the economically promising brandnew field of technology. Most of them had none or very little experience with computers and programming. We were just in the transition of the large mainframes to minicomputers like the PDP8, the PDP10 and later the VAX. The micro's and PC's were to come a few years later. UNIX was rising above the European horizon. In that situation we had to take decisions on the first programming language for 'Informatica'.

The decision in Nijmegen was to proceed with ALGOL68. The Technical University of Twente (Enschede) followed, using Koster's books on structured programming. The two universities of Amsterdam hesitated quite some time, and let the time go by, although – in the wake of the Mathematical Centre – both sympathized with the language for quite some time.

In Utrecht we started one year later with 'Informatica' as a principal study and had to make our choice in 1982. We would get a VAX and UNIX, and almost consequently had to drop ALGOL68 in favour of PASCAL. To my personal chagrin I had to turn my back to ALGOL68, although we still could use the language on the Cyber until about two years ago – but in the unacceptable batch mode of two decades ago.

At the Technical University of Twente the hesitation lasted five years. The academic year there was subdivided in trimesters. Believe it or not: in the first trimester ALGOL68 was taught, in the second it was PASCAL, and in the third trimester the students could freely choose between the two – for a practical exercise. The main troublespot was the concept of a pointer: in the first trimester the ALGOL68 **ref ref** was just a normal incidental stone in an orthogonal building – in the second trimester the PASCAL **pointer** was a tricky new concept with many pitfalls – in the third trimester most students remembered precicely these and consequently used PASCAL for the practical exercise. Twente dropped ALGOL68 in 1985.

In Nijmegen ALGOL68 held until 1987. In that year the language died in The Netherlands.

Algol 68 — 25 Years in the USSR

Mikhail A. Bulyonkov (*Novosibirsk*)
Alexandr F. Rar (*Novosibirsk*)
Andrei N. Terekhov (*St.Petersburg*)

The first information on Algol — then 67 — was brought to our country by Dr. Ershov in 1966. There arised then in Russia three focuses, in which current work of van Wijngaarden's team was attentively observed. These were Dr. Ershov's group in Novosibirsk, Dr. Lavrov in Podlipki, Moscow Region, Dr. Levinson in Moscow. In these places people began to send numerous remarks on the language and to consider its Russian terminology. The process of translating the report on Algol 68 into Russian followed the process of its up-to-dating in English so closely that both the Report and its Russian translation [1] were issued in the same year of 1969.

Even before that, in February 1968 first lectures on the new language were held in Bakuriani, Georgia, at the special Winter School on Algol 68.

Later on, the expansion of the language in the USSR was noticeable, but restricted. Groups of ardent adherents of Algol 68 appeared in Kiev, Kharkov (Ukraine), Izhevsk, Kazan, Tomsk, Berdsk and other Russian cities. The most strong and most fruitful of these groups was that of the Leningrad State University. For many years the Algol 68 was the principal programming language studied in the course of Computer Science in the Leningrad University. The Leningrad implementations of Algol 68 will be discussed later. But all these Algol islands were rare in the ocean of Fortran and, later, in the seas of Pascal, Modula, Ada.

Starting from 1976 the official national organization on Algol 68 began to act, under whose observation were the translation of the Revised Report, publishing of literature, testing and adopting of compilers. Up to 1982 this body was called "The Scientific-Technical Commission" and was headed by A.P.Ershov. Lately it became a "Working Group" with G.S.Tseytin as its head. The last session of the Group took place in 1988, in which the national Standard of Algol 68 has been adopted. No attempt was made since then to call a new session of the Group.

Our first publication of the Report, mentioned before, was bilingual. It contained the whole text of the Report with all pragmatic remarks and pictures from Winnie-the-Pooh. The work on the Russian translation and especially the sophisticated job of transforming the syntactic and metasyntactic rules into the Russian form gave rise to considerations of how to formalize the rules for creating national variants of Algol 68. Methods of designing syntactical and metasyntactical charts of Algol 68 were also considered, and properly designed charts accompanied the publication. Both these problems became topics of the report presented at the Working conference "Algol 68 Implementation" which IFIP held in Munich in July 1970 [2]. The considerations on constructing national variants have been taken into account by the authors of the Revised Report and corresponding proposals have been partly included into that Report.

The translation of the Revised Report was not performed as quickly as in the case of the previous Report. It was being produced steadily and thoroughly, by a single author rather than by four of them, but under the vigilant surveillance of the national Algol 68 Commission, and the final result had been published only in 1980 [3].

There were not many publications on Algol 68 afterwards in our country. We can mention the translations of the "Informal Introduction" by C. H.Lindsey and S.G. van der Meulen [4] and of "A Practical Guide" by F. G.Pagan [5]. As for the original books on the language, there can be mentioned short descriptions by A.N.Maslov [6], by V.A.Vasilyev [7] and also the "Introduction to Algol 68" by A. N.Terekhov. The latter forms a part of the book describing the Leningrad compiler [8].

As it has been said before, the national Standard of Algol 68 has been adopted in 1988. The text of this Standard was that of the Russian translation of the Revised Report with some minor variations in form but not in substance. At the same time another national Standard has been adopted, namely "The Standard of the Extended Algol 68". This document made use of the IFIP proposals on modules and separate compilation and also of G.S.Tseytin's proposals on exception handling.

The complexity of Algol 68, intrinsic to it, has contibuted to the fact that the language was nested primarily in the academic and university environment and did not find a large support in the industry. There was a danger that Algol 68 could have become an object of purely mathematical investigations with all closeness proper to them. The members of the Working Group did realize clearly this problem and paid considerable attention to practical implementation of Algol 68 as the main way of its propagation. No session of the Group took place where problems of the language implementation were ommitted.

There were several attempts to implement Algol 68 in the USSR. For some reasons, which will be shortly mentioned below, only one of them, namely that of Leningrad, survived, but it really became widely spreaded.

One of the first implementations of Algol 68 was done in a Kiev computer-producing factory in the end of seventies for Siemens computers. Its authors are S.I.Shtitelman, M.G.Shteinbukh, L.A.Makogon. The implementation was oriented to an information management system called "START", for which Algol 68 was the only language it used. The authors of the project were interested in Algol 68 primarily as in a source of a data base language. The Kiev implementation anticipated many features of modern languages of that sort: persistent objects, an elaborated system of types, orthogonal design, a large share of interpretativity, and so on. The system of types and orthogonality were in fact due to the Algol 68 itself but the persistence feature urged for some corrections of the language. Namely an "everlasting block" has been introduced, meant to preserve between the executions of the program and could be used by different programs. In fact it was a data base. Some other variations were also done without any regards to the standardization efforts for Algol 68: all arrays were considered to be flexible, a control variable of a loop was **long int** rather than **int**, complex values were absent and so on. Accepted by the Working Group in 1979 this system exists no more because of changement of hardware in the factory.

At the same time Algol 68 was being implemented on the base of the DEC architecture. This implementation was being performed under guidance of Dr. M.Levinson. Though uncompleted it brought some original ideas into the implemantation techniques. The main distinction of this implementation was in the scope checking: the life-time of any object was not restricted by the execution time of the block, in which the object was declared.

An inplementation of Algol 68 compiler for "Elbrus" computer complex had been designed by V.V.Brol, V.M.Gushchin, V.B.Yakovlev (Moscow). Its source language is the full Algol 68 defined by the Revised Report and extended by some facilities of module handling. The compiler provides a good quality of the object code, a rather complete error diagnostics. It makes a good use of similarity between main concepts of Algol 68 and those of "Elbrus" architecture and operating system. The compiler has been accepted by the national Working Group in 1985. The Leningrad group participated actively in testing of that compiler. About ten large application packages developed in Leningrad University were ported almost without a problem to the "Elbrus". But the tragedy of the "Elbrus" compiler was that these were practically the only real-life programs that were processed by it.

The promising "BETA project" in Novosibirsk primarily designed by Dr. A.P.Ershov, M.Shvartsman, A.A.Baehrs was intended to produce compilers from language descriptions almost authomatically, and it had Algol 68, PL/I and Simula 67 as its first objectives [9]. The system has really been created, but not in the form initially thought of, and the languages it now encompasses areœSimula 67, Pascal (these languages being implemented by G.G.Stepanov and S.B.Pokrovsky), Modula 2 (L.A. Zakharov), a subset of Ada (S.V.Ten), but not Algol 68 or PL/I [10]. Nevertheless, the concepts of Algol 68 were used in the BETA system for creating both its universal compiling scheme and its internal language.

Mostly developed were the works on Algol 68 implementation in the Leningrad University, in the group headed by Dr. G.S. Tseytin and Dr. A.N.Terekhov. Primarily these works were coordinated with the work on BETA system in the frame of a strategy elaborated by the Working Group. It was supposed that the Leningrad group would construct a debugging compiler, a sort of an avant-garde, meant to win new application areas for main forces, namely basic compiler to be constructed in Novosibirsk. A natural presumption was that any application program, debugged by the Leningrad compiler should run on the Novosibirsk compiler without a need to be modified.

As it has been just said, the Leningrad project was originally oriented for practical uses and that fact determined both its principal decisions and its history.

The first version of the Leningrad compiler has been completed to the year 1976. Its analysing part was written in Algol 60 and run on the ODRA 1204 computer. Its generating part was written in the Macroassembler and worked on IBM mainframe.

After that the whole compiler has been rewritten in Algol 68 and extended: every procedure of the compiler (more than 1000 of them) has been translated on ODRA from Algol 68 to an intermediate language (IL), the punch tape obtained has been input to the IBM computer and translated from IL to the IBM object code. The translation of an average procedure took 10 minutes on ODRA and 20 minutes on IBM. Since debuggung of these procedures urged remodifying and recompiling them, time consumed was even greater. The result of this bootstrapping process was the residential compiler which compiled every procedure for 2–3 minutes.

In 1978 the first bootstrapping has been done and the resulting compiler has been yielded to many users in different fields (mathematical physics, radar techniques, simulation). Immediately afterwards the second bootstrapping began. At that moment the designers realized necessity of library preludes and separate compilation of procedures. Therefore a new construct has been added to Algol 68, which appeared to be rather like **nest** not yet known to the authors.

Great efforts were needed to optimize procedure calls. The code size has been reduced from 16 to 6 bytes per call. For comparison, PL/I-F compiler takes 150 commands per call and the optimizing PL/I takes 30 commands. Ten years later there arised an idea to reverse the direction of the stack, which provided a considerable reduction of code for procedure calls.

The second bootstrapping has been completed to the year 1979 and this version lasted over 10 ten years with minimal modifications. In that time the designers were busy primarily with the programming technology, for there appeared that Algol 68 is too complicated to be a language for simple tasks, but as for big tasks (real time, for instance) language facilities only do not suffice. But compilation problems were not forgotten: a dozen of cross-compilers for different specialized computers were compiled, an optimization techniques has been improved, the compiler was being integrated with other technical facilities.

At that moment it became clear that it was highly improbable that the basic compiler would ever appear. So a new pass was incorporated in the Leningrad compiler, namely an optimization pass. In contrast to that that was planned in the BETA project, the Leningrad compiler exploited only local optimizations, because an introduction of global optimization would have required a significant revision of compiler structure. Statistics showed later that even among all local optimization only two of them are the most effective — those for parameter passing and for array indexing.

In a sense, implementation of Algol 68 undermined the social basis of global optimization: in most cases the results of global optimization can be expressed in the same language.

In 1980 the Leningard group was addressed by the scientific-industrial union "Krasnaja Zarja" (which is the major telephone production company in the USSR) with proposal for cooperation in programming of a large class of control and communication task and, in particular, for design of functional software for telephone stations controlled by specialized computers. It took several years to get inside the specifics of the new field, to develop prototype implementations, and to settle organizational issues. The people from Leningrad group were convinced by their previous experience that it was absolutely necessary to use high-level programming languages. However they had to start with raising the programming culture of the applied programmers. This was caused by the reason that traditionally in the area of embedded real-time software development computers with non-standard architecture oriented to an application domain are used. (In fact, it is not evident, what this orientation should be. For example, if a specialized computer perform nicely some special operations, but works badly on branching and procedure calls which occurs thousand times as often as specialized operation, then could one consider it to be orientated to that application domain?). Non-standard architecture and small number of specialized computers lead to the absence of sufficiently developed operating systems, compilers, debuggers, and other common programming tools. So the group had to deal with punchcards and switchboards.

In a short period of time new cross-assembler and interpreter were developed, which together with documentation system and some service programs constituted the basis of the first industrial technological system based on Algol 68 and which was intensively used by hundreds of applied programmers. Naturally, the technology was quite restricted but still popular due to the following objective reasons:

- A widely accessible mainframe with wide services was used instead of specialized computers;
- Rich debugging tools of the interpreter which were not possible on a specialized computer;
- Comprehension by applied programmers of the necessity of documentation and easiness of its preparation, correction, and copying in the new environment;
- Practically unbounded possibilities for the development of the technology. That was surprisingly quickly adopted by applied programmers and provided a back-feed of ideas and suggestions.

Currently Leningrad group has a wide experience of using Algol 68 in various application areas. The compiler A68LGU which is used as the implementation tool has quite satisfactory characteristics of reliability, compilation time and object code quality. However, recently the authors of Algol 68 proposed new interesting extensions of the language concerning modularity, separate compilation and exception handling. On the other hand, it turned out that the A68LGU compiler does not fit well for incorporating of new technological tools, that were not foreseen from the design point, e.g. debugging in terms of source text. In the course of long exploitation of the compiler some other minor drawbacks were discovered (too narrow range of integer, for example) as well as more serious errors such as incorrect memory allocation in some cases. This lead to the decision to design a new programming system, which was called WBC.

The distinguishing features of the new programming system are integrity and interactive style of work. It has special means for configuration control and large project development support. Yet another specific of the WBC system is its simultaneous orientation on several computers (IBM mainframe, DEC architecture, CAMCOH, PS 1001, PC compatibles, and some specialized computer). Several cross-compilers were realized on the basis of the A68LGU compiler. There was an experience of porting the compiler to different computers. The very fact that the greater part of the compiler is written in Algol 68 provokes an idea of its portability. However, the real porting turned out to be much more complicated. It was necessary to reorganize the whole stucture of the compiler and its dynanic environment, to specify precisely parts which depend on hardware or on operation system, to unify the mechanisms of communications. The interface with compiler tables was specified so that various units implementing the interface were possible even on one and the same computer depending on the objectives of a particular compiler.

All compilers of the WBC systems have the following common components and features:

- syntax of the intermediate languages;
- stucture of compiler tables and access procedures;
- programs of mode independent and mode dependent analyses, fragments of optimization phase, listing generation, debugger, and monitor;
- algorithms for memory and register allocation;
- technique of code generation;
- run-time support, including I/O procedures;
- the way to choose variants of compilation for the language constructs;
- Algol 68 as the implementation language.

Such unification makes the system open for extension to other computers, and the progress on the way from IBM mainframe to DEC architecture, to CAMCOH, to IBM PC, etc., can be a justification for that.

References

[1] *Report on the Algorithmic Language Algol 68*, Russian translation by A.A.Baers, A.P.Ershov, A.F.Rar and L.L.Zmievskaya. "Kibernetika", Kiev, Part 6 of 1969 and Part 1 of 1970.

[2] A.A.Baers, A.P.Ershov, A.F.Rar. *On Description of Syntax of Algol 68 and its National Variants.* In: "Algol 68 Implementation", edited by J.E.L.Peck, N.H. Publ. Co., Amsterdam-London, 1971.

[3] *Revised Report on the Algorithmic Language Algol 68.* A. van Wijngaarden, B.J.Mailloux, J.E.L.Peck, C.H.A.Koster, M.Sintsoff, C.H.Lindsey, L.G.L.T.Meertens and R.G.Fiskers (Eds.). Russian translation by A.A.Baers, "MIR" Publishers, Moscow, 1980.

[4] C.H.Lindsey and S.G. van der Meulen. *Informal Introduction to Algol 68.* Russian translation by L.Leifman, "MIR" Publishers, Moscow, 1977.

[5] F.G.Pagan. *A Practical Guide to Algol 68.* Russian translation by A.F Rar, "MIR" Publishers, Moscow, 1979.

[6] A.N.Maslov. *Algol 68. Structure of Programs*, Moscow State University Publishers, 1978. (In Russian.)

[7] V.A.Vasilyev, The Language Algol 68. Basic Concepts. "Nauka" Publishers, Moscow, 1972. (In Russian.)

[8] G.Deykalo et al. *New Programming Instruments for ES EVM.* "Finansy i Statistika" Publishers, Moscow, 1984. (In Russian.)

[9] A.P.Ershov. *A Multilanguage Programming System Oriented to Language Description and Universal Optimization Algorithms.* In: "Algol 68 Implementation".

[10] L.A.Zakharov, S.B.Pokrovsky, G.G.Stepanov, S.V.Ten, *A Multilanguage Compiling System.* Computing Centre of the Siberian Division of the USSR Academy of Sciences, Novosibirsk, 1987. (In Russian.)

The Design of Elegant Languages

Lambert Meertens

Department of Algorithmics and Architecture, CWI, Amsterdam, and
Department of Computing Science, Utrecht University, The Netherlands

0 Introduction

It was a dark and stormy week in Munich, the third week of December 1968, in which IFIP Working Group 2.1 decided to submit MR 100 as "the consolidated outcome of the work of the Group". MR 100 was the document describing the design of the Algorithmic Language ALGOL 68 [4], after many iterations, with Aad van Wijngaarden's MR 76 [3] as the starting point.

The weather outdoors was fair for the time of the year, a crisp cold; but darkness had descended upon the hearts of the Working Group, and storms were raging in the hall of the *Bayerische Akademie der Wissenschaften* where WG 2.1 was assembled. A substantive minority of the members had strong criticism of the outcome of the whole enterprise; so strong in fact, that it was apparently impossible to discuss the alleged shortcomings of the proposed language[0] in a technical way, let alone suggest improvements that might result in a design that could have found grace in their eyes[1].

The criticism was laid down in a Minority Report, signed by Edsger Dijkstra, Fraser Duncan, Jan V. Garwick[2], Tony Hoare, Brian Randell, Gerhard Seegmüller, Wład Turski and Mike Woodger.

The following is quoted from the Minority Report.

> Now the language itself, which should be judged, among other things, as a language in which to *compose* programs. Considered as such, a programming language implies a conception of the programmer's task. [...] More than ever, it will be required from an adequate programming tool that it assists, by structure, the programmer in the most difficult aspects of his job, viz. in the *reliable creation* of sophisticated programs. In this respect we fail

0. There was also strong criticism of the "how" of the description itself next to the "what" of what was described, but I shall leave this aside.
1. In the interest of fairness it should be pointed out, however, that Fraser Duncan had before produced proposals for more important changes, and although these were not followed he remained very active up to the end in reporting technical deficiencies and suggesting minor improvements, most of which were indeed incorporated.
2. Garwick was actually not present at the meeting, but requested afterwards that his name be included among the signatories. Also not present was S.S. Lavrov, who in fact had drafted an early version of the Minority Report.

> to see how the language proposed here is a significant step forward: on the contrary, we feel that its implicit view of the programmer's task is very much the same as, say, ten years ago. This forces upon us the conclusion that, regarded as a programming tool, the language must be regarded as obsolete.

What I set out to do here next, is to look at programming languages from a conception of the programmer's task and deal with some aspects in the evolution of programming languages viewed, specifically, as languages in which to *compose* programs. The treatment reflects largely my personal experience and taste in programming, and as such will not at all be comprehensive. In doing this I shall pay particular attention to ALGOL 68. It is, however, not my aim to give a "critical but balanced" assessment of this language. Also, I will freely ascribed "innovations" to language B, even though it may be argued that the essence of the idea existed before in language A, if B was the first to do it right, or with sufficient generality.

1 Limitations of the human mind

Except for rare cases, programs are not written in machine language, but in some programming language. Nowadays one important aim of using a (commonly available) programming language is to achieve program portability. This is an aspect that I shall not consider here. I want to look at programming languages here purely as languages in which to *compose* programs.

As such, a programming language is undeniably a tool, and, following the Minority Report, we can require of an adequate programming tool that it assists, by structure, the programmer in the most difficult aspects of his job, viz. in the *reliable creation* of sophisticated programs. It is not necessary to give a detailed analysis of the programmer's task to agree that the statement that this is difficult (and difficult it is) is also a statement about limitations of the human mind. If we had no such limitations, we would not need computers or programs to start with.

The main limitations we have that are relevant here are probably the following three.

In the first place, human long-term memory is bad for remembering "meaningless" things, like telephone numbers or nonsense text. If we do not use or at least recall something like that daily, we tend to forget it. For meaningful things that we do recall, we often make substitution errors. For example, we may recall the phrase "Well, he's probably pining for the fjords" as "He's pining for the fjords, you know".

Secondly, the amount of things that we can mentally handle *simultaneously* is severely limited. As an example, try (without writing down intermediate results) substituting simultaneously $a+b$ for each occurrence of a and, likewise, $a-b$ for each occurrence of b in the formula $(2a+b)(a-2b)-(a+b)(a-3b)$ while at the same time expanding the result by "multiplying it out"[3].

Finally, when we have to perform some routine task repeatedly, we lose attention

3. For those who want to try this, here is a sin le check on the result. Evaluate the resulting formula for $a=2$ and $b=1$. If the outcome is not 7, something went wrong.

after some time and start making the silliest clerical errors imaginable, precisely in those things we understand and know perfectly well.

In the best case, a programming language helps the programmer to cope with the task of program construction by offering ways to get around such limitations. A simple example is the programmer's ability to choose meaningful identifiers, such as ''*vector*'', ''*velocity*'' and ''*version*'', instead of ''*V*'', ''*V1*'' and ''*V2*''. In a program with hundreds and hundreds of identifiers — not at all uncommon — this is of inestimable help. In the worst case, programming is a struggle with the programming language itself, a struggle in which more mental effort is spent in trying to cope with the intricacies and idiosyncrasies, if not idiocies, of the language, than on the actual problem.

2 The programmer's task

Part of the difficulty of programming, and sometimes the most difficult part, may be to decide what the program is to do in the first place, rather than how this is to be done. For example, in creating a good code generator, the hard task is to decide and specify what code it will generate; after that the actual ''coding'', although perhaps not entirely trivial, is definitely only a small part of the whole problem.

Inasmuch as I have witnessed grandiose failures of software projects, fortunately usually from a comfortable distance, these were always foreshadowed by a failure to get a clear position on the ''what'', or even to reach agreement between the actors involved as to the basic objectives of the project.

Various kinds of formalisms can help to record the decisions taken, if any, but as far as I am aware they tend not to be particularly useful in reaching these decisions, and I see no clear role here for what I consider to be programming languages.

Deciding on the ''what'' cannot be entirely separated from the ''how'': sometimes a small change in the specification that is almost irrelevant from the point of view of the user of the program may be the difference between entirely feasible and entirely infeasible. For example, in code optimization, there are fast and reasonably simple techniques that give very good register allocation, but for obtaining a truly optimal allocation — only marginally better than very good — the fastest algorithms we have may easily take more time than one could reasonably hope to gain by the optimization.

Let us, however, assume that the ''what'' is given. The next step is to go from the ''what'' to the ''how'', which typically involves designing, jointly, suitable data structures, and algorithms operating on data encoded in terms of such structures[4]. In my experience, this part is only rarely a difficult job. Typically I see ''immediately'' some obvious approach. Usually there is an ''obvious'' decomposition into sub-problems for which rather standard data structures and straightforward algorithmic techniques will do the job. (It helps, of course, to have some knowledge of and experience with such techniques.) In the rare cases in which the approach is not obvious, there is at least the satisfaction of an intellectual challenge.

So where we are now is that in principle the algorithms to be used, including the relevant data structures, are sufficiently clear in the programmer's mind, and that the task

4. I am not considering the design of distributed programs, which requires a very different approach.

at hand is to cast these abstract but clear ideas into the form of a program; to create a concrete embodiment of the algorithms and data structures by means of a specific programming language. It is — still in my experience — here that suffering starts, and that that which was so clear becomes obscure, if not a mess.

As a first step a mapping of the abstract data structures has to be given in terms of the available data types of the programming language. The mapping has to be such that the basic operations in terms of which the abstract algorithm is formulated can be implemented efficiently. This may involve explicit allocation and deallocation, requiring extensive administration to keep track of what is being used when by whom.

The implementation of the basic operations can be viewed as programming tasks on their own, creating as it were an abstract machine on top of which the abstract algorithm is implemented. Ideally, the effect is that the programming language is extended with new data types, in a way as if the language had been designed with these data types built-in from the start. In the point of view in which a program is seen as emerging from its proof, this separation corresponds to a factorization of the proof by separation of concerns. Whereas this was a bottom-up phase, expressing the abstract algorithm in these primitives is largely a top-down process, corresponding to the (possibly repeated) decomposition of the problem into sub-problems.

Now where do the problems come in? What is it that makes this so difficult? Let me sketch the worst-case scenario. Assume that the language is such that no suitable interface can be created between the "abstract machine" and the implementation of the abstract program because its abstraction and encapsulation mechanisms are too deficient. This means that the concrete implementation of the abstract operations has to be spelled out again and again. The inbuilt data types are weak, so that this implementation is complex and involves awkward bookkeeping, which has to be woven through the program under construction. So all the time the programmer has to keep two different abstraction levels in mind, each with their own meaning, representations, and invariants. Moreover, the programming language offers little textual support for composing programs from sub-programs solving sub-problems, so that the programmer also has to keep track of where the decomposition is in the traversal of a virtual tree, and this possibly again simultaneously at different abstraction levels. This is by no means the end of the suffering of our poor programmer. By the time this process reaches the point where a construct from the programming language can actually be used (pant pant), the question arises what its concrete syntax is. What was its name again? Was this keyword abbreviated or not? Are the separators here commas or semicolons? What was the order of the parameters? Then, are there perhaps restrictions that apply here? Or some semantic exception?

With this scenario the programmer has to keep, all the time, a large amount of detail in mind, while continually retrieving confusing items from long-term memory, in order to perform a rather mindless task, the repeated expansion of substituting the concrete implementation in the abstract operations, and weaving through that the bookkeeping code.

Almost impossible. And yet this is what most programmers do all the time. For most programming languages, and certainly those that have a good deal of currency, suffer to some extent from almost all of these problems.

3 On elegance

What makes some designs more elegant than others? In general, when we call a design elegant, we mean that the design displays ''good taste'', both in the choice of the elements of which the design is composed and in the way they are combined. So elegance is — to a certain extent, but undeniably — a matter of taste. A well-known saying tells us that there is no accounting for taste, and indeed, discussing such an elusive æsthetic judgment as elegance in an academic context is a somewhat precarious enterprise. Yet, as I shall argue, elegance in programming-language design is a difficult but important aim.

The notion of ''elegance'' can be clarified somewhat further by some reflection on what we would, definitely, consider inelegant. Something can be inelegant because of ''too-muchness'': when the design is suffering from an excess of elaboration, with too many frills. Inelegance can also be due to a lack of balance, which can be defined as a local ''too-muchness''. Finally, a design can display bad taste in the incompatibility of its components, for example in style.

An elegant design, then, is one that is characterized by the apparent simplicity with which its effect is obtained, evidenced by a certain restraint in the choice of elements, in number as well as in style: no individual part may give the impression it is superfluous, and the overall design should give the impression of a conceptual unity.

Most programming-language designers will argue — or so I expect — that simplicity is a desirable property of programming languages, and that their own pet language is simple. Now there is simple and simple. A knife is simpler than a pair of scissors. But clipping an article from a newspaper using scissors is a simpler task than cutting it out with a knife. It is meaningless to apply a notion like simplicity to a programming language without reference to its use as a tool for constructing programs, and without considering in particular the nature of that task in relation to the limitations of the human mind.

An essential aspect of that task is that the ''features'' of a language are not *individually* used as tools the way a carpenter uses a plane and then puts it aside to use next a chisel, and so on. It is the very act of combining various elements by which the program is constructed.

From the above discussion of the programmer's task, we can see that all kinds of ''bells and whistles'' do more harm than good, and that, more than many ''ready-made power features'', we as programmers need a careful choice of elements that lend themselves to easy and graceful combination. An important part of that is that the rules for what may be combined when and how are easy. Of paramount importance for composability is the presence of abstraction mechanisms by which interfaces between abstraction levels can be created.

We see thus how the element of elegance comes in. A language in which these things come together nicely will be felt by its users to be elegant. A somewhat different viewpoint is that in which programs are constructed by deriving them formally. However, the difference is more apparent than real. Precisely the same properties are required to make this formal activity doable, and in fact even more so.

4 Before ALGOL 60

The evolution of programming languages, already before ALGOL 60, has been away from the low-level model provided by hardware architectures, creating instead a more elegant abstract machine model. This evolution started with so-called assemblers, which are traditionally viewed as giving a direct mapping to machine code but with some unpleasant chores having been taken over, and with mnemonics to aid the programmers. However, the difference between assembly languages and higher-level languages is not necessarily a matter of principle. For example, in C the underlying hardware with its linear memory model as a contiguously addressable sequence of words keeps staring us in the face. Conversely, assembler languages may have nice abstraction mechanisms that are a valuable help in programming. It is entirely possible than to program as if the language was high-level.

Already there we find that details of the mapping to the concrete machine that are not relevant are taken away from the responsibility of the programmer. A quantity has to be stored somewhere, and the same location should not be used for something else during its lifetime, but which actual location is chosen is completely irrelevant. The higher the language level, the more such irrelevant things tend to get hidden. The main innovations of FORTRAN were an explicit parameter mechanism, automatic mapping of more-dimensional arrays to linear memory, and—whence its name—"formula translation".

5 The contribution of ALGOL 60

Suppose we have to write a program for some task *T*, which is too complex to be expressed directly as a basic step. However, whenever a certain condition *C* is satisfied, the task *T* can be reduced to the simpler task *T0*, whereas otherwise it can be simplified to *T1*. So, assuming we have programs for *T0* and *T1*, we can now give one for *T*. In FORTRAN such as it was when ALGOL 60 was being designed, conditional execution was achieved by using "conditional jumps" in the program. This language construct was modelled rather straightforwardly after the same low-level machine-code instruction on the IBM 704. Using ALGOL 60 syntax style, we get then:

```
      if not C then goto L1;
      T0;
      goto L2;
L1:   T1;
L2:
```

If *T0* and *T1* are still too complex and are likewise decomposed into subtasks, and so on, we arrive at the third step at the following:

```
        if not C then goto L1;
        if not C0 then goto L01;
        if not C00 then goto L001;
        T000;
        goto L2;
L001:   T001;
        goto L2;
L01:    if not C01 then goto L011;
        T010;
        goto L2;
L011:   T011;
        goto L2;
L1:     if not C1 then goto L11;
        if not C10 then goto L101;
        T100;
        goto L2;
L101:   T101;
        goto L2;
L11:    if not C11 then goto L111;
        T110;
        goto L2;
L111:   T111;
L2:
```

The result looks like spaghetti but is rather more unpalatable. It is far from easy to see from this text under what conditions exactly *T101*, for example, will be executed. It is, of course, not accidental that this resembles compiled code. It *is* compiled code: hand-compiled from the abstract idea into to the limited constructs of a programming language.

A major innovation of ALGOL 60 was that specific program-composition constructions were provided for the common cases of task decomposition. For conditional execution ALGOL 60 has

```
if C
then T0
else T1
```

With this notation, the above spaghetti program becomes:

```
if C
then if C0
        then if C00
                then T000
                else T001
        else if C01
                then T010
                else T011
else if C1
        then if C10
                then T100
                else T101
        else if C11
                then T110
                else T111
```

The advantage should be clear.

As a result, the definition of what a permissible program is got a recursive nature. To describe this exactly, the authors of the ALGOL 60 Report had to invent[5] a new grammatical formalism, which became known as BNF. Compiler writers tend, for reasons of their own, to like silly restrictions that make the programmer's life harder, such as: identifiers may have only seven characters; or: procedures may not be nested inside other procedures; or: expressions may only be nested five levels deep and contain at most 511 subexpressions; and so forth. In doing serious programming, programmers keep running into such restrictions, and getting around them may take more than half of the coding effort, if it is possible at all. The total amount of time wasted this way is several orders of magnitude more than the time gained by the lazy compiler writer. If the syntax of a language is described by "verbal prose", such restrictions are easy to put in. Using BNF, it is easier to describe a language without such arbitrary restrictions than with. This is, indeed, what happened with ALGOL 60.

6 Some problems with ALGOL 60

The syntactic generality of ALGOL 60 made it, decidedly, a much more elegant language than FORTRAN. The majority of its authors had a background in numerical mathematics. This shows in the absence of any inbuilt facilities to compute with texts, or with any other kind of structure than vectors and matrices of numbers. By the mid sixties, it was evident that this was a serious deficiency; non-numeric computing had become at least as important as numeric computation.

Some facility was needed by which programmers could add their own types. This required, of course, generalizing constructs to arbitrary types. In ALGOL 60 the types were given explicitly in the syntax rules. There was a rule for a conditional integer

5. More properly: to re-invent. Chomsky had before described context-free grammar in a linguistic context to characterize his "Type 2" languages (now generally known as context-free languages).

expression, and also for a conditional boolean expression, but not, for example, for a conditional string expression.

Another problem with ALGOL 60 had to do with type checking. The following is a program, written in ALGOL 60, that contains a type error:

```
begin procedure a(b, c, d, e, f, g);
        b(c, d, e, f, g, a);

        procedure s(t, u, v, w, x, y, z);
        z(s, t, u, v, w, x, y);

        a(a, a, a, a, a, s)
end
```

The procedure-call rule of ALGOL 60 results in the following call sequence:

```
        a(a, a, a, a, a, s)
        a(a, a, a, a, s, a)
        a(a, a, a, s, a, a)
        a(a, a, s, a, a, a)
        a(a, s, a, a, a, a)
        a(s, a, a, a, a, a)
        s(a, a, a, a, a, a)
```

Here *s* is called with six parameters. But *s* requires seven parameters. In general there is no foolproof way to determine in advance whether an arbitrary ALGOL 60 program contains such a type error. This is rather obvious if we only consider "reachable code", since it is not decidable which parts of the program are reachable. One can take a more textual view and require that under repeated replacement of calls by expanded bodies there are no parameter mismatches. But even then the problem is undecidable, as was shown by Langmaack [2].

There are, of course, more shortcomings of ALGOL 60, in particular the conspicuous absence of input/output facilities. Adding these is, however, "merely" a matter of adding. It does not require a really new language. The problems mentioned above could not be solved without creating a new language.

7 From ALGOL 60 to ALGOL 68

Van Wijngaarden's insight was that the required generalization of the syntax rules could be obtained by introducing parametrized grammar rules [3]. Using BNF-like notation, and applying this to ALGOL 60 syntax (somewhat simplified), we see that the rules

⟨conditional boolean expression⟩ ::=
    if ⟨condition⟩ ***then*** ⟨boolean expression⟩ ***else*** ⟨boolean expression⟩

⟨conditional integer expression⟩ ::=
    if ⟨condition⟩ ***then*** ⟨integer expression⟩ ***else*** ⟨integer expression⟩

⟨conditional real expression⟩ ::=
    if ⟨condition⟩ ***then*** ⟨real expression⟩ ***else*** ⟨real expression⟩

can be unified to a single rule

⟨conditional TYPE expression⟩ ::=
    if ⟨condition⟩ ***then*** ⟨TYPE expression⟩ ***else*** ⟨TYPE expression⟩

if "TYPE" can stand for any of "boolean", "real" and "integer". The generalization requires now to describe what "TYPE" can stand for. This description forms the "metalevel" of the grammar. I do not know if Van Wijngaarden was aware of other varieties of two-level grammars, such as affix grammars or attribute grammars, but where these had either a limited or not grammatically specified domain for the metalevel, Van Wijngaarden grammars have for the metalevel a conventional context free grammar.

Just like context free grammar permitted to describe the recursive formation rules for program texts in ALGOL 60, here they allow to give recursive formation rules for the types[6] of ALGOL 68. And, just as for ALGOL 60, this makes it harder rather than easier to have exceptions.

Today it is entirely commonplace that the type system of a language has recursive formation rules. In 1968 it was an innovation; and precisely what is needed to allow the users to define a good interface between the implementation of their "abstract machine" and the implementation of the abstract algorithm.

The main criticism I see for the type system of ALGOL 68 is the low level of the type constructor "***ref***", in particular the coupling of assignability with referring[7]. For more detailed criticism, see Koster [1].

At least as important as the type system *per se* is the fact that ALGOL 68 is strongly typed.

8 After ALGOL 68

It is inevitable that any retrospective activity has some of the power of hindsight. ALGOL 68 was not only the brainchild of van Wijngaarden, but also a child of its time. With our present knowledge of the principles of programming languages, surely the current major programming languages do a much better job. Or do they?

It has been remarked that ALGOL 60 was an improvement over most of its

6. In ALGOL 68 idiom, the term "mode" is used for what is usually called "type". In this article I use the more common term "type".
7. This is especially severe since the obvious way to define "algebraic" types, with recursion through "***union***" and "***struct***", requires the use of "***ref***" for "shielding to **yin**". (see rules 7.4.1 of the Revised Report).

successors[8]. To make a joke in the style of van Wijngaarden, but reflecting my personal opinion: ALGOL 68 would have been an improvement over most of its successors, had it had any.

Is it true that the "implicit view of the programmer's task" underlying the design of ALGOL 68 was very much the same as, say, 1958? Whether this was so or not, ALGOL 68 introduced a view on types that makes programming easier. It is a view that is common now. Unfortunately, not much has been added since, at least not in major available languages. Although C owes much of its type system from ALGOL 68, it is clearly a step back towards the machine level. In particular, it is annoying — to put it extremely mildly — that the allocation and deallocation of dynamic "non-stack" storage is the responsibility of the C programmer, and that the semantics of C gives no support for keeping pointers in check.

A true improvement, in my opinion, is type polymorphism. I am more dubious about the object-oriented paradigm. I've seen no linguistic approach to that seems of an acceptable neatness to me.

9 Final remarks

There is a tension between two viewpoints concerning the relation between language design and program correctness. One viewpoint is that a good language makes it hard to write bad programs. The other viewpoint is that it makes it easy to write good programs. Van Wijngaarden was an outspoken adherent of the latter viewpoint; he considered the first one "paternalistic". Nevertheless, the design of ALGOL 68 is such that many if not most "clerical errors" may be statically detected, and this was already so in the version of 1968.

No language design can really prevent programmers to create an inextricable mess, and a language in which it is really hard to write bad programs probably also makes it hard to write good programs. Nevertheless, some things in ALGOL 68 are error prone (like "***ref***" mentioned before), and I think the language could have been better if more attention had been paid to such problems. The same is true for almost all languages that saw the light since.

For the record, I want to state that — although my name is associated with, in particular, the Revised Report [5] — I do not feel I have, personally, a stake in any evaluation of the design of ALGOL 68. My role has mainly been confined to "debugging" and polishing an existing description of an existing design, and this has had at most a marginal influence on the language as a programmer perceives it. I do feel responsible for remaining errors in the Revised Report (not counting the section on Transput declarations); in particular, being reminded of the painful fact that I overlooked that "**real field letter y integral field letter l**" contains "**yin**" (see rule 7.3.1.c of the Revised Report) still can make me blush.

8. I do not remember to whom this witticism should be ascribed, but Tony Hoare comes to mind as a plausible source. In any case I am pretty sure it was not Niklaus Wirth.

References

1. C.H.A. Koster (1976). The mode system in ALGOL 68. *New Directions in Algorithmic Languages 1975* (S.A.|Schuman, ed.), 125-138, IRIA, Rocquencourt.
2. H. Langmaack (1973). On correct procedure parameter transmission in higher programming languages. *Acta Informatica* **2**, *110-142.*
3. A. van Wijngaarden (1965). *Orthogonal design and description of a formal language.* Mathematisch Centrum, Amsterdam, Mathematical Centre Report MR 76.
4. A. van Wijngaarden (Editor), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster (1968). *Final Draft Report on the Algorithmic Language ALGOL 68.* Mathematisch Centrum, Amsterdam, Mathematical Centre Report MR 100.
5. A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens and R.G. Fisker, Eds (1975). *Revised Report on the Algorithmic Language ALGOL 68.* Acta Informatica **5**, 1-236.

Aad van Wijngaarden and the Mathematisch Centrum
A personal recollection

John Peck
em. University of British Columbia

Preamble

After nine years of retirement, it is surprising to receive an invitation to attend a conference on Computer Science. It is comforting to be told that one is not expected to indulge in deep archival research and that makes it a pleasant task. So do not look for too many facts and do not search for references, for there will be none. You will probably find all these in a paper by Charles Lindsey. Rather we shall try to recall the flavour of the events of the 1960's as they appeared to unfold in the eyes of one of the bit players.

It is probably true to say that I have lost contact with Computer Science over the last nine years, although I have not lost my interest in programming. My current love is HyperCard and HyperTalk on a Mac IIci. I keep myself occupied by doing voluntary programming for my son's factory, my bicycle club and for a local educational office that is trying to encourage the learning of French in British Columbia's school, and all of this on a Macintosh computer in HyperCard. It would not be easy to accomplish these things in ALGOL 68.

It may surprise younger participants here to know that when I went to school, a computer was a person who turned the handle of a mechanical calculator, that television did not exist, and radios were large boxes with pride of place in the living room. There were no transistors, no micro-wave ovens, no fax machines, no computers, no cellular 'phones, and nobody was worried about the environment. In my first job as an analytical chemist, all calculations were done with a slide rule. It is pleasant to have witnessed such vast changes stimulated by science and technology, and by Computer Science in particular. It seems that life has become easier and more enjoyable.

Personal

I was trained as a mathematician. I used to worry about such things as Banach spaces and topological semi-groups, and when I began teaching, no university had a Computer Science department. At McGill University in 1955 a note was circulated in the department of mathematics that Canadair had a Datatron computer and mathematicians were invited to program it. I can well remember my colleagues saying, "why bother with that? It's just a simple application of Boolean algebra." However, I took up the invitation and from then was hooked.

We shall skip the rest, except to say that I wrote a program for the Datatron, then learned to program the IBM 650. In 1959 I moved to Calgary, a booming oil town that had one computer, a Royal McBee, which I learned to program also. In the early '60s, the University of Calgary acquired an IBM 1620, and that then took my fancy. But perhaps it is time to turn to the subject at hand, which is my association with Aad van Wijngaarden, ALGOL 68 and the Mathematisch Centrum.

1966

I first met Aad van Wijngaarden in 1966. It happened this way. I was teaching mathematics at the University of Calgary, and in that year I was due for the first sabbatical leave of my career. I had become interested in computing and had been programming on the IBM 1620. We had attempted an implementation of ALGOL 60 on that machine, but we knew little about how to do it. For example, we did not know how to include recursion. Although we executed a few programs, our implementation was not much more than a curiosity.

We knew that interesting things concerning ALGOL were happening in Europe, so I wrote to a few places. The reply from Aad van Wijngaarden was the most encouraging, so I arrived at the Mathematisch Centrum in September of 1966. It was there that I heard that the Algol Working Group was planning a new ALGOL and that Aad had a preliminary design which he would present at the next meeting in Warsaw.

Aad suggested that someone was needed to write a users manual for the new language, and he invited me to sit with him and Barry Mailloux, another Canadian, to learn something of the new language. I found it very confusing. The language was defined by a new grammatical mechanism which I had difficulty in understanding. Moreover there was a rush to get a document ready for the Warsaw meeting.

I attended the October Working Group meeting in Warsaw as an observer. This was a humiliating meeting for me, because most of the discussion was beyond my comprehension. However I began to absorb new meanings to words such as notion, protonotion, consistent substitution, overloading, extensions, orthogonal design and so on. Also I was able to associate faces with names that were already familiar, such as John McCarthy, Niklaus Wirth, Tony Hoare, Brian Randell, Edsger Dijkstra and others. Aad insisted and received his mandate to continue with the design of the language, known then as ALGOL X. At that meeting I learned of the mandate given to Aad at St. Pierre de Chartreuse and some disturbing news of the Kootwijk meeting where two proposals were supposed to be merged, but their champions could not agree.

Even at that stage it was clear to me that not everyone believed that Aad's vision was the right one. Also his manner in debate was sometimes a little too keen. He would goad his adversary into admitting several aspects of a certain subject, and then tactlessly humiliate him by showing that those views were inconsistent. In this way there was a clash of sharp minds which sometimes left personality wounds, although I was never a recipient of his rapier thrusts. Despite these clashes, Aad was always a delightful dinner companion, with many stories of interesting trivia to entertain us. One day he said "Did you know that Halloween equals Christmas?". I had to plead ignorance, so he wrote on a piece of paper "OCT 31 = DEC 25". I felt rather embarrassed that I took me some time to realise that octal 31 is decimal 25. He also asked "How many pins are there in a new shirt". I said "About six". "No", he said, "there is always one more". He was full of such little jokes and puzzles to keep one entertained

Back at the Mathematisch Centrum I sat in on the design sessions and began to understand something of the defining mechanism. It was not long before I was suggesting changes and my confidence rose when these were accepted. We were working towards a new document, which became known as MR88, to be presented at the Zandfort meeting in May of 1967. At that time there was no thought of using the computer as a document editor. I bought a new portable typewriter with some useful symbols on it, and since I had plenty of time, I became the unofficial recorder of the design sessions, producing a few new pages each morning. These gradually transformed the Warsaw document into MR88. I still have the old pages at home in my personal archives. It was clear that Aad had underestimated the time required for design, so all thought of a users manual was forgotten, since the design was changing almost daily.

1967

It was in Zandfort then that the document acquired three Authors: van Wijngaarden, Mailloux, and Peck. At the end of that meeting, I was voted a member of the Working Group which helped me to obtain travel expenses to subsequent meetings. That year in Amsterdam was a turning point. It shaped the remainder of my career in Computer Science.

In the Fall of 1967 I returned to teaching duties in Calgary and kept in touch with the design activities by mail. How much easier it would have been had FAX transmission been common at that time. The document under preparation was MR93, which appeared in the ALGOL Bulletin in February of 1968.

1968

In May of that year an ALGOL anniversary meeting was held in Zurich and MR93 was attacked for its "obscurity, complexity, inadequacy and length", despite the fact that Lindsey in Manchester and Tseytin in Leningrad, neither of whom were involved in the design, had shown that they could understand it and could explain it to others. To me it was a matter of surprise and disappointment to discover the extent of the antagonism.

Aad spent some time in Chicago, where he had access to an IBM selectric typewriter, the one with the type faces on a "golf ball". You could change the golf ball to get different fonts. He personally typed many pages of the Report, and experimented with using standard and italic to get different effects. When I met him next, he pointed to a full stop (or period) in the text and said "How can you tell whether that stop is standard or italic?". So I replied that there was obviously no difference, since they were both small circles. "No", he said, "the standard stop has orthogonal axes and the italic stop has oblique axes", at which he beamed with pleasure at his little joke.

The next Working Group meeting was in Tirrenia, Italy in June of 1968. I remember little of that meeting except for Fraser Duncan's valiant attempt to replace the two-level grammar description method with a syntactic device of his own. Also we were welcomed by the city of Pisa at the City Hall, where there was a large painting on the wall of a bloody historical battle. Men and animals were depicted locked in mortal combat. Someone whispered aside to me: "Doesn't that remind you of a Working Group meeting?"

I was able to spend the summer of 1968 at the Mathematisch Centrum to help in the preparation of what became MR95. That summer I was in Esztergom, Hungary, for a week. It was a conference of Eastern computer scientists at which I gave several lectures explaining the new ALGOL. I was translated both into Hungarian and into Russian, so things moved slowly. I remember trying to explain the coercion mechanism and that constructs in a strong position could be strongly coerced. Just at that time the Russians moved into Czechoslovakia, which was just across the river. My listeners were more concerned with the news on their radios, than with my lectures, the Poles wondering how they would get home. One of the students told me that the borders had been closed, and that I should start learning Hungarian, which would only take me twenty years, he said. However, someone else must have been listening to the lectures, for he remarked, "Now I understand the coercion nomenclature, you see, Czechoslovakia is clearly in a strong position!" Another incident related to that meeting was that copies of the MR95 document were to be sent to Esztergom for the students. However, at the last minute we discovered that they had been sent to Copenhagen instead. I believe that I made a hurried trip to Copenhagen to retrieve them and then assigned them to the same train that took me to Esztergom. When I arrived in Budapest, it was a relief to find that the documents were there, without being held up by the Iron Curtain. Twenty-two years later I was on a bicycle tour in Hungary and we passed through Esztergom again.

In the Fall a Working Group meeting was held in North Berwick, Scotland where MR95 was presented. It was there that I first met a tall fast-talking visitor who came in from the cold, so to speak, and who explained to the Working Group the language that they were designing. We were delighted to discover that he knew more about the Report than most of the members of the Group. It was Charles Lindsey, who became a firm friend and who was to play a major role in the Revised Report.

The rush was then on to produce the next version, MR100, to be presented at the Munich meeting of the Working Group in December of 1968, the take it or leave it meeting. I arrived in Amsterdam a few days before that meeting. I well remember the rush there was to prepare that document. Changes were being made at the last minute. We spent one whole night reproducing perhaps 50 copies of its 120 or so pages, and were frustrated when the Xerox machine broke down. There was no time to send the copies by any other means than to carry them to Munich ourselves. I remember that all the documents were packed in a large suitcase, which came with us to the airport, and on which over-weight charges were collected by the airline. Books weigh like lead as luggage. As he settled down in his seat on the plane, Aad called for extra drinks all round to calm us down, after the hectic activity.

The Munich meeting was dramatic. At first there was some technical discussion on the report, followed by a grudging acceptance. Then there was discussion about a covering letter to go to TC2 indicating that not everyone agreed. However, at the last moment a minority report made its appearance. I well remember the breathless tension in the room. First Edsger Dijkstra and then Brian Randell strode up and down, behind their seats, explaining in measured tones that they believed that a new language was not the way to go. What was needed, they said, was a new approach to programming instead. I thought about all the work that Aad and his three helpers had done, how Aad had more or less staked his academic reputation on the acceptance of his design, and what a blow it would be to him if this minority report were to destroy the credibility of his creation. I thought about politicians who have worked hard for their country, only to be driven out of office by a fickle electorate. How does one recover from a blow such as that? It cannot be easy. Most Algol Working Group meetings were "cliff hangers", i.e., you never knew whether an important decision would go the way you wished it. Your reputation was on the line. You could either feel humiliated or exhilarated, but the final vote only came at the last moment.

In the end the ALGOL 68 report was accepted for transmission to Technical Committee 2, together with a covering letter and the Minority Report. TC2, for reasons of its own, sent the Report and a modified covering letter to IFIP, but not the Minority Report.

So these were the birth pangs of the new language. Its first publication was as MR101, and other publications soon followed. These were known as the "Report on the Algorithmic Language ALGOL 68" and must be distinguished from the "Revised Report ..." which came later.

Can one now look back on these years and suggest that it might have been otherwise? I feel that perhaps a fundamental mistake made by WG 2.1 was to believe that a new programming language could be designed and readily accepted without insisting that at least one implementation should exist. Of course, if they had insisted on an implementation, then the design process would have taken much longer, perhaps twice as long, so it was a gamble that almost paid off. However, with a concurrent implementation, the design might well have been better, as we soon discovered when the first implementation appeared.

Implementations

1969 was the year of publication of the ALGOL 68 Report as MR 101 and elsewhere, its translation to other languages, including Russian, and the year that I was invited to head the new Department of Computer Science at the University of British Columbia. The Working Group was to meet in Banff, Alberta in the Fall, so we took the opportunity to arrange a quick informal implementation conference in Vancouver the week before. We next saw Aad in Banff. Since there were two members of the editorial group, Barry Mailloux and myself, in Western Canada, it was natural that WG 2.1 should meet there at least once, and Banff was an obvious choice.

The informal implementation conference revealed that not much had been done yet except to think about the problem. The Banff meeting looked at some necessary errata for the Report, discussed some problems of implementation, and talked about what WG 2.1 should do next. Compared to the previous meeting in Munich, it was somewhat low key, since most of those who had disagreed with the Report had already resigned, to form a new working group on Programming Methodology.

In 1970 a formal ALGOL 68 implementation conference was held in Munich. We had confidently expected that the Munich group would be the first implementers of the new language, but it was not to be. To the surprise of most, Ian Currie appeared to tell us of an implementation at the Royal Radar Establishment in Malvern, England of what he called ALGOL 68-R. It was not a faithful implementation, a major discrepancy being the omission of the proceduring coercion, but it was significant. It taught us that perhaps the design should be amended to make implementations easier.

I was asked to edit the proceedings of that conference. The book arrived from North-Holland when I was with Aad. He opened it proudly at random smiling in appreciation at the work I had done. Suddenly he was convulsed with laughter. I felt terribly embarrassed and wanted to sink through the floor. What was it that he could be laughing about? Then he showed it to me. In a paper by Branquart and Lewi, there was reference to Randell and Russell's book on ALGOL 60 Implementation, except that the printer's devil had been at work and it appeared as ALGOL 68 Implementation. But we all knew that Brian Randell was an outspoken opponent of ALGOL 68 and one of the instigators of the Minority Report. Aad was a superb proof reader, he could spot a printing error, as a hawk does a mouse, faster than anyone, and it was unlucky for me that he had opened the book at page 231, where the devil had been at work.

A Revised Report

1971 saw the Working Group at Novosibirsk, where a decision was taken to produce a Revised Report on ALGOL 68. Soon after this I made an application to NATO for a grant to help us with the task of revision. I was careful not to tell NATO that the Working Group was truly international and had members from several countries behind the Iron Curtain. Anyway, our application was successful and the modest $8000 provided allowed us to bring the editors together, first in Vancouver and then in Edmonton, for extended design sessions. But this was when Aad began to bow out of the working sessions. He remained only a fatherly figure remotely in Amsterdam. The real brain-storming was done by Charles Lindsey, Michel Sintzoff and Lambert Meertens.

More sophisticated tools were used to produce the Revised Report. First we had the remarkable advances made by Sintzoff, who gave us the NEST syntax, a device which enabled us to include the identification of identifiers in the syntax. Second we committed the Report and its syntax to computer files which enabled us to edit the copy and produce new versions with ease. This also meant that the Revised Report was eventually printed directly from computer files. I no longer had the role of typing new pages on my typewriter.

But perhaps this is where my story should end. As I have indicated above, much of the work of revision was done in Western Canada and Aad and the Mathematisch Centrum were far away. It is sufficient to remind you that the Revised Report was eventually accepted by the Working Group at a meeting in Los Angeles at the end of 1973.

China

In 1987 I was on a bicycle tour in China and found myself in Beijing. I contacted Lu Ruqian at the Academy of Scientists and went to visit him there. This was before the Tienanamen Square massacre. Lu Ruqian is the translator of the ALGOL 68 Report into Chinese. Apart from discussing translation problems, I asked him what he did during the cultural revolution. He said that he was sent to the country to herd buffalo. It remarked that it must have been good for his health. “Yes”, he said, “but not much good for my mind.”

Conclusion

I have already mentioned that ALGOL 68 dominated my career in Computer Science. I greatly respected Aad van Wijngaarden, the brilliance of his mind and the kindness of his disposition. I also look back with pleasure at the many days spent at the Mathematisch Centrum. The design of ALGOL 68 was an intellectual challenge that I enjoyed immensely.

The History of ALGOL 68

conference on 25 years of ALGOL 68

Held on the occasion of the 47th anniversary of CWI
Amsterdam, February 11th, 1993

Program

Registration and coffee from 09:00

09:30	Welcome by *Cor Baayen*, scientific director CWI
09:50	*F.L. Bauer* (Technische Universität München)
	History of Programming languages, a Survey
10:30	*C.H. Lindsey* (Dept. Comp. Sc.; Manchester)
	The History of ALGOL 68; Part I: general

Coffee at 11:10

10:30	*C.H. Lindsey* (Dept. Comp. Sc.; Manchester)
	The History of ALGOL 68; Part II: details
12:10	*C.H.A. Koster* (KU Nijmegen)
	The Making of Algol 68

Lunch at 12:40

14:00	*S.G. van der Meulen* (RU Utrecht)
	An orthogonal first programming language
14:20	*A. Rar, M. Bulyonkov* (Novosibirsk), A. Terekhov (St. Petersburg)
	ALGOL 68 - 25 Years in the USSR
14:50	*Maurice V. Wilkes* (Cambridge UK)
	The Impact of ALGOL 68 at Cambridge
15:00	Panel Discussion on the impact of ALGOL 68

Teatime is 15:30

15:50	*L.G.L.T. Meertens* (CWI)
	The Design of Elegant Languages
16:30	*J.E.L. Peck* (Univ. of British Columbia)
	Aad van Wijngaarden and the Mathematisch Centrum,
	A personal recollection

Reception at 17:10